22.11.2016

*Чытаю ў турба рэжыме*

*p. 2*

*С+11 і С+14 – auto type declarations, move semantics, lambda expressions, concurrency support. Асноўныя канцпэтуальныя ўласцівасці мовы.*

*Прадмет кнігі: карэктнае выкарыстанне новых магчымасцяў С++.*

*Property: Quality of software – correct, efficient, maintainable and portable.*

*Concept of book: software.*

*Type of book: practical book which explain how to use new concepts of C++ correctly*

*Author personalitate: PhD*

*p. 3*

*book structure: Praise, proving authority.*

*Citation: “*Finding utmost expertise is hard enough. Finding teaching perfectionism— an author’s obsession with strategizing and streamlining explanations—is also difficult. You know you’re in for a treat when you get to find both embodied in the same person. Effective Modern C++ is a towering achievement from a consummate technical writer.

It layers lucid, meaningful, and well-sequenced clarifications on top of complex and interconnected topics, all in crisp literary style. You’re equally unlikely to find a technical mistake, a dull moment, or a lazy sentence in Effective Modern C++. —Andrei Alexandrescu

*” M: theory of explanation and representation.*

*p. 4*

*М: book structure, citation: “*Effective Modern C++ is a great tool to improve your modern C++ skills. Not only does it teach you how, when and where to use modern C++ and be effective, it also explains why. Without doubt, Scott’s clear and insightful writing, spread over 42 well-thought items, gives programmers a much better understanding of the language. —Bart Vandewoestyne

*”*

*p. 13 Book stracture: From the publisher*

*Aspects of intelligence own: “*This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you’re reproducing a significant portion ofthe code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O’Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product’s documentation does require permission.*”*

*p. 15*

*Book structure: Acknowledgments*

*M: dynamics of thinking: “*I started investigating what was then known as C++0x (the nascent C++11) in 2009. I posted numerous questions to the Usenet newsgroup comp.std.c++, and I’m grateful to the members of that community (especially Daniel Krügler) for their very helpful postings. In more recent years, I’ve turned to Stack Overflow when I had questions about C++11 and C++14, and I’m equally indebted to that community for its help in understanding the finer points of modern C++. In 2010, I prepared materials for a training course on C++0x (ultimately published as Overview of the New C++, Artima Publishing, 2010). Both those materials and my knowledge greatly benefited from the technical vetting performed by Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober, and Anthony Williams. Without their help, I would probably never have been in a position to undertake Effective Modern C++. That title, incidentally, was suggested or endorsed by several readers responding to my 18 February 2014 blog post, “Help me name my book,” and Andrei Alexandrescu (author of Modern C++ Design, Addison-Wesley, 2001) was kind enough to bless the title as not poaching on his terminological turf.*”*

*M: Content: author tells about origins of information. Many of them are from community.*

*p. 17*

*M: representationTheory: Problems of code representation: “*My decision to limit the line length in code displays to 64 characters (the maximum likely to display properly in print as well as across a variety of digital devices, device orientations, and font configurations) was based on data provided by Michael Maher.*”*

*M: Content: acknowledgments follow.*

*p. 19*

*content: Introduction*

*M: evolution of sisved: “Idiomatic changes occurred in new C++ standards”. Raznovidnost of concept “changes”.*

*M: C++ conceptual: “*0 and typedefs are out, Nullptr and alias declarations are in. Enums should now be scoped. Smart pointers are now preferable to built-in ones. Moving objects is normally better than copying them*” M: thinking: to better understand.*

*M: book purpose: “*It’s devoted not to describing the features of C++11 and C++14, but instead to their effective application.*”*

*M: C++ conceptual: “*Want to understand the various forms of type deduction? Or know when (and when not) to use auto declarations? Are you interested in why const member functions should be thread safe, how to implement the Pimpl Idiom using std::unique\_ptr, why you should avoid default capture modes in lambda expressions, or the differences between std::atomic and volatile? The answers are all here.*” M: thinking: it is necessary to better understand.*

*M: thinking theory: types of thinking: “*The Items in this book are guidelines, not rules, because guidelines have exceptions.*”*

*M: technique of using book: “*The most important part of each Item is not the advice it offers, but the rationale behind the advice. Once you’ve read that, you’ll be in a position to determine whether the circumstances of your project justify a violation of the Item’s guidance.*”*

*p. 20*

*M: book goal: “*goal of this book isn’t to tell you what to do or what to avoid doing, but to convey a deeper understanding of how things work in C++11 and C++14.*” Comment: three universal structure aspects of understanding. The first two – 6-7 level. The last one concerns 4-7 level.*

*M: thinking theory: “when we absolutely understand topic we understand it only on 4-7 l. 1-3 l. don’t exist because we understand all. I haven’t met this in my thinking practice. Usually because of necessity to remind we here and now can be on 1-3. Because we don’t remember.”*

*M: content: terminology and conventions.*

*M: aspects of understanding each other: purpose of concept conventions: “*To make sure we understand one another, it’s important to agree on some terminology, beginning, ironically, with “C++.”

*M: book content: history of C++.*

*M: concepts relations: C++14 is superset of C++11. M: thinking: it is necessary to better understand.*

*M: book concepts: C++, C++98, C++11, C++14.*

*M: evolution of C++: “*As a result, I might say that C++ places a premium on efficiency (true for all versions), that C++98 lacks support for concurrency (true only for C++98 and C++03), that C++11 supports lambda expressions (true for C++11 and C++14), and that C++14 offers generalized function return type deduction (true for C++14 only).*”*

*M: concept: move semantics (M: thinking: to better understand).*

*M: concept definition: rvalues -* indicate objects eligible for move operations; *lvalues – don’t.*

*M: concept: move operations (thinking: it is necessary to better understand).*

*M: concept “rvalue” structure:* In concept (though not always in practice), rvalues correspond to temporary objects returned from functions, *M: concept “lvalue” structure: “*lvalues correspond to objects you can refer to, either by name or by following a pointer or lvalue reference. *“; “*A useful heuristic to determine whether an expression is an lvalue is to ask if you can take its address. If you can, it typically is. If you can’t, it’s usually an rvalue.*”*

*M: C++ structure: relations between concepts “rvalue”, “lvalue”, “type of an expression”:*

*M: C++ structure: heuristic rule: “*A nice feature of this heuristic is that it helps you remember that the type of an expression is independent of whether the expression is an lvalue or an rvalue.*”*

*M: C++ structure: underwater rock: “*It’s especially important to remember this when dealing with a parameter of rvalue reference type, because the parameter itself is an lvalue*” Thinking: it is necessary to better understand. Explanation: on page 21.*

*p. 21*

*M: concept: move constructor: thinking: it is necessary to better understand. Comment: concept was introduced in C++11.*

That code snippets… Page. 21.

*25.11.2016*

*M: concept “lvalue” structure: all parameters are lvalues.*

*M: book structure: conventions and designations.*

*M: programming structure: concept “class” structure: it is user-defined type. There are types included in programming language by default. Libraries are something average.*

*M: C++ structure: concept: move operations: structure: i.e., move constructor and move assignment operator. M: thinking: It necessary to better understand last two concepts.*

*M: representation theory: goal of concept “text formatting”: “I apply special formatting to pats of code or parts of comments to draw your attention to them.” Representation theory structure: how to orient attention of person.*

*M: C++ structure: concept “variadic templates”. It is necessary to better understand.*

*M: C++ structure: interchangeability: keywords typename and class can be interchanged and are preferences of programmer. But standard recommends to declare type parameters by keyword class.*

*P. 22*

*M: C++ structure: copy can be cope-constructed and move-constructed.*

*M: move construction structure: std::move*

*M: C++ structure: connection between concepts copy, rvalue and lvalue: “Copies of rvalues are generally move constructed, while copies of lvalues are usually cope constructed”.*

*M: C++ code analysis: “*An implication is that if you know only that an object is a copy of another object, it’s not possible to say how expensive it was to construct the copy. In the code above, for example, there’s no way to say how expensive it is to create the parameter w without knowing whether rvalues or lvalues are passed to someFunc.*”*

*M: C++ code analysis: definitions of function arguments and function parameters. Demonstration of difference between them: “*In the first call to someFunc above, the argument is wid. In the second call, the argument is std::move(wid). In both calls, the parameter is w.*”*

*M: C++ nuance: “*The distinction between arguments with which they are initialized may be rvalues or lvalues*”. Nuance importance in the context of perfect forwarding: “*This is especially relevant during the process of perfect forwarding, whereby an argument passed to a function is passed to a second function such that the original argument’s rvalueness or lvalueness is preserved. (Perfect forwarding is discussed in detail in Item 30.)*”*

*M: C++ structure: concept perfect forwarding. It is necessary to better understand.*

*M: C++ concepts: exception safe and basic guarantee. It is necessary to better understand. What is basic?*

*M: basic guarantee definition: “*Such functions assure callers that even if an exception is thrown, program invariants remain intact (i.e., no data structures are corrupted) and no resources are leaked.*”. It is necessary to better understand.*

*M: strong guarantee definition: “M: basic guarantee definition: “*Such functions assure callers that even if an exception is thrown, program invariants remain intact (i.e., no data structures are corrupted) and no resources are leaked.*”. It is necessary to better understand.*

*”. It is necessary to better understand.*

*It is necessary to better understand, what is program state. It is necessary to better understand what is difference between strong guarantee and basic guarantee.*

*p. 23*

*M: C++ concept: function object. Its definition:* When I refer to a function object, I usually mean an object of a type supporting an operator() member function. In other words, an object that acts like a function. Occasionally I use the term in a slightly more general sense to mean anything that can be invoked using the syntax of a non-member function call (i.e., “function Name(arguments)”).”

*M: C++ concept: C-like function pointers. It is necessary to better understand. (intbu)*

*M: C++ concept: callable objects. Intbu.*

*M: C++ concept: lambda expressions. Intbu.*

*M: C++ concept: closures. Definition: “*Function objects created through lambda expressions are known as closures.*”. Intbu.*

*M: book structure: lambdas: “*It’s sel-

dom necessary to distinguish between lambda expressions and the closures they create, so I often refer to both as lambdas.*”*

*M: C++ concepts: function templates, template functions, class templates, template classes. It is necessary to better understand.*

*M: book structure: author doesn’t distinguish between function templates and template functions. author doesn’t distinguish between class templates, template classes.*

*M: C++ structure: “*Many things in C++ can be both declared and defined.*”*

*M: C++ concept: declaration: “*Declarations introduce names and types without giving details, such as where storage is located or how things are implemented*”*

*M: C++ concept: definition: “*Definitions provide the storage locations or implementation details*”. Connection with concept “declaration”:* A definition also qualifies as a declaration.

*p. 24*

I define a …

25.11.2016

*C++ concept: function signature: “part of function declaration that specifies parameter and return type. Function and parameter names are not part of the signature. In the example above, func’s signature is bool(const Widget&). The official definition of “signature” is slightly different from mine, but for this book, my definition is more useful”.*

*What is noexcept and constexpr?*

*C++ evolution: “*New C++ Standards generally preserve the validity of code written under older ones, but occasionally the Standardization Committee deprecates features. Such feature are on standardization death row and may be removed from future Standards. Compilers may or may not warn about the use of deprecated features, but you should do your best to avoid them. Not only can they lead to future porting headaches, they’re generally inferior to the features that replace them. For example, std::auto\_ptr isdeprecated in C++11, because std::unique\_ptr does the same job, only better. *”. M: qualitative using of C++: deprecate features.*

*C++ structure: undefined behavior: “*Sometimes a Standard says that the result of an operation is undefined behavior. That means that runtime behavior is unpredictable, and it should go without saying that you want to steer clear of such uncertainty.*” M: qualitative using of C++: steer undefined behavior.*

*C++ structure: undefined behavior: examples: “*Examples of actions with undefined behavior include using square brackets (“[]”) to index beyond the bounds of a std::vector\* dereferencing an uninitialized iterator\* or engaging in a data race (i.e., having two or more threads, at least one of which is a writer, simultaneously access the same memory location)*”*

*book concept: raw pointers: “I call built-in pointer, such as those returned from new, raw pointers.”*

*Book concept: smart pointer: “Smart pointers normally overload the pointer-dereferencing operators (operator -> and operator \*), though Item 20 explains that std::weak\_ptr is an exception”. M: uniqueness of weak\_ptr: it is exception among pointers.*

*Аспекты1 маёй сістэмы: выключэнне: выключэнне з чаго? Магчыма, кампутар убачыць у тэксце выключэнне, але не зможа зразумець, з чаго гэта выключэнне. Таму варта гэта будзе адзначыць. Трэба ўвесці аб’ект запытвальніка ў сістэму. Гэта аб’ект, які інкапсуліруе пытанне інтэрпрэтатара да чалавека, каб лепей нешта зразумець.*

*Connection between concepts: “the opposite of a raw pointer is a smart pointer”.*

*Book designations: dtor – destructor\* ctor – constructor.*

*Structure of process of representation creation: “I’ve done my best to fill this book with clear, accurate, useful information, but surely there are ways to make it better. If you find errors of any kind (technical, expository, grammatical, typographical, etc.), or if you have suggestions for how the book could be improved, please email me at* [*emc++@aristeia.com*](mailto:emc++@aristeia.com)*”. M: kinds of errors: intbu.*

*Аспекты маёй сістэмы: павінен быць аб’ект, які ўказвае разуменне. Хаця разуменне кампутар можа спазнаць толькі праз метамаркеры, бо я магу проста выпісаць тэкст з кнігі і па тым, як я пішу, кампутар не вызначыць, якое маё разуменне. А вось задаючы пытанні, можа. Магчыма, варта будзе ўвесці і гэты блок. Але гэта ўжо цяжка. Магчыма, і непатрэбна. Трэба пакуль вырашаць задачу мінімум. Каб добра ведаць змест кнігі.*

*p.27*

## *Deducing Types*

*C++ structure: rules for type deduction. M: kinds: for templates\* for auto\* fro decltype.*

*Keyword auto: itnbu.*

*Keyword decltype: itnbu.*

*C++ history: “*C++98 had a single set of rules for type deduction: the one for function templates. C++11 modifies that ruleset a bit and adds two more, one for auto and one for decltype. C++14 then extends the usage contexts in which auto and decltype may be employed.*”*

*Purpose and advantage of type deduction: “*The increasingly widespread application of type deduction frees you from the tyranny of spelling out types that are obvious or redundant.*” M: thinking theory: problem of excess work.*

*Purpose and advantage of type deduction: “*It makes C++ software more adaptable, because changing a type at one point in the source code automatically propagates through type deduction to other locations..*” M: structure of concept adaptability of software.*

*Underwater stones of type deduction: “*However, it can render code more difficult to reason about, because the types deduced by compilers may not be as apparent as you’d like.*”*

*Importance of type deduction: “*Without a solid understanding of how type deduction operates, effective program-ming in modern C++ is all but impossible.*”*

*Context of type deduction: “*in calls to function templates\* in most situations where auto appears\* in decltype expressions\* and, as of C++14, where the enigmatic decltype(auto) construct is employed\**”*

*What is “decltype(auto) construct”?*

*Book structure: “*This chapter provides the information about type deduction that every C++ developer requires. It explains how template type deduction works, how auto builds on that, and how decltype goes its own way. It even explains how you can force compilers to make the results of their type deductions visible, thus enabling you to ensure that compilers are deducing the types you want them to.*”*

*Task of C++ programming: “*how you can force compilers to make the results of their type deductions visible, thus enabling you to ensure that compilers are deducing the types you want them to.*”*

### ***Item 1: Understand template type deduction***

*M: type deduction advantage\* type deduction meaning\* quality criteria of complex system from the user point of view: “*When users of a complex system are ignorant of how it works, yet happy with what it does, that says a lot about the design of the system. By this measure, template type deduction in C++ is a tremendous success.*”*

*p. 28*

*C++ feature: auto.*

*Structure of auto\* meaning of auto\*: “*type deduction for templates is the basis for one of modern C++’s most compelling features: auto*”*

*Аспекты маёй сістэмы: можна вызначыць усе месцы, дзе з’явілася паняцце. А потым паглядзець, ці адлюстраваны яны ў яго стурктуры. І сачыць за гэтым.*

*Nuance of type deduction: “*The bad news is that when the template type deduction rules are applied in the context of auto, they sometimes seem less intuitive than when they’re applied to templates.*”*

*Book structure: “*The bad news is that when the template type deduction rules are applied in the context of auto, they sometimes seem less intuitive than when they’re applied to templates. For that reason, it’s important to truly understand the aspects of template type deduction that auto builds on. This Item covers what you need to know.*”*

*C++ feature: type deduction.*

*C++ concepts: compiler\* compilation.*

*C++ relation: compiler perform compilation.*

*Сэнсы слоў якасна розныя. Таму іх варта ўносіць у сістэму як пералічэнні. Сувязі між лікамі і ўказацелямі не могуць адлюстраваць сувязі між сэнсамі. А вось камбініраванне праграмных сутнасцяў, кожная з якіх адлюстравана чалавекам на пэўны сэнс, магчыма, ужо і можа адлюстроўваць больш глыбокія сувязі.*

*Description of kind of type deduction “for template”: p. 28*

*C++ concept: template function.*

*Structure of template function: type parameter is different from type of function argument, because the last often contains adornments, e.g., const or reference qualifiers.*

*C++ concept: qualifiers.*

*Example of template type deduction: p. 28.*

*Structure of template type deduction: “*The type deduced for T is dependent not just on the type of expr, but also on the form of ParamType.*”*

*C++ concept: universal reference. What does it mean?*

*p. 29*

*Difference between concepts universal reference\* lvalue reference\* rvalue reference.*

*Structure of template type deduction: scenerios: ParamType is a pointer or reference type, but not a universal reference (n case1 n)\* ParamType is a universal reference (n case2 n)\* ParamType is neither a pointer nor a reference (n case3 n).*

### ***Case 1: ParamType is a Reference or Pointer, but not a Universal Reference***

*Case 1 structure: If expr’s type is a reference, ignore the reference part. Then pattern-match expr’s type against ParamType to determine T. There are examples on page 29. We simply ignore & in type of argument.*

*P. 30*

*Case 1 structure: safety of passing a const object to a template. It won’t be modifiable.*

*Case 1 structure: in the third example, note that even though rx’s typs is a reference, T is deduced to be a non-reference. That’s because rx’s reference-ness is ignored during type deduction.*

*Аспекты маёй сістэмы: праблема ідэнтыфікатараў: вось вышэй сказ, які тычыцца другога пункту кэйса 1. Але як кампутару гэта высветліць. Па лічбе 2, магчыма. Але калі там 2 не будзе? Значыць, трэба гэты сказ нейкім класіфікатарам абазначаць? Паразважаць.*

*Case 1 structure: type deduction works in the same way as for lvalues as for rvalues.*

*Case 1 structure: another example of type deduction.*

*Question concerning examples on pages 29 and 30: when we pass lvalue and when rvalue?*

*Case 1 stucture: “*If

param

were a pointer (or a pointer to const) instead of a reference, things would

work essentially the same way*”. Examples are on p. 30-31.*

*p. 31*

*Quality of type deduction system: “Everything is obvious! Which is exactly what you want in a type deduction system”.*

*Type deduction purpose: it is not necessary to define concrete template function as I understand (my comment).*

*Аспекты маёй сістэмы: кампутар не можа зразумець, што ёсць маё меркаванне, а што выпісана з кнігі, а што перарфазіравана. Толькі, калі ён будзе здольны аналізаваць мой стыль пісьма. Паразважаць.*

### ***Case 2: ParamType is a Universal Reference***

*p. 31*

*28.11.2016*

*Паняцце “Другі выпадак прывядзення тыпу” – параметр ёсць універсальная спасылка.*

*Разнавіднасць паняцця “параметр” – “універсальны параметр”. Абазначэнне паняцця – T&&*

*«But they behave differently when lvalue arguments are passed int» - характарыстыка паняцця.*

*Дадзенае месца злучана з Item 24.*

*Структура дадзенага выпадку. Ён уключае два наступныя выпадкі.*

* *If expr is an lvalue, both T and ParamType are deduced to be lvalue references. That’s doubly unusual. First, it’s only situation in template type deduction where T is deduced to be a reference. Second, although ParamType is declared using the syntax for an rvalue reference, its deduced type is an lvalue reference.*
* *If expr is an rvalue, the “normal” (i.e., Case 1) rules apply.*

*p. 32*

*Каментар: відаць, гэта такія правілы С++. “Item 24 explains exactly why these examples play out the way they do.”*

*Characteristic of “type deduction”: “*The key point here is that the type deduction rules for universal reference parameters are different from those for parameters that are lvalue references or rvalue references. In particular, when universal references are in use, type deduction distinguishes between lvalue arguments and rvalue arguments*”. Properties of type deduction, rvalue and lvalue.*

### **Case 3: ParamType is Neither a Pointer nor a Reference**

*Case of “”type deduction”.*

*Its structure: “That means that param will be a cope of whatever is passed in – a completely new object”.*

*Its rules:*

1. *“as before, if expr’s type is a reference, ignore the reference part”*
2. *“If, after ignoring expr’s reference-ness, expr is const, ignore that, too. If it’s volatile, also ignore that. (volatile objects are uncommon. They’re generally used only for implementing device drivers. For details, see Item 40)”. Purpose and properties of “const”, and “volatile”.*

*ules explanation: “*That makes sense. Param is an object that’s completely independent of cx and rx—a copy of cx or rx. The fact that cx and rx can’t be modified says nothing about whether param can be. That’s why expr’s constness (and volatileness, if any) is ignored when deducing a type for param: just because expr can’t be modified doesn’t mean that a copy of it can’t be.*”*

*Type deduction and const properties: “*It’s important to recognize that

const

(and volatile) is ignored only for by-value

parameters. As we’ve seen, for parameters that are references-to- or pointers-toconst,

the constness of

expr

is preserved during type deduction.*” Concepts connection.*

*p. 33*

*interesting example of deduction of const pointer. Properties of “const” and “pointer”.*

*Аспекты маёй сістэмы: мая сістэма можа ведаць, што дадзены блок тэкста тычыцца такіх-та паняццяў, такіх та атамарных тэм і гэтак далей. Калі я запрашу “уласцівасці паняцця”, то сістэма выдасць мне ўсё, што я пазначыў як уласцівасці паняцця. А калі я хачу нейкія больш канкрэтныя ўласцівасці, то мне іх і трэба запісваць. Я магу запрасіць таксама ўсе паняцці, злучаныя з дадзеным напрамую. Структуру сістэмы ведаў. І г.д. Паразважаць.*

### ***Array Arguments***

*Structure of type deduction.*

*Concept “array arguments”.*

*Differences between concepts “pointers” and “arrays”: “*It’s that array types are different from pointer types, even though they sometimes seem to be interchangeable. A primary contributor to this illusion is that, in many contexts, an array decays into a pointer to its first element.*”*

*Example of array declaration. Decay of array to pointer.*

*p. 34*

*properties of function, array and pointer: “array as function parameter decays to pointer”.*

*Roots of C in C++. – рубрыка сісвед.*

*Аспекты маёй сістэмы: можна вылучаць рубрыку сісвед. Усё тое, што тычыцца, дадзенай рубрыкі, можна спрабаваць вывесці.*

*Two interesting cases of array type deducing.*

*There is code example in this page. It is necessary to better understand it.*

*p. 35*

*what is “constexpr”?*

*what is std::array??*

*“*Of course, as a modern C++ developer, you’d naturally prefer a std::array to a built-in array*”. Advantage of “std::array” on “built-in array”.*

*What is noexcept.*

*Noexcept advantage: “*As for

arraySize

being declared noexcept, that’s to help compilers generate better code.*” Book graph structure. See Item 14. Connection of noexcept and compiler.*

### **Function Arguments**

*C++ structure: “*Arrays aren’t the only things in C++ that can decay into pointers. Function types can decay into function pointers, and everything we’ve discussed regarding type deduction for arrays applies to type deduction for functions and their decay into function pointers.*”*

*Type deducing case: function object. Analogy with array. Connection of function object and pointer. Aspects of function pointer.*

*What is function pointer?*

*What is function object?*

*Example of “function type”.*

*Kinds of decay: “array-to-pointer decay” and “function-to-pointer decay”.*

*p. 36*

*type deducing description: “*So there you have it: the auto-related rules for template type deduction. I remarked at the outset that they’re pretty straightforward, and for the most part, they are. The special treatment accorded lvalues when deducing types for universal referencesmuddies the water a bit, however, and the decay-to-pointer rules for arrays and functions stirs up even greater turbidity. Sometimes you simply want to grab your compilers and demand, “Tell me what type you’re deducing!” When that happens, turn to Item 4, because it’s devoted to coaxing compilers into doing just that.*”*

*Book graph structure. Item 4.*

*Type deducing description: “*Things to Remember

• During template type deduction, arguments that are references are treated as non-references, i.e., their reference-ness is ignored.

• When deducing types for universal reference parameters, lvalue arguments get special treatment.

• When deducing types for by-value parameters, const and/or volatile arguments are treated as non-const and non-volatile.

• During template type deduction, arguments that are array or function names decay to pointers, unless they’re used to initialize references.*”*

## **Item 2: Understand auto type deduction**

*p. 36*

*29.11.2016*

*Property of auto type deduction and template type deduction: “*If you’ve read Item 1 on template type deduction, you already know almost everything you need to know about auto type deduction, because, with only one curious exception, auto type deduction is template type deduction. But how can that be? Template type deduction involves templates and functions and parameters, but auto deals with none of those things.*”*

*Connection between auto type deduction and template type deduction: “*That’s true, but it doesn’t matter. There’s a direct mapping between template type deduction and auto type deduction. There is literally an algorithmic transformation from one to the other.*”*

*Compiler and type deduction property: how compiler performs type deduction.*

*p. 37*

*comparison text and graphical representation: “*This is easier to show than to describe, so consider this example*”.*

*Comparison of auto type deducing and type deducing for templates: “*As I said, deducing types for auto is, with only one exception (which we’ll discuss soon), the same as deducing types for templates.*”*

*Three cases of auto type deduction: “*In a variable declaration using auto, the type specifier takes the place of ParamType, so there are three cases for that, too:

• Case 1: The type specifier is a pointer or reference, but not a universal reference.

• Case 2: The type specifier is a universal reference.”

Case 3: The type specifier is neither a pointer nor a reference.*”*

*p. 38*

*Item 2 is described by analogy with Item 1.*

*Auto type deduction property: “*Item 1 concludes with a discussion of how array and function names decay into pointers for non-reference type specifiers. That happens in auto type deduction, too: *”*

*Auto type deduction purpose: as far as I understood, auto allow programmer not to indicate type directly. Compiler will understand type itself. It is convenient feature.*

*Connection of auto and typedef:* [*https://habrahabr.ru/post/182920/ - typedef зараз састарэлы, бо ён выкарыстоўваўся для простасці ў мінулым кодзе*](https://habrahabr.ru/post/182920/%20-%20typedef%20зараз%20састарэлы,%20бо%20ён%20выкарыстоўваўся%20для%20простасці%20ў%20мінулым%20кодзе)*.*

*Using auto as function return value:* [*https://habrahabr.ru/post/182920/ - it indicates that return value must be found by compiler in the end of function*](https://habrahabr.ru/post/182920/%20-%20it%20indicates%20that%20return%20value%20must%20be%20found%20by%20compiler%20in%20the%20end%20of%20function)*. Compiler structure.*

*p. 39*

*C++ structure, concept int, concept declaration: 4 ways of int type declaration. Advantage of auto declaration under int declaration. Kinds of declarations. Connection with Item 5.*

*Auto declaration (type deducing) nuances: “*Straightforward textual substitution yields this code:

auto x1 = 27;

auto x2(27);

auto x3 = { 27 };

auto x4{ 27 };

These declarations all compile, but they don’t have the same meaning as the ones they replace. The first two statements do, indeed, declare a variable of type int with value 27. The second two, however, declare a variable of type std::initializer\_list<int> containing a single element with value 27!*”*

*Аспекты маёй сістэмы: у файлах будуць захоўвацца тэкставыя элементы і структура сістэмы ведаў. Дакладна там будуць паняцці і сувязі між імі.*

*Auto type deducing case: “*This is due to a special type deduction rule for auto. When the initializer for an auto-declared variable is enclosed in braces, the deduced type is a std::initializer\_list. If such a type can’t be deduced (e.g., because the values in the braced initializer are of different types), the code will be rejected:

auto x5 = { 1, 2, 3.0 }; // error! can't deduce T for

// std::initializer\_list<T>*”*

*p. 40*

*There is previous case description.*

*Difference of auto type deduction and template type deduction. std::initializer\_list structure. What is it? “*The treatment of braced initializers is the only way in which auto type deduction and template type deduction differ. When an auto–declared variable is initialized with a braced initializer, the deduced type is an nstantiation of std::initializer\_list.

But if the corresponding template is passed the same initializer, type deduction fails, and the code is rejected:

auto x = { 11, 23, 9 }; // x's type is

// std::initializer\_list<int>

template<typename T> // template with parameter

void f(T param); // declaration equivalent to

// x's declaration

f({ 11, 23, 9 }); // error! can't deduce type for T

However, if you specify in the template that param is a std::initializer\_list<T> for some unknown T, template type deduction will deduce what T is:

template<typename T>

void f(std::initializer\_list<T> initList);

f({ 11, 23, 9 }); // T deduced as int, and initList's

// type is std::initializer\_list<int>*”*

*Difference between auto and template type deduction: “*So the only real difference between auto and template type deduction is that auto assumes that a braced initializer represents a std::initializer\_list, but template type deduction doesn’t.*”*

*The reason of this difference is not understood by author.*

*C++ practice and feature: philosophy of uniform initialization. Connection with auto: “*It’s especially important to bear this in mind if you embrace the philosophy of uniform initialization—of enclosing initializing values in braces as a matter of course.*” What is the purpose of this philosophy?*

*p. 41*

*error in C++ programming: “*in C++11 programming is accidentally declaring a std::initializer\_list variable

when you mean to declare something else. This pitfall is one of the reasons some developers put braces around their initializers only when they have to.*” Connection with Item 7: “*(When you have to is discussed in Item 7.)*”*

*Аспекты маёй сістэмы ведаў: графавыя сувязі кнігі, узгаданыя выяўна – гэта і для структуры кнігі важна, а таксама ўказвае структура сісвед.*

*С++ historyю Auto designation. Connection between function return value type and auto. Connection between lambda and auto: “*For C++11, this is the full story, but for C++14, the tale continues. C++14 permits auto to indicate that a function’s return type should be deduced (see Item 3), and C++14 lambdas may use auto in parameter declarations. However, these uses of auto employ template type deduction, not auto type deduction.*” Connection with Item 3. Context of using of auto type deduction and template type deduction. Underwater stone of C++.*

*Аспекты маёй сістэмы: графавая структура кнігі можа злучаць або дадзеную атамарную тэму з усёй айтэм3, або з яе атамарная тэмай. Лепей захоўваць і тое, і тое.*

*Аспекты маёй сістэмы: дапусцім, некаторыя блокі тэкста тычацца С++ гісторыі. Важна пазначаць, што яны тычацца і С++ прывядзення тыпаў. Гэта прызначэнне хэштэгаў.*

*C++ error: “*So a function with an auto return type that returns a braced initializer won’t compile:

auto createInitList()

{

return { 1, 2, 3 }; // error: can't deduce type

} // for { 1, 2, 3 }*”*

*Аспекты маёй сістэмы: Калі нешта змешчана ў нейкім айтэме, то для разумення гэтага, можа знаходзіць усё, што тычыцца гэтага айтэма. Калі робіш запыт.*

*Аспекты маёй сістэмы: важна, каб інфармацыя, выдаваемая сістэмай, не была адразу поўнай, а рэпрэзентавалася скарочана, па назвах. А тое, што чалавек не прыгадаў, можна выдаваць ужо поўна.*

*Example of lambda: “*The same is true when auto is used in a parameter type specification in a C++14 lambda:

std::vector<int> v;

…

auto resetV =

[&v](const auto& newValue) { v = newValue; }; // C++14

…

resetV({ 1, 2, 3 }); // error! can't deduce type

// for { 1, 2, 3 }*”*

*Things to remember: “*auto type deduction is usually the same as template type deduction, but auto type deduction assumes that a braced initializer represents a std::initializer\_list, and template type deduction doesn’t. • auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.*”*

*C++ concept, connecting with this item: brace initializer. What is initializer?*

## **Item 3: Understand decltype**

*p. 41*

*30.11.2016*

*Concept definition and purpose. It allows to define expression’s type. Connected with concept expression.*

*Typical feature of concept: “*Typically, what it tells you is exactly what you’d predict*”.*

*Underwater stones of concept: “*predict. Occasionally however, it provides results that leave you scratching your head and turning to reference works or online Q&A sites for revelation.*” C++ problem. Error is type of problem.*

*Aspects of my system: typical feature and allow to define are common structure connections.*

*Connection with item 1 and 2. Decltype property: “*In contrast to what happens during type deduction for templates and auto (see Items 1 and 2), decltype typically parrots back the exact type of the name or expression you give it:*”*

*Decltype examples:*

*“*

const int i = 0; // decltype(i) is const int

bool f(const Widget& w); // decltype(w) is const Widget&

// decltype(f) is bool(const Widget&)

struct Point {

int x, y; // decltype(Point::x) is int

}; // decltype(Point::y) is int

Widget w; // decltype(w) is Widget

if (f(w)) … // decltype(f(w)) is bool

template<typename T> // simplified version of std::vector

class vector {

public:

…

T& operator[](std::size\_t index);

…

};

vector<int> v; // decltype(v) is vector<int>

…

if (v[0] == 0) … // decltype(v[0]) is int&

*”*

*Decltype context of using: “*In C++11, perhaps the primary use for decltype is declaring function templates where the function’s return type depends on its parameter types. For example, suppose we’d like to write a function that takes a container that supports indexing via square brackets (i.e., the use of “[]”) plus an index, then authenticates the user before returning the result of the indexing operation. The return type of the function should be the same as the type returned by the indexing operation.*” Case of template function using.*

*Book example: concerning operator []. Book connection with Item 6: “*operator[] on a container of objects of type T typically returns a T&. This is the case for std::deque, for example, and it’s almost always the case for std::vector. For std::vector<bool>, however, operator[] does not return a bool&. Instead, it returns a brand new object. The whys and hows of this situation are explored in Item 6, but what’s important here is that the type returned by a container’s operator[] depends on the container.*”*

*p. 43*

*C++ structure: concept std – standard library of C++; stl – standard library of templates is part of std. (*[*https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F\_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0\_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0\_C%2B%2B*](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_C%2B%2B) *)*

*Properties of stl containers.*

*Explaining example of this use case of decltype and stl container:*

*“*template<typename Container, typename Index> // works, but

auto authAndAccess(Container& c, Index i) // requires

-> decltype(c[i]) // refinement

{

authenticateUser();

return c[i];

}*”*

*It is necessary to better understand line 3 of it.*

*С++ structure: trailing return type: “*The use of auto before the function name has nothing to do with type deduction. Rather, it indicates that C++11’s trailing return type syntax is being used, i.e., that the function’s return type will be declared following the parameter list (after the “->”). A trailing return type has the advantage that the function’s parameters can be used in the specification of the return type.*”. Advantage of this concept. Example of it.*

*Aspects of my system: гэта прыклад, які апісвае тры тэмы сісвед. Прычым паміж блокамі тэкста, прысвечанымі самому прыкладу, узгадваюцца блокі тэкста, прысвечаныя дадзеным тэмам. Яны ж, зразумела, апісваюць і сам прыклад. Але выяўляюць трошку іншую тэму, чым іншыя апісанні прыклада. Яны паказваюць розныя рысы і бакі прыклада.*

*С++ history: “*C++11 permits return types for single-statement lambdas to be deduced, and C++14 extends this to both all lambdas and all functions, including those with multiple statements.*” Concepts: single statement lambdas, lambdas (what kinds exist?) and functions.*

*Type deduction case: “*With that form of declaration, auto does mean that type deduction will take place.*” Aspect of the given example. Connection of type take place.*

*аспекты маёй сістэмы: сувязь тыпу “мае месца”.*

*Compiler work. Connection of compiler and type deduction.*

*Compiler is universal entity connecting with all many concepts of C++ by connection “realization”: “*In particular, it means that compilers will deduce the function’s return type from the function’s implementation*”*

*Аспекты маёй сістэмы: разнавіднасць элемента структуры і сувязь з іншымі элемнетамі.*

*Example continue:*

template<typename Container, typename Index> // C++14;

auto authAndAccess(Container& c, Index i) // not quite

{ // correct

authenticateUser();

return c[i]; // return type deduced from c[i]

}

*This example show some alternatives.*

*Аспекты маёй сістэмы: атмарная тэма можа паказваць альтэрнатывы. Альбо яна можа паказваць паслядоўную мадыфікацыю сутнасці. Як мне ажыццяўляць эфектыўны пошук патрэбных метам сярод зашытых у сістэму. Як сістэме распазнаваць, якія тут ёсць сувязі? Па тэксце? Пакуль бачу толькі, што мне самому трэба пачынаць пісаць сувязь, а кампутар павінен падхопліваць яе – ці ёсць у яго такая.*

*Example is connected with Item 2 and Item 1.*

*p. 44*

*Example of concept client code: “*Consider what that means for this client code:

std::deque<int> d;

…

authAndAccess(d, 5) = 10; // authenticate user, return d[5],

// then assign 10 to it;

// this won't compile!

Here, d[5] returns an int&, but auto return type deduction for authAndAccess will strip off the reference, thus yielding a return type of int. That int, being the return value of a function, is an rvalue, and the code above thus attempts to assign 10 to an rvalue int. That’s forbidden in C++, so the code won’t compile.*” example of error. Example of rvalue int. Intbu this concept. C++ forbidden.*

*C++ forbidden – is important part of structure of C++ sysved.*

*Decltype purpose: “*To get authAndAccess to work as we’d like, we need to use decltype type deduction for its return type, i.e., to specify that authAndAccess should return exactly the same type that the expression c[i] returns. The guardians of C++, anticipating the need to use decltype type deduction rules in some cases where types are inferred, make this possible in C++14 through the decltype(auto) specifier.*” What is specifier. Specifier decltype(auto).*

*Decltype and auto and decltype(auto) specifier property: “*What may initially seem contradictory (decltype and auto?) actually makes perfect sense: auto specifies that the type is to be deduced, and decltype says that decltype rules should be used during the deduction.*”*

*Decltype is some rules.*

*Auto is rules or simply indicator for some compiler action. C++ structure.*

*Example continue: “*template<typename Container, typename Index> // C++14; works,

decltype(auto) // but still

authAndAccess(Container& c, Index i) // requires

{ // refinement

authenticateUser();

return c[i];

}*”*

*Аспекты маёй сістэмы: прыклад патэрна рэпрэзентацыі, калі адзін і той жа прыклад мадыфіцыруецца, або разглядаюць розныя альтэрнатывы, а ў атамарных тэмах паміж імі разглядаюць розныя аспекты, якія маюць месца ў гэтым прыкладзе. Думаю, у сістэме варта пазначаць і патэрн рэпрэзентацыі.*

*Decltype and example aspect: “*Now authAndAccess will truly return whatever c[i] returns. In particular, for the common case where c[i] returns a T&, authAndAccess will also return a T&, and in the uncommon case where c[i] returns an object, authAndAccess will return an object, too.*”*

*Using context of decltype and other example: “*The use of decltype(auto) is not limited to function return types. It can also be convenient for declaring variables when you want to apply the decltype type deduction rules to the initializing expression:

Widget w;

const Widget& cw = w;

auto myWidget1 = cw; // auto type deduction:

// myWidget1's type is Widget

decltype(auto) myWidget2 = cw; // decltype type deduction:

// myWidget2's type is

// const Widget&*” каментар: без decltype myWidget2 не быў бы спасылкай, але аб’ектам па значэнні.*

*p.45*

*example continue: “*Look again at the declaration for the C++14 version of authAndAccess:

template<typename Container, typename Index>

decltype(auto) authAndAccess(Container& c, Index i);

The container is passed by lvalue-reference-to-non-const, because returning a reference to an element of the container permits clients to modify that container. But this means it’s not possible to pass rvalue containers to this function. Rvalues can’t bind to lvalue references (unless they’re lvalue-references-to-const, which is not the case here).*” rvalue and lvalue properties and connection. Пытанне: чаму калі кліенты не могуць мадыфіцыраваць кантэйнер, то ён не можа быць рвэлью?*

*аспекты маёй сістэмы: як кампутар зразумее, які прыклад я працягваю.*

*С++ not good practice: “*Admittedly, passing an rvalue container to authAndAccess is an edge case.*” What is rvalue container?*

*Аспекты маёй сістэмы: мы бачым главу, дзе ёсць адзін прыклад, які змяшчае шмат аспектаў, з якімі злучаны розныя шматлікія тэмы.*

*Example of dangling and rvalue: “*An rvalue container, being a temporary object, would typically be destroyed at the end of the statement containing the call to authAndAccess, and that means that a reference to

an element in that container (which is typically what authAndAccess would return) would dangle at the end of the statement that created it. Still, it could make sense to pass a temporary object to authAndAccess.*” Temporary object is rvalue example?*

*Connection of rvalue, lvalue and overloading: “*Overloading would work (one overload would declare an lvalue reference parameter, the other an rvalue reference parameter), but then we’d have two functions to maintain.*”. Universal reference is alternative: “*A way to avoid that is to have authAndAc cess employ a reference parameter that can bind to lvalues and rvalues, and Item 24 explains that that’s exactly what universal references do.*”. Universal reference property and designation. Book connection – item 24.*

*Аспекты маёй сістэмы: сувязь кнігі – гэта спасылка ад аднаго элемента кнігі на другі элемент кнігі. Але гэтыя элементы кнігі могуць тычыцца розных элемнетаў сісвед. Важна адрозніваць структуру кнігі ад структуры сісвед.*

*Example continue: “*authAndAccess can therefore be declared like this:

template<typename Container, typename Index> // c is now a

decltype(auto) authAndAccess(Container&& c, // universal

Index i); // reference*”*

*p. 46*

*C++ potentially dangerous practice: “*Employing pass-byvalue for objects of an unknown type generally risks the performance hit of unnecessary copying, the behavioral problems of object slicing (see Item 41), and the sting of our coworkers’ derision, but in the case of container indices, following the example of the Standard Library for index values (e.g., in operator[] for std::string, std::vector, and std::deque) seems reasonable, so we’ll stick with pass-by-value for them. *” Possible problems:* performance hit of unnecessary copying, behavioral problems of object slicing. *Intby the last. Book connection with Item 41. Exception is std::operator[].*

*“*However, we need to update the template’s implementation to bring it into accord with Item 25’s admonition to apply std::forward to universal references:*” intbu. What is std::forward? Connection of book – Item 25. Intbu.*

template<typename Container, typename Index> // final

decltype(auto) // C++14

authAndAccess(Container&& c, Index i) // version

{

authenticateUser();

return std::forward<Container>(c)[i];

}

*C++ history and compiler types: “*This should do everything we want, but it requires a C++14 compiler.*”*

*Example continue: “*

template<typename Container, typename Index> // final

auto // C++11

authAndAccess(Container&& c, Index i) // version

-> decltype(std::forward<Container>(c)[i])

{

authenticateUser();

return std::forward<Container>(c)[i];

}*” Difference between C++ standards. What C++14 cand C++11 can’t.*

*Аспекты маёй сістэмы: можна было заўважыць, што адзін з прыкладаў кода я выяўна не пазначыў, як прыклад. Але кампутар можа дадумацца, які прыклад апісваецца – па падабенстве кода. А вось калі я ў рамках таго ж прыклада напішу альтэрнатыву, якая адрозніваецца знешне, то дадумацца кампутару будзе значна цяжэй.*

*Decltype use cases and their frequency to meet: “*The other issue that’s likely to be nagging at you is my remark at the beginning of this Item that decltype almost always produces the type you expect, that it rarely surprises. Truth be told, you’re unlikely to encounter these exceptions to the rule unless you’re a heavy-duty library implementer.*” It is predictable in the most cases.*

To fully …

p. 46

*01.12.2016*

*Decltype purpose: “*Applying decltype to a name yields the declared type for that name.*” Aspects of declaration.*

*Connection between names and lvalue: “*Names are lvalue expressions, but that doesn’t affect decltype’s behavior.*” Intbu.*

*Decltype structure. Lvalue aspects: “*For lvalue expressions more complicated than names, however, decltype ensures that the type reported is always an lvalue reference. That is, if an lvalue expression other than a name has type T, decltype reports that type as T&. This seldom has any impact, because the type of most lvalue expressions inherently includes an lvalue reference qualifier.*” Lvalue structure.*

*“*This seldom has any impact, because the type of most lvalue expressions inherently includes an lvalue reference qualifier.*” Lvalue structure. When it is not so?*

*p. 47*

*C++ structure. Lvalue, lvalue reference, function: “*Functions returning lvalues, for example, always return lvalue references.*”*

*Decltype nuance, underwater stone: “*There is an implication of this behavior that is worth being aware of, however. In int x = 0; x is the name of a variable, so decltype(x) is int. But wrapping the name x in parentheses—“(x)”—yields an expression more complicated than a name. Being a name, x is an lvalue, and C++ defines the expression (x) to be an lvalue, too. decltype((x)) is therefore int&. Putting parentheses around a name can change the type that decltype reports for it! *”*

*Decltype underwater stone: “*In C++11, this is little more than a curiosity, but in conjunction with C++14’s support for decltype(auto), it means that a seemingly trivial change in the way you write a return statement can affect the deduced type for a function:

decltype(auto) f1()

{

int x = 0;

…

return x; // decltype(x) is int, so f1 returns int

}

decltype(auto) f2()

{

int x = 0;

…

return (x); // decltype((x)) is int&, so f2 returns int&

}

Note that not only does f2 have a different return type from f1, it’s also returning a reference to a local variable! That’s the kind of code that puts you on the express train to undefined behavior—a train you certainly don’t want to be on.*” C++ error. Kind of undefined behavior “return reference to local variable”.*

*The upper example connects specificator “decltype(auto)”.*

*This specificator using technique: “*The primary lesson is to pay very close attention when using decltype(auto). Seemingly insignificant details in the expression whose type is being deduced can affect the type that decltype(auto) reports. To ensure that the type being deduced is the type you expect, use the techniques described in Item 4.*” Book connection. Context of using of technique in Item 4.*

*Аспекты маёй сістэмы: пры разуменні мы можам сувязь разумець раней, чым само паняцце, якое ёй адпавядае. Як маёй сістэме гэты выпадак апрацоўваць?*

*Reliability of decltype mechanism: “*At the same time, don’t lose sight of the bigger picture. Sure, decltype (both alone and in conjunction with auto) may occasionally yield type-deduction surprises, but that’s not the normal situation. Normally, decltype produces the type you expect. This is especially true when decltype is applied to names, because in that case, decltype does just what it sounds like: it reports that name’s declared type.*” characterization of decltype name case.*

*p. 48*

*Cases of decltype uses: for names and for expressions. There are concepts of C++.*

Things to Remember

• decltype almost always yields the type of a variable or expression without any modifications.

• For lvalue expressions of type T other than names, decltype always reports a type of T&.

• C++14 supports decltype(auto), which, like auto, deduces a type from its initializer, but it performs the type deduction using the decltype rules.

*Auto main idea: “deduces type from initializer”. Initializer – concept of C++.*

## ***Item 4:* Know how to view deduced types**

*This is possibility of C++. Or feature of C++.*

*Common for C++ concept “software development process” and its three stages: “*The choice of tools for viewing the results of type deduction is dependent on the phase of the software development process where you want the information. We’ll explore three possibilities: getting type deduction information as you edit your code, getting it during compilation, and getting it at runtime.*” Cases of getting information about types: classification over software development process.*

### **IDE Editors**

*IDE is also more common feature. C++ is part of software development process. IDE too. IDE is helper for C++ and other development process. Concretely it includes many different features. Documentation of them is enough great.*

*Common concept of programming: program entites. Their kinds: variables, parameters, functions, etc.*

*Example of type deduction view in IDE: “*For example, given this code,

const int theAnswer = 42;

auto x = theAnswer;

auto y = &theAnswer;

an IDE editor would likely show that x’s deduced type was int and y’s was const int\*.*” This is good clever IDE.*

*IDE type view structure: “*For this to work, your code must be in a more or less compilable state, because what makes it possible for the IDE to offer this kind of information is a C++ compiler (or at least the front end of one) running inside the IDE. If that compiler can’t make enough sense of your code to parse it and perform type deduction, it can’t show you what types it deduced. For simple types like int, information from IDEs is generally fine. As we’ll see soon, however, when more complicated types are involved, the information displayed by IDEs may not be particularly helpful.*” IDE type deduction view disadvantage.*

*p. 49*

### **Compiler Diagnostics**

*Kind of type deduction view – compiler deiagnostics: “*An effective way to get a compiler to show a type it has deduced is to use that type in a way that leads to compilation problems. The error message reporting the problem is virtually sure to mention the type that’s causing it.*”. Case of using it. Extremal one ))*

*C++ concept template. This is C++ feature and program entity. It has definition. Definition is important part of C++ language.*

*Using context of decltype – as template parameter: “*TD<decltype(x)> xType; // elicit errors containing

TD<decltype(y)> yType; // x's and y's types*”*

*Method example. Very extremal method: “*Suppose, for example, we’d like to see the types that were deduced for x and y in the previous example. We first declare a class template that we don’t define. Something like this does nicely:

template<typename T> // declaration only for TD;

class TD; // TD == "Type Displayer"

Attempts to instantiate this template will elicit an error message, because there’s no template definition to instantiate. To see the types for x and y, just try to instantiate TD with their types:

TD<decltype(x)> xType; // elicit errors containing

TD<decltype(y)> yType; // x's and y's types

I use variable names of the form variableNameType, because they tend to yield error messages that help me find the information I’m looking for. For the code above, one of my compilers issues diagnostics reading, in part, as follows (I’ve highlighted the type information we’re after):

error: aggregate 'TD<int> xType' has incomplete type and

cannot be defined

error: aggregate 'TD<const int \*> yType' has incomplete type

and cannot be defined

A different compiler provides the same information, but in a different form:

error: 'xType' uses undefined class 'TD<int>'

error: 'yType' uses undefined class 'TD<const int \*>'

Formatting differences aside, all the compilers I’ve tested produce error messages with useful type information when this technique is employed.*”*

*Аспекты маёй сістэмы: выявім, што я выкарыстоўваю словы замяняльнікі. Як кампутару дадумацца, што гэтыя словы – замяняльнікі. У яго павінна быць вялікая вербальная база дадзен, дзе ажыццяўляецца пошук. Наўрацці ў нашым розуме ажыццяўляецца пошук. Паразважаць.*

### **Runtime Output**

*Case of type deduction view.*

*Printf or std::cout and typeid context of using and disadvantage: “*The printf approach to displaying type information (not that I’m recommending you use printf) can’t be employed until runtime, but it offers full control over the formatting of the output. The challenge is to create a textual representation of the type you care about that is suitable for display. “No sweat,” you’re thinking, “it’s typeid and std::type\_info::name to the rescue.” In our continuing quest to see the types deduced for x and y, you may figure we can write this:

std::cout << typeid(x).name() << '\n'; // display types for

std::cout << typeid(y).name() << '\n'; // x and y*”*

*p. 50*

*“std::type\_info object” structure: “*This approach relies on the fact that invoking typeid on an object such as x or y yields a std::type\_info object, and std::type\_info has a member function, name, that produces a C-style string (i.e., a const char\*) representation of the name of the type.”

Example structure: “Calls to std::type\_info::name are not guaranteed to return anything sensible, but implementations try to be helpful. The level of helpfulness varies. The GNU and Clang compilers report that the type of x is “i”, and the type of y is “PKi”, for example. These results make sense once you learn that, in output from these compilers, “i” means “int” and “PK” means “pointer to konst const.” (Both compilers support a tool, c++filt, that decodes such “mangled” types.) Microsoft’s compiler produces less cryptic output: “int” for x and “int const \*” for y.*” Disadvantage of Linux. Differences between compilers.*

*Effectiveness of compiler type deducing view. Real example: “*Because these results are correct for the types of x and y, you might be tempted to view the type-reporting problem as solved, but let’s not be hasty. Consider a more complex example:

template<typename T> // template function to

void f(const T& param); // be called

std::vector<Widget> createVec(); // factory function

const auto vw = createVec(); // init vw w/factory return

if (!vw.empty()) {

f(&vw[0]); // call f

…

}

This code, which involves a user-defined type (Widget), an STL container (std::vector), and an auto variable (vw), is more representative of the situations where you might want some visibility into the types your compilers are deducing. For example, it’d be nice to know what types are inferred for the template type parameter T and the function parameter param in f.

Loosing typeid on the problem is straightforward. Just add some code to f to display the types you’d like to see:

template<typename T>

void f(const T& param)

{

using std::cout;

cout << "T = " << typeid(T).name() << '\n'; // show T

cout << "param = " << typeid(param).name() << '\n'; // show

… // param's

} // type

Executables produced by the GNU and Clang compilers produce this output:

T = PK6Widget

param = PK6Widget

We already know that for these compilers, PK means “pointer to const,” so the only mystery is the number 6. That’s simply the number of characters in the class name that follows (Widget). So these compilers tell us that both T and param are of type const Widget\*.Microsoft’s compiler concurs:

T = class Widget const \*

param = class Widget const \*

Three independent compilers producing the same information suggests that the information is accurate. But look more closely. In the template f, param’s declared type is const T&. That being the case, doesn’t it seem odd that T and param have the same type? If T were int, for example, param’s type should be const int&—not the same type at all.*”*

*p. 51*

*Reliability and structure of std::type\_info: “*Sadly, the results of std::type\_info::name are not reliable. In this case, for example, the type that all three compilers report for param are incorrect. Furthermore, they’re essentially required to be incorrect, because the specification for std::type\_info::name mandates that the type be treated as if it had been passed to a template function as a by-value parameter. As Item 1 explains, that means that if the type is a reference, its reference-ness is ignored, and if the type after reference removal is const (or volatile), its constness (or volatileness) is also ignored. That’s why param’s type—which is const Widget \* const &—is reported as const Widget\*. First the type’s reference-ness is removed, and then the constness of the resulting pointer is eliminated.*”*

*Question: what type* const Widget \* const & means???

IDE type deducing view reliability: “Equally sadly, the type information displayed by IDE editors is also not reliable—or at least not reliably useful. For this same example, one IDE editor I know reports T’s type as (I am not making this up):

const

std::\_Simple\_types<std::\_Wrap\_alloc<std::\_Vec\_base\_types<Widget,

std::allocator<Widget> >::\_Alloc>::value\_type>::value\_type \*

The same IDE editor shows param’s type as:

const std::\_Simple\_types<...>::value\_type \*const &

That’s less intimidating than the type for T, but the “...” in the middle is confusing until you realize that it’s the IDE editor’s way of saying “I’m omitting all that stuff that’s part of T’s type.” With any luck, your development environment does a better job on code like this.” *aspects of representation theory.*

*p. 52*

*Boost type index library – good instrument of type deducing view.*

*Boost libraries properties: “*Furthermore, the fact that Boost libraries (available at boost.com) are cross-platform, open source, and available under a license designed to be palatable to even the most paranoid corporate legal team means that code using Boost libraries is nearly as portable as code relying on the Standard Library.*” Translation must to be improved.*

*Using of boost type index library example: “*Here’s how our function f can produce accurate type information using Boost.TypeIndex:

#include <boost/type\_index.hpp>

template<typename T>

void f(const T& param)

{

using std::cout;

using boost::typeindex::type\_id\_with\_cvr;

// show T

cout << "T = "

<< type\_id\_with\_cvr<T>().pretty\_name()

<< '\n';

// show param's type

cout << "param = "

<< type\_id\_with\_cvr<decltype(param)>().pretty\_name()

<< '\n';

…

}”

The way this works is that the function template boost::typeindex::type\_id\_with\_cvr takes a type argument (the type about which we want information) and doesn’t remove const, volatile, or reference qualifiers (hence the “with\_cvr” in the template name). The result is a boost::typeindex::type\_index object, whose pretty\_name member function produces a std::string containing a human-friendly representation of the type.*”*

*Then there is demonstration that boost library works well.*

*Purpose of type deducing view methods. It is not replacement for understanding of Item1-3: “*Such nearniformity is nice, but it’s important to remember that IDE editors, compiler error messages, and libraries like Boost.TypeIndex are merely tools you can use to help you figure out what types your compilers are deducing. All can be helpful, but at the end of the day, there’s no substitute for understanding the type deduction information in Items 1–3.

…

The results of some tools may be neither helpful nor accurate, so an understanding of C++’s type deduction rules remains essential.*”*

Things to Remember

• Deduced types can often be seen using IDE editors, compiler error messages, and the Boost TypeIndex library.

• The results of some tools may be neither helpful nor accurate, so an understanding of C++’s type deduction rules remains essential.

# ***Chapter 2. Auto***

*p. 55*

*05.12.2016*

*Auto concept purposes (3): “*In concept, auto is as simple as simple can be, but it’s more subtle than it looks. Using it saves typing, sure, but it also prevents correctness and performance issues that can bedevil manual type declarations.*” C++ concept: “manual type declaration”. What correctness and performance problems are connected with them? Classes of problems.*

*Auto nuances and C++ good practice of programming: “*Furthermore, some of auto’s type deduction results, while dutifully conforming to the prescribed algorithm, are, from the perspective of a programmer, just wrong. When that’s the case, it’s important to know how to guide auto to the right answer, because falling back on manual type declarations is an alternative that’s often best avoided.*” Why is manual typing to be avoided?*

### **Item 5: Prefer auto to explicit type declarations**

*Аспекты маёй сістэмы: для таго, каб прыгадаць галіну, можна атрымліваць пытанні, а потым адказы на іх, але з рэпрэзентацыяй структуры галіны. Калі нейкі аспект з’яўляецца фундаментальным для пэўнай часткі структуры, то гэту сувязь важна падтрымліваць аўтаматычна.*

*Concept indeterminate value: “*Ah, the simple joy of int x; Wait. Damn. I forgot to initialize x, so its value is indeterminate. Maybe. It might actually be initialized to zero. Depends on the context. Sigh*”*

*p. 56*

*Didn’t understand translation: “*Ugh. “typename std::iterator\_traits<It>::value\_type” to express the type of the value pointed to by an iterator? Really? I must have blocked out the memory of how much fun that is. Damn. Wait—didn’t I already say that? Okay, simple joy (take three): the delight of declaring a local variable whose type is that of a closure. Oh, right. The type of a closure is known only to the compiler, hence can’t be written out. Sigh. Damn.*” this is book example.*

*Auto designation: solution of C++ problem “potentially uninitialized variable”: “*Well, it didn’t used to be. But as of C++11, all these issues go away, courtesy of auto. Auto variables have their type deduced from their initializer, so they must be initialized. That means you can wave goodbye to a host of uninitialized variable problems as you speed by on the modern C++ superhighway:

int x1; // potentially uninitialized

auto x2; // error! initializer required

auto x3 = 0; // fine, x's value is well-defined*” Auto property: auto variable must be initialized. Auto is mechanism of variable declaration.*

*Auto structure and book connection (Item 2): “*auto uses type deduction*”.*

*There is C++ history on this page.*

*Auto example: “*auto derefUPLess = // comparison func.

[](const std::unique\_ptr<Widget>& p1, // for Widgets

const std::unique\_ptr<Widget>& p2) // pointed to by

{ return \*p1 < \*p2; };*”*

*C++ history and structure: “*Very cool. In C++14, the temperature drops further, because parameters to lambda expressions may involve auto:

auto derefLess = // C++14 comparison

[](const auto& p1, // function for

const auto& p2) // values pointed

{ return \*p1 < \*p2; }; // to by anything

// pointer-like*” what is lambda expression in C++?*

*p. 57*

*what is “closure” in this book?*

*What is std::function object?*

*“*Coolness notwithstanding, perhaps you’re thinking we don’t really need auto to declare a variable that holds a closure, because we can use a std::function object. It’s true, we can, but possibly that’s not what you were thinking. And maybe now you’re thinking “What’s a std::function object?” So let’s clear that up.*” theory of representation. I think so ))) Example of cognitive orientation of reader.*

*What is function pointer? This is C++ feature.*

*History of C++ and std::function designation: “*std::function is a template in the C++11 Standard Library that generalizes the idea of a function pointer. Whereas function pointers can point only to functions, however, std::function objects can refer to any callable object, i.e., to anything that can be invoked like a function.*” Concept “callable object” definition. What is callable object that is not function? Intbu.*

*Function pointer (what is its using context?) and std::function description: “*Just as you must specify the type of function to point to when you create a function pointer (i.e., the signature of the functions you want to point to), you must specify the type of function to refer to when you create a std::function object. You do that through std::function’s template parameter. For example, to declare a std::function object named func that could refer to any callable object acting as if it had this signature,

bool(const std::unique\_ptr<Widget>&, // C++11 signature for

const std::unique\_ptr<Widget>&) // std::unique\_ptr<Widget>

// comparison function

you’d write this:

std::function<bool(const std::unique\_ptr<Widget>&,

const std::unique\_ptr<Widget>&)> func;*”*

*Lambda expressions are callable objects: “*Because lambda expressions yield callable objects, closures can be stored in std::function objects. That means we could declare the C++11 version of derefUP Less without using Auto as follows:

std::function<bool(const std::unique\_ptr<Widget>&,

const std::unique\_ptr<Widget>&)>

derefUPLess = [](const std::unique\_ptr<Widget>& p1,

const std::unique\_ptr<Widget>& p2)

{ return \*p1 < \*p2; };*”*

*Difference between auto and std::function: memory use: “*It’s important to recognize that even setting aside the syntactic verbosity and need to repeat the parameter types, using std::function is not the same as using auto. An auto-declared variable holding a closure has the same type as the closure, and as such it uses only as much memory as the closure requires. The type of a std::functiondeclared variable holding a closure is an instantiation of the std::function template, and that has a fixed size for any given signature. This size may not be adequate for the closure it’s asked to store, and when that’s the case, the std::function constructor will allocate heap memory to store the closure. The result is that the std::function object typically uses more memory than the auto-declared object.*”*

*p. 58*

*comparison by performance: “*And, thanks to implementation details that restrict inlining and yield indirect function calls, invoking a closure via a std::function object is almost certain to be slower than calling it via an auto-declared object. In other words, the std::function approach is generally bigger and slower than the auto approach, and it may yield out-of-memory exceptions, too.*” Exception kinds – out of memory. Potential danger of std::function. C++ problem kind.*

*Comparison by work: “*Plus, as you can see in the examples above, writing “auto” is a whole lot less work than writing the type of the std::function instantiation.*”*

*Total advantage of auto over std::function: “*In the competition between auto and std::function for holding a closure, it’s pretty much game, set, and match for auto.*”*

*Comparison of std::bind (what is it?) and lambda expressions: “*(A similar argument can be made for auto over std::functionfor holding the result of calls to std::bind, but in Item 34, I do my best to convince you to use lambdas instead of std::bind, anyway.)*” book connection (Item 34).*

*3 advantages of auto: “*The advantages of auto extend beyond the avoidance of uninitialized variables, verbose variable declarations, and the ability to directly hold closures.*” What does third mean?*

*C++ problem “type shortcuts” leading to deterioration of portability of C++-programs: “*One is the ability to avoid what I call problems related to “type shortcuts.” Here’s something you’ve probably seen—possibly even written:

std::vector<int> v;

…

unsigned sz = v.size();

The official return type of v.size() is std::vector<int>::size\_type, but few developers are aware of that. std::vector<int>::size\_type is specified to be an unsigned integral type, so a lot of programmers figure that unsigned is good enough and write code such as the above. This can have some interesting consequences. On 32-bit Windows, for example, both unsigned and std::vector<int>::size\_type are the same size, but on 64-bit Windows, unsigned is 32 bits, while std::vector<int>::size\_type is 64 bits. This means that code that works under 32-bit Windows may behave incorrectly under 64-bit Windows, and when porting your application from 32 to 64 bits, who wants to spend time on issues like that?*” One of the problems is different size of types in different systems. Qt types solve this problem. Прыклад памылкі, якая патрабуе вялікіх намаганняў па выпраўленні (час уходзіць не на работу, а на пошук прычыны), але значымасць якой таго не каштуе.*

*Mayers strongly recommends using auto.*

*Another example: “*Still unsure about the wisdom of using auto? Then consider this code:

std::unordered\_map<std::string, int> m;

…

for (const std::pair<std::string, int>& p : m)

{

… // do something with p

}

Recognizing what’s amiss requires remembering that the key part of a std::unor dered\_map is const, so the type of std::pair in the hash table (which is what a std::unordered\_map is) isn’t std::pair<std::string, int>, it’s std::pair <const std::string, int>. But that’s not the type declared for the variable p in the loop above. As a result, compilers will strive to find a way to convert std::pair<const std::string, int> objects (i.e., what’s in the hash table) to std::pair<std::string, int> objects (the declared type for p). They’ll succeed by creating a temporary object of the type that p wants to bind to by copying each object in m, then binding the reference p to that temporary object. At the end of each loop iteration, the temporary object will be destroyed. If you wrote this loop, you’d likely be surprised by this behavior, because you’d almost certainly intend to simply bind the reference p to each element in m. Such unintentional type mismatches can be autoed away:

for (const auto& p : m)

{

… // as before

}*” compiler work. Temporary objects for type conversion. C++ feature: type conversion. Auto advantage. Book example.*

*p. 59*

*std::unordered\_map is hash table.*

*Easier to type – advantage of auto.*

*Example of very subtle mistake: “*This is not only more efficient, it’s also easier to type. Furthermore, this code has the very attractive characteristic that if you take p’s address, you’re sure to get a pointer to an element within m. In the code not using auto, you’d get a pointer to a temporary object—an object that would be destroyed at the end of the loop iteration.

*07.12.2016*

*Dangerous of explicit type specifying: dangerous implicit type conversions: “*The last two examples—writing unsigned when you should have written std::vector<int>::size\_type and writing std::pair<std::string, int> when you should have written std::pair<const std::string, int>—demonstrate how explicitly specifying types can lead to implicit conversions that you neither want nor expect. If you use auto as the type of the target variable, you need not worry about mismatches between the type of variable you’re declaring and the type of the expression used to initialize it.*” Advantage of auto type deduction.*

*Auto nuances: Item 2 and Item 6: “*There are thus several reasons to prefer auto over explicit type declarations. Yet auto isn’t perfect. The type for each auto variable is deduced from its initializing expression, and some initializing expressions have types that are neither anticipated nor desired. The conditions under which such cases arise, and what you can do about them, are discussed in Items 2 and 6, so I won’t address them here. Instead,*”*

*Item nuance: readability of sourcing code: “*I’ll turn my attention to a different concern you may have about using auto in place of traditional type declarations: the readability of the resulting source code.*”*

*p. 60*

*contexts of using auto and explicit type declaration: “*First, take a deep breath and relax. Auto is an option, not a mandate. If, in your professional judgment, your code will be clearer or more maintainable or in some other way better by using explicit type declarations, you’re free to continue using them. But bear in mind that C++ breaks no new ground in adopting what is generally known in the programming languages world as type inference.*” C++ feature. What is type interface?*

*Classification and properties of programming languages: “*Other statically typed procedural languages (e.g., C#, D, Scala, Visual Basic) have a more or less equivalent feature, to say nothing of a variety of statically typed functional languages (e.g., ML, Haskell, OCaml, F#, etc.). In part, this is due to the success of dynamically typed languages such as Perl, Python, and Ruby, where variables are rarely explicitly typed.*” Type interface in different programming languages?*

*Argument to auto: what is type interface? : “*The software development community has extensive experience with type inference, and it has demonstrated that there is nothing contradictory about such technology and the creation and maintenance of large, industrial-strength code bases.*” Why auto is type interface feature? Type designation.*

*Aspects of programmer mental model and variable names: aspect of representation: “*Some developers are disturbed by the fact that using auto eliminates the ability to determine an object’s type by a quick glance at the source code. However, IDEs’ ability to show object types often mitigates this problem (even taking into account the IDE type-display issues mentioned in Item 4), and, in many cases, a somewhat abstract view of an object’s type is just as useful as the exact type. It often suffices, for example, to know that an object is a container or a counter or a smart pointer, without knowing exactly what kind of container, counter, or smart pointer it is. Assuming well-chosen variable names, such abstract type information should almost always be at hand.*” Think about with examples. Book connection – with Item 4.*

*Comment: auto is especially useful in cycles. But for better readability typedef is sometimes better. Relations between auto and typedef.*

*Auto purpose, property and pitfalls: “*Things to Remember • auto variables must be initialized, are generally immune to type mismatches that can lead to portability or efficiency problems, can ease the process of refactoring, and typically require less typing than variables with explicitly specified types.• auto-typed variables are subject to the pitfalls described in Items 2 and 6.*”.*

*Аспекты маёй сістэмы: як кампутару адрозніць, дзе ў вышэй змешчаным сказе ўласцівасць, дзе падводны камень, а дзе прызначэнне?*

*p. 61*

### **Item 6: Use the explicitly typed initializer idiom when auto deduces undesired types**

*Using context of explicit typing. Explicitly type initializer is type of initializer.*

*Auto pitfalls: “*Item 5 explains that using auto to declare variables offers a number of technical advantages over explicitly specifying types, but sometimes auto’s type deduction zigs when you want it to zag.*” Book connection: Item 5. Book structure: there will be examples of auto unpredictable work.*

*Патэрн праектавання “праверка структуры”: “*For example, suppose I have a function that takes a Widget and returns a std::vector<bool>, where each bool indicates whether the Widget offers a particular feature:

std::vector<bool> features(const Widget& w);*”*

*Then example of unpredictable auto work is shown and connected with some “interesting” properties of std::vector<bool>: “*For example, suppose I have a function that takes a Widget and returns a std::vector<bool>, where each bool indicates whether the Widget offers a particular feature:

std::vector<bool> features(const Widget& w);

Further suppose that bit 5 indicates whether the Widget has high priority. We can thus write code like this:

Widget w;

…

bool highPriority = features(w)[5]; // is w high priority?

…

processWidget(w, highPriority); // process w in accord

// with its priority

There’s nothing wrong with this code. It’ll work fine. But if we make the seemingly innocuous change of replacing the explicit type for highPriority with auto,

auto highPriority = features(w)[5]; // is w high priority?

the situation changes. All the code will continue to compile, but its behavior is no longer predictable:

processWidget(w, highPriority); // undefined behavior!

As the comment indicates, the call to processWidget now has undefined behavior. But why? The answer is likely to be surprising. In the code using auto, the type of highPriority is no longer bool. Though std::vector<bool> conceptually holds bools, operator[] for std::vector<bool> doesn’t return a reference to an element of the container (which is what std::vector::operator[] returns for every type except bool). Instead, it returns an object of type std::vector<bool>::reference (a class nested inside std::vector<bool>). std::vector<bool>::reference exists because std::vector<bool> is specified to represent its bools in packed form, one bit per bool. That creates a problem for std::vector<bool>’s operator[], because operator[] for std::vector<T> is supposed to return a T&, but C++ forbids references to bits. bool&, operator[] for std::vector<bool> returns an object that acts like a bool&. For this act to succeed, std::vector<bool>::reference objects must be usable inessentially all contexts wherebool&s can be. Among the features in std::vector<bool>::reference that make this work is an implicit conversion to bool. (Not to bool&, to bool. To explain the full set of techniques used by std::vector<bool>::reference to emulate the behavior of a bool& would take us too far afield, so I’ll simply remark that this implicit conversion is only one stone in a larger mosaic.)*” C++ property: C++ forbids references to bits. Why?*

*p. 62*

*this example is described further.*

*“…* data structure holding the bits that are managed by temp…*” – designation of object.*

*p. 63*

*example of dangling pointer: address of place data from which were deleted.*

*Design pattern proxy class: “*a class that exists for the purpose of emulating and augmenting the behavior of some other type. Proxy classes are employed for a variety of purposes.*” Context of its implementation. Example of it: “… exists to offer the illusion that operator[] for std::vector<bool> returns a reference to a bit”.*

*Book structure: reference to Chapter 4: “Standard Library’s smart pointer types are proxy classes that graft resource management onto raw pointers”. Smart pointers are Proxy.*

*Spread of using: “*In fact, the design pattern “Proxy” is one of the most longstanding members of the software design patterns Pantheon.*”*

*“*Some proxy classes are designed to be apparent to clients. That’s the case for std::shared\_ptr and std::unique\_ptr, for example. Other proxy classes are designed to act more or less invisibly. std::vector<bool>::reference is an example of such “invisible” proxies, as is its std::bitset compatriot, std::bitset::reference.*” Kinds of Proxy. What is std::bitset and std::bitset::reference?*

*Also in that camp…*

*p. 63*

*08.12.2016*

*Programming pattern: expression templates: “*Also in that camp are some classes in C++ libraries employing a technique known as expression templates. Such libraries were originally developed to improve the efficiency of numeric code. Given a class Matrix and Matrix objects m1, m2, m3, and m4, for example, the expression Matrix sum = m1 + m2 + m3 + m4;

can be computed much more efficiently if operator+ for Matrix objects returns a proxy for the result instead of the result itself. That is, operator+ for two Matrix objects would return an object of a proxy class such as Sum<Matrix, Matrix> instead of a Matrix object.*” Proxy class using. It is necessary to better understand this example and this technique. What is advantage of proxy class using.*

*Question: how can I provide implicit conversion?*

*Example of bad types for clients: “*As was the case with std::vector<bool>::reference and bool, there’d be an implicit conversion from the proxy class to Matrix, which would permit the initialization of sum from the proxy object produced by the expression on the right side of the “=”. (The type of that object would traditionally encode the entire initialization expression, i.e., be something like Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix>. That’s definitely a type from which clients should be shielded.)*” Why this example of expression templates is more efficient that return of matrix?*

*Auto implementation context: “*As a general rule, “invisible” proxy classes don’t play well with auto. Objects of such classes are often not designed to live longer than a single statement, so creating variables of those types tends to violate fundamental library design assumptions.*”*

*p. 64*

*Mayers recommendation: “*You therefore want to avoid code of this form:

auto someVar = expression of "invisible" proxy class type;*”*

*book task: “*But how can you recognize when proxy objects are in use?*”*

*“*Let’s take the how-do-you-find-them question first. Although “invisible” proxy classes are designed to fly beneath programmer radar in day-to-day use, libraries using them often document that they do so. The more you’ve familiarized yourself with the basic design decisions of the libraries you use, the less likely you are to be blindsided by proxy usage within those libraries. Where documentation comes up short, header files fill the gap. It’s rarely possible for source code to fully cloak proxy objects. They’re typically returned from functions that clients are expected to call, so function signatures usually reflect their existence.*” Two ways of proxy classes search: documentation and source code. There is example of it. Importance of source code investigation.*

*Error and auto implementation context: “*In practice, many developers discover the use of proxy classes only when they try to track down mystifying compilation problems or debug incorrect unit test results. Regardless of how you find them, once auto has been determined to be deducing the type of a proxy class instead of the type being proxied, the solution need not involve abandoning auto. Auto itself isn’t the problem.*”*

*p. 65*

*The explicitly typed initializer idiom: “*The explicitly typed initializer idiom involves declaring a variable with auto, but casting the initialization expression to the type you want auto to deduce. Here’s how it can be used to force highPriority to be a bool, for example:

auto highPriority = static\_cast<bool>(features(w)[5]);

Here, features(w)[5] continues to return a std::vector<bool>::reference object, just as it always has, but the cast changes the type of the expression to bool, which auto then deduces as the type for highPriority. At runtime, the std::vector<bool>::reference object returned from std::vector<bool>::operator[] executes the conversion to bool that it supports, and as part of that conversion, the still-valid pointer to the std::vector<bool> returned from features is dereferenced. That avoids the undefined behavior we ran into earlier. The index 5 is then applied to the bits pointed to by the pointer, and the bool value that emerges is used to initialize highPriority.*” Idiom definition. This is C++ technique. Not pattern with objects like GoF.*

*Implementation context of this idiom: “*Applications of the idiom aren’t limited to initializers yielding proxy class types. It can also be useful to emphasize that you are deliberately creating a variable of a type that is different from that generated by the initializing expression.*” Then there are examples which show that “*The explicitly typed initializer idiom makes things transparent*”. We see, that we convert types.*

*p. 66*

*“*Things to Remember • “Invisible” proxy types can cause auto to deduce the “wrong” type for an initializing expression. • The explicitly typed initializer idiom forces auto to deduce the type you want it to have.

*p. 67*

## ***Chapter 3. Moving to Modern C++***

*C++ evolution and history: “*When it comes to big-name features, C++11 and C++14 have a lot to boast of. auto, smart pointers, move semantics, lambdas, concurrency—each is so important, I devote a chapter to it.*” 5 modern C++ features.*

*Book structure, dynamic of thinking and questions: “*It’s essential to master those features, but becoming an effective modern C++ programmer requires a series of smaller steps, too. Each step answers specific questions that arise during the journey from C++98 to modern C++. When should you use braces instead of parentheses for object creation? Why are alias declarations better than typedefs? How does constexpr differ from const? What’s the relationship between const member functions and thread safety?*” what is alias declaration? What is constexpr? What is thread safety functions?*

### **Item 7: Distinguish between () and {} when creating objects**

*Parentheness and braces – C++ operator.*

*They are with object initializers. Their difference in this context.*

*Pittfal of C++11 using: “*Depending on your perspective, syntax choices for object initialization in C++11 embody either an embarrassment of riches or a confusing mess.*”*

*Three types of initialization values specification and one combined: “*As a general rule, initialization values may be specified with parentheses, an equals sign, or braces:

int x(0); // initializer is in parentheses

int y = 0; // initializer follows "="

int z{ 0 }; // initializer is in braces

In many cases, it’s also possible to use an equals sign and braces together:

int z = { 0 }; // initializer uses "=" and braces*”*

*Equality of initialization specification: “*For the remainder of this Item, I’ll generally ignore the equals-sign-plus-braces syntax, because C++ usually treats it the same as the braces-only version.*”*

*p. 68*

*initialization and assignment. Their difference in cases: built-in types and user defined types. Importance of difference in the last case: “*The “confusing mess” lobby points out that the use of an equals sign for initialization often misleads C++ newbies into thinking that an assignment is taking place, even though it’s not. For built-in types like int, the difference is academic, but for userdefined types, it’s important to distinguish initialization from assignment, because different function calls are involved:

Widget w1; // call default constructor

Widget w2 = w1; // not an assignment; calls copy ctor

w1 = w2; // an assignment; calls copy operator=

*” Cases of default constructor, copy constructor and operator= calls.*

*Disadvantage of C++98: “*Even with several initialization syntaxes, there were some situations where C++98 had no way to express a desired initialization. For example, it wasn’t possible to directly indicate that an STL container should be created holding a particular set of values (e.g., 1, 3, and 5).*” This is a big problem.*

*Initialization senarios – concept of programming languages.*

*C++ history and C++ feature – uniform initialization: “*To address the confusion of multiple initialization syntaxes, as well as the fact that they don’t cover all initialization scenarios, C++11 introduces uniform initialization: a single initialization syntax that can, at least in concept, be used anywhere and express everything. It’s based on braces, and for that reason I prefer the term braced initialization. “Uniform initialization” is an idea. “Braced initialization” is a syntactic construct.*” Braced initialization is realization of idea “uniform initialization”.*

*Example of braced initialization: “*Braced initialization lets you express the formerly inexpressible. Using braces, specifying the initial contents of a container is easy:

std::vector<int> v{ 1, 3, 5 }; // v's initial content is 1, 3, 5

Braces can also be used to specify default initialization values for non-static data members. This capability—new to C++11—is shared with the “=” initialization syntax, but not with parentheses:

class Widget {

…

private:

int x{ 0 }; // fine, x's default value is 0

int y = 0; // also fine

int z(0); // error!

};*” to implementation context of braced initialization.*

*Uncopiable objects – kind of objects. What is std::atomic ? : “*On the other hand, uncopyable objects (e.g., std::atomics—see Item 40) may be initialized using braces or parentheses, but not using “=”:

std::atomic<int> ai1{ 0 }; // fine

std::atomic<int> ai2(0); // fine

std::atomic<int> ai3 = 0; // error!*” implementation context of operator=, parentheness and braced initializer. Book structure – connection with Item 40.*

*p. 69*

*braced initialization description: “*It’s thus easy to understand why braced initialization is called “uniform.” Of C++’s three ways to designate an initializing expression, only braces can be used everywhere.*”*

*Brace initialization property: “*A novel feature of braced initialization is that it prohibits implicit narrowing conversions among built-in types. If the value of an expression in a braced initializer isn’t guaranteed to be expressible by the type of the object being initialized, the code won’t compile:

double x, y, z;

…

int sum1{ x + y + z }; // error! sum of doubles may

// not be expressible as int

Initialization using parentheses and “=” doesn’t check for narrowing conversions, because that could break too much legacy code:

int sum2(x + y + z); // okay (value of expression

// truncated to an int)

int sum3 = x + y + z; // ditto*” What is expressibility of value? What is narrowing type conversion?*

*Braced initialization property: “*Another noteworthy characteristic of braced initialization is its immunity to C++’s most vexing parse.*”*

*What is most vexing parse? Example of it and subtle error of C++: “*A side effect of C++’s rule that anything that can be parsed as a declaration must be interpreted as one, the most vexing parse most frequently afflicts developers when they want to default-construct an object, but inadvertently end up declaring a function instead. The root of the problem is that if you want to call a constructor with an argument, you can do it like this,

Widget w1(10); // call Widget ctor with argument 10

but if you try to call a Widget constructor with zero arguments using the analogous syntax, you declare a function instead of an object:

Widget w2(); // most vexing parse! declares a function

// named w2 that returns a Widget!

Functions can’t be declared using braces for the parameter list, so defaultconstructing an object using braces doesn’t have this problem:

Widget w3{}; // calls Widget ctor with no args

*” C++ rule. Braced initialization implementation context.*

*p. 70*

*Book question: “*There’s thus a lot to be said for braced initialization. It’s the syntax that can be used in the widest variety of contexts, it prevents implicit narrowing conversions, and it’s immune to C++’s most vexing parse. A trifecta of goodness! So why isn’t this Item entitled something like “Prefer braced initialization syntax”?*”*

*drawback of braced initialization: “*The drawback to braced initialization is the sometimes-surprising behavior that accompanies it.*”*

*C++ structure: “*Such behavior grows out of the unusually tangled relationship among braced initializers, std::initializer\_lists, and constructor overload resolution.*” what is constructor overload resolution and std::initializer\_lists?*

*C++ pitfall: “*Their interactions can lead to code that seems like it should do one thing, but actually does another.*” Disadvantage of taggled connections.*

*Braced initialization drawback. Relationship with auto in the context of their using: “*For example, Item 2 explains that when an auto-declared variable has a braced initializer, the type deduced is std::initializer\_list, even though other ways of declaring a variable with the same initializer would yield a more intuitive type. As a result, the more you like auto, the less enthusiastic you’re likely to be about braced initialization.*” book structure (Item 2).*

*С++ behavior of braced initialization while class has constructor with parameter std::initializer\_list: “*In constructor calls, parentheses and braces have the same meaning as long as std::initializer\_list parameters are not involved:

class Widget {

public:

Widget(int i, bool b); // ctors not declaring

Widget(int i, double d); // std::initializer\_list params

…

};

Widget w1(10, true); // calls first ctor

Widget w2{10, true}; // also calls first ctor

Widget w3(10, 5.0); // calls second ctor

Widget w4{10, 5.0}; // also calls second ctor

If, however, one or more constructors declare a parameter of type std::initial izer\_list, calls using the braced initialization syntax strongly prefer the overloads taking std::initializer\_lists. Strongly. If there’s any way for compilers to construe a call using a braced initializer to be to a constructor taking a std::initial izer\_list, compilers will employ that interpretation. If the Widget class above is augmented with a constructor taking a std::initializer\_list<long double>, for example,

class Widget {

public:

Widget(int i, bool b); // as before

Widget(int i, double d); // as before

Widget(std::initializer\_list<long double> il); // added

…

};

Widgets w2 and w4 will be constructed using the new constructor, even though the type of the std::initializer\_list elements (long double) is, compared to the non-std::initializer\_list constructors, a worse match for both arguments! Look:

Widget w1(10, true); // uses parens and, as before,

// calls first ctor

Widget w2{10, true}; // uses braces, but now calls

// std::initializer\_list ctor

// (10 and true convert to long double)

Widget w3(10, 5.0); // uses parens and, as before,

// calls second ctor

Widget w4{10, 5.0}; // uses braces, but now calls

// std::initializer\_list ctor

// (10 and 5.0 convert to long double)

*” problem demonstration.*

*p. 71*

*what is copy and move construction in C++?*

*Example of type convertion: “*Even what would normally be copy and move construction can be hijacked by

std::initializer\_list constructors:

class Widget {

public:

Widget(int i, bool b); // as before

Widget(int i, double d); // as before

Widget(std::initializer\_list<long double> il); // as before

operator float() const; // convert

… // to float

};*”*

*another example of braced constructor drawback: “*Even what would normally be copy and move construction can be hijacked by std::initializer\_list constructors:

class Widget {

public:

Widget(int i, bool b); // as before

Widget(int i, double d); // as before

Widget(std::initializer\_list<long double> il); // as before

operator float() const; // convert

… // to float

};

Widget w5(w4); // uses parens, calls copy ctor

Widget w6{w4}; // uses braces, calls

// std::initializer\_list ctor

// (w4 converts to float, and float

// converts to long double)

Widget w7(std::move(w4)); // uses parens, calls move ctor

Widget w8{std::move(w4)}; // uses braces, calls

// std::initializer\_list ctor

// (for same reason as w6)*” example of moving construction.*

*Compiler peculiarities and braced initialization disadvantage: “*Compilers’ determination to match braced initializers with constructors taking std::initializer\_lists is so strong, it prevails even if the best-match std::initializer\_list constructor can’t be called. For example:

class Widget {

public:

Widget(int i, bool b); // as before

Widget(int i, double d); // as before

Widget(std::initializer\_list<bool> il); // element type is

// now bool

… // no implicit

}; // conversion funcs

Widget w{10, 5.0}; // error! requires narrowing conversions*”*

*For exaplanation of narrowing type conversion: “*Calling that constructor would require converting. An int (10) and a double (5.0) to bools. Both conversions would be narrowing (bool can’t exactly represent either value), and narrowing conversions are prohibited inside braced initializers, so the call is invalid, and the code is rejected.*” what kinds of narrowing conversion exist?*

*Compiler behavior and braced initialization drawback: “*Only if there’s no way to convert the types of the arguments in a braced initializer to the type in a std::initializer\_list do compilers fall back on normal overload resolution. For example, if we replace the std::initializer\_list<bool> constructor with one taking a std::initializer\_list<std::string>, the nonstd::initializer\_list constructors become candidates again, because there is no way to convert ints and bools to std::strings:

class Widget {

public:

Widget(int i, bool b); // as before

Widget(int i, double d); // as before

// std::initializer\_list element type is now std::string

Widget(std::initializer\_list<std::string> il);

… // no implicit

}; // conversion funcs

Widget w1(10, true); // uses parens, still calls first ctor

Widget w2{10, true}; // uses braces, now calls first ctor

Widget w3(10, 5.0); // uses parens, still calls second ctor

Widget w4{10, 5.0}; // uses braces, now calls second ctor*”*

*What is std::initialization\_list. What its purpose?*

*p. 73*

*this item devoted to braced initialization and constructor overloading.*

*Overloading is feature of C++ and general feature of programming.*

*Braced initialization drawback: “*Suppose you use an empty set of braces to construct an object that supports default construction and also supports std::initializer\_list construction. What do your empty braces mean? If they mean “no arguments,” you get default construction, but if they mean “empty std::initializer\_list,” you get construction from a std::initializer\_list with no elements. *It calls default constructor.” Many drawbacks connects with std::initializer\_list.*

*C++ task and its solution: “*If you want to call a std::initializer\_list constructor with an empty std::initializer\_list, you do it by making the empty braces a constructor argument—by putting the empty braces inside the parentheses or braces demarcating what you’re passing:

Widget w4({}); // calls std::initializer\_list ctor

// with empty list

Widget w5{{}}; // ditto

*” std::initializer\_list using technique.*

*p. 74*

*braced initialization drawbacks directly concerns std::vector class: “*At this point, with seemingly arcane rules about braced initializers, std::initial izer\_lists, and constructor overloading burbling about in your brain, you may be wondering how much of this information matters in day-to-day programming. More than you might think, because one of the classes directly affected is std::vector. std::vector has a non-std::initializer\_list constructor that allows you to specify the initial size of the container and a value each of the initial elements should have, but it also has a constructor taking a std::initializer\_list that permits you to specify the initial values in the container. If you create a std::vector of a numeric type (e.g., a std::vector<int>) and you pass two arguments to the constructor, whether you enclose those arguments in parentheses or braces makes a tremendous difference:

std::vector<int> v1(10, 20); // use non-std::initializer\_list

// ctor: create 10-element

// std::vector, all elements have

// value of 20

std::vector<int> v2{10, 20}; // use std::initializer\_list ctor:

// create 2-element std::vector,

// element values are 10 and 20*”*

*Book recommendation and C++ programming technique concerning constructor overloading: “*But let’s step back from std::vector and also from the details of parentheses, braces, and constructor overloading resolution rules. There are two primary takeaways from this discussion. First, as a class author, you need to be aware that if your set of overloaded constructors includes one or more functions taking a std::initializer\_list, client code using braced initialization may see only the std::initializer\_list overloads. As a result, it’s best to design your constructors so that the overload called isn’t affected by whether clients use parentheses or braces. In other words, learn from what is now viewed as an error in the design of the std::vector interface, and design your classes to avoid it. An implication is that if you have a class with no std::initializer\_list constructor, and you add one, client code using braced initialization may find that calls that used to resolve to non std::initializer\_list constructors now resolve to the new function. Of course, this kind of thing can happen any time you add a new function to a set of overloads: calls that used to resolve to one of the old overloads might start calling the new one. The difference with std::initializer\_list constructor overloads is that a std::initializer\_list overload doesn’t just compete with other overloads, it overshadows them to the point where the other overloads may hardly be considered. So add such overloads only with great deliberation.*”*

The second lesson is that…

p. 74

*знізу старонкі*

*12.12.2016*

*C++ using technique: “*The second lesson is that as a class client, you must choose carefully between parentheses and braces when creating objects.*”*

*p. 75*

*comparison of braces and parentheness: “*Braces-by-default folks are attracted by their unrivaled breadth of applicability, their prohibition of narrowing conversions, and their immunity to C++’s most vexing parse. Such folks understand that in some cases (e.g., creation of a std::vector with a given size and initial element value), parentheses are required. On the other hand, the goparenthesesgo crowd embraces parentheses as their default argument delimiter. They’re attracted to its consistency with the C++98 syntactic tradition, its avoidance of the auto-deduced-a-std::initializer\_list problem, and the knowledge that their object creation calls won’t be inadvertently waylaid by std::initializer\_list constructors. They concede that sometimes only braces will do (e.g., when creating a container with particular values). There’s no consensus that either approach is better than the other, so my advice is to pick one and apply it consistently.*” Author recommendation. Example of using context where there is impossible to choose something definite.*

*Difficulty of choosing between two mechanisms: nuances of template using: “*Braces-by-default folks are attracted by their unrivaled breadth of applicability, their prohibition of narrowing conversions, and their immunity to C++’s most vexing parse. Such folks understand that in some cases (e.g., creation of a std::vector with a given size and initial element value), parentheses are required. On the other hand, the goparenthesesgo crowd embraces parentheses as their default argument delimiter. They’re attracted to its consistency with the C++98 syntactic tradition, its avoidance of the auto-deduced-a-std::initializer\_list problem, and the knowledge that their object creation calls won’t be inadvertently waylaid by std::initializer\_list constructors. They concede that sometimes only braces will do (e.g., when creating a container with particular values). There’s no consensus that either approach is better than the other, so my advice is to pick one and apply it consistently. If doSomeWork uses parentheses when creating localObject, the result is a std::vector with 10 elements. If doSomeWork uses braces, the result is a std::vector with 2 elements. Which is correct? The author of doSomeWork can’t know. Only the caller can.*”. C++ concept – variadic template. Book structure (Item 25).*

*p. 76*

*what is std::make\_unique and std::make\_shared? This problem (C++ problem) is connected with these concepts: “*This is precisely the problem faced by the Standard Library functions std::make\_unique and std::make\_shared (see Item 21). These functions resolve the problem by internally using parentheses and by documenting this decision as part of their interfaces.*” Book structure (Item 21).*

*Book structure. Item resume: “*Things to Remember • Braced initialization is the most widely usable initialization syntax, it prevents narrowing conversions, and it’s immune to C++’s most vexing parse. • During constructor overload resolution, braced initializers are matched to std::initializer\_list parameters if at all possible, even if other constructors offer seemingly better matches. An example of where the choice between parentheses and braces can make a significant difference is creating a std::vector<numeric type> with two arguments. Choosing between parentheses and braces for object creation inside templates can be challenging.*”*

**Item 8: Prefer nullptr to 0 and NULL**

*C++ structure and interpretation of pointer. Aspect of zero pointer. Conversion int to zero pointer: “*So here’s the deal: the literal 0 is an int, not a pointer. If C++ finds itself looking at 0 in a context where only a pointer can be used, it’ll grudgingly interpret 0 as a null pointer, but that’s a fallback position. C++’s primary policy is that 0 is an int, not a pointer.*”*

*NULL structure: “*Practically speaking, the same is true of NULL. There is some uncertainty in the details in NULL’s case, because implementations are allowed to give NULL an integral type other than int (e.g., long). That’s not common, but it doesn’t really matter, because the issue here isn’t the exact type of NULL, it’s that neither 0 nor NULL has a pointer type.*”*

*Тут ёсць спасылка на вырашэнне праблемы выкарыстання розных тыпаў дужак у шаблонах.*

*p. 77*

*С++ 98 problem: “*In C++98, the primary implication of this was that overloading on pointer and integral types could lead to surprises. Passing 0 or NULL to such overloads never called a pointer overload:

void f(int); // three overloads of f

void f(bool);

void f(void\*);

f(0); // calls f(int), not f(void\*)

f(NULL); // might not compile, but typically calls

// f(int). Never calls f(void\*)*”*

*C++ history and recommendation: “*This counterintuitive behavior is what led to the guideline for C++98 programmers to avoid overloading on pointer and integral types. That guideline remains valid in C++11, because, the advice of this Item notwithstanding, it’s likely that some developers will continue to use 0 and NULL, even though nullptr is a better choice.*” Advantage of nullptr over NULL and 0.*

*Nullptr structure: “*nullptr’s advantage is that it doesn’t have an integral type. To be honest, it doesn’t have a pointer type, either, but you can think of it as a pointer of all types. nullptr’s actual type is std::nullptr\_t, and, in a wonderfully circular definition, std::nullptr\_t is defined to be the type of nullptr. The type std::nullptr\_t implicitly converts to all raw pointer types, and that’s what makes nullptr act as if it were a pointer of all types.*” What is raw pointer type? What is structure of this concept? It is necessary to better understand.*

*Advantage of nullptr and C++ error “overload resolution surprises”: “*nullptr’s advantage is that it doesn’t have an integral type. To be honest, it doesn’t have a pointer type, either, but you can think of it as a pointer of all types. nullptr’s actual type is std::nullptr\_t, and, in a wonderfully circular definition, std::nullptr\_t is defined to be the type of nullptr. The type std::nullptr\_t implicitly converts to all raw pointer types, and that’s what makes nullptr act as if it were a pointer of all types. Calling the overloaded function f with nullptr calls the void\*overload (i.e., the pointer overload), because nullptr can’t be viewed as anything integral:

f(nullptr); // calls f(void\*) overload

Using nullptr instead of 0 or NULL thus avoids overload resolution surprises, but that’s not its only advantage.*”*

*Usefulness of nullptr for auto variables: “*It can also improve code clarity, especially when Auto variables are involved. For example, suppose you encounter this in a code base:

auto result = findRecord( /\* arguments \*/ );

if (result == 0) {

…

}

If you don’t happen to know (or can’t easily find out) what findRecord returns, it may not be clear whether result is a pointer type or an integral type. After all, 0 (what result is tested against) could go either way. If you see the following, on the other hand,

auto result = findRecord( /\* arguments \*/ );

if (result == nullptr) {

…

}

there’s no ambiguity: result must be a pointer type.

*” Code clarity recommendation.*

*Using context of nullptr: usefulness with templates: “*nullptr shines especially brightly when templates enter the picture. Suppose you have some functions that should be called only when the appropriate mutex has been locked. Each function takes a different kind of pointer:

int f1(std::shared\_ptr<Widget> spw); // call these only when

double f2(std::unique\_ptr<Widget> upw); // the appropriate

bool f3(Widget\* pw); // mutex is locked

Calling code that wants to pass null pointers could look like this:

std::mutex f1m, f2m, f3m; // mutexes for f1, f2, and f3

using MuxGuard = // C++11 typedef; see Item 9

std::lock\_guard<std::mutex>;

…

{

MuxGuard g(f1m); // lock mutex for f1

auto result = f1(0); // pass 0 as null ptr to f1

} // unlock mutex

…

{

MuxGuard g(f2m); // lock mutex for f2

auto result = f2(NULL); // pass NULL as null ptr to f2

} // unlock mutex

…

{

MuxGuard g(f3m); // lock mutex for f3

auto result = f3(nullptr); // pass nullptr as null ptr to f3

} // unlock mutex

*” It is necessary to better understand this example. I don’t understand what is mutex. Book structure (Item 9).*

*p. 78*

*template purpose: “*The failure to use nullptr in the first two calls in this code is sad, but the code works, and that counts for something. However, the repeated pattern in the calling code—lock mutex, call function, unlock mutex—is more than sad. It’s disturbing. This kind of source code duplication is one of the things that templates are designed to avoid, so let’s templatize the pattern:

template<typename FuncType,

typename MuxType,

typename PtrType>

auto lockAndCall(FuncType func,

MuxType& mutex,

PtrType ptr) -> decltype(func(ptr))

{

MuxGuard g(mutex);

return func(ptr);

}

*” Itnbu. What is mutex?*

*p. 79*

*Remind what is decltype(auto): “*If the return type of this function (auto … -> decltype(func(ptr)) has you scratching your head, do your head a favor and navigate to Item 3, which explains what’s going on. There you’ll see that in C++14, the return type could be reduced to a simple decltype(auto):

template<typename FuncType,

typename MuxType,

typename PtrType>

decltype(auto) lockAndCall(FuncType func, // C++14

MuxType& mutex,

PtrType ptr)

{

MuxGuard g(mutex);

return func(ptr);

}

Given the lockAndCall template (either version), callers can write code like this:

auto result1 = lockAndCall(f1, f1m, 0); // error!

…

auto result2 = lockAndCall(f2, f2m, NULL); // error!

auto result3 = lockAndCall(f3, f3m, nullptr); // fine

*” itnbu. Itnbu this example. There is first example which I didn’t understand.*

*p. 80*

*I don’t understand this explanation of example: “*Well, they can write it, but, as the comments indicate, in two of the three cases, the code won’t compile. The problem in the first call is that when 0 is passed to lockAnd Call, template type deduction kicks in to figure out its type. The type of 0 is, was, and always will be int, so that’s the type of the parameter ptr inside the instantiation of this call to lockAndCall. Unfortunately, this means that in the call to func inside lockAndCall, an int is being passed, and that’s not compatible with the std::shared\_ptr<Widget> parameter that f1 expects. The 0 passed in the call to lockAndCall was intended to represent a null pointer, but what actually got passed was a run-of-the-mill int. Trying to pass this int to f1 as a std::shared\_ptr<Widget> is a type error. The call to lockAndCall with 0 fails because inside the template, an int

is being passed to a function that requires a std::shared\_ptr<Widget>. The analysis for the call involving NULL is essentially the same. When NULL is passed to lockAndCall, an integral type is deduced for the parameter ptr, and a type error occurs when ptr—an int or int-like type—is passed to f2, which expects to get a std::unique\_ptr<Widget>. In contrast, the call involving nullptr has no trouble. When nullptr is passed to lockAndCall, the type for ptr is deduced to be std::nullptr\_t. When ptr is passed to f3, there’s an implicit conversion from std::nullptr\_t to Widget\*, because std::nullptr\_t implicitly converts to all pointer types.*”*

*Advantage of nullptr and its advantage: “*The fact that template type deduction deduces the “wrong” types for 0 and NULL (i.e., their true types, rather than their fallback meaning as a representation for a null pointer) is the most compelling reason to use nullptr instead of 0 or NULL when you want to refer to a null pointer. With nullptr, templates pose no special challenge. Combined with the fact that nullptr doesn’t suffer from the overload resolution surprises that 0 and NULL are susceptible to, the case is ironclad. When you want to refer to a null pointer, use nullptr, not 0 or NULL.*”*

*Book structure. Resume: “*Things to Remember. Prefer nullptr to 0 and NULL. Avoid overloading on integral and pointer types.*”*

*p. 81*

## **Item 9: Prefer alias declarations to typedefs**

*Author opinion about STL containers and std::unique\_ptr: “*I’m confident we can agree that using STL containers is a good idea, and I hope that Item 18 convinces you that using std::unique\_ptr is a good idea, but my guess is that neither of us is fond of writing types like “std::unique\_ptr<std::unordered\_map<std::string, std::string>>” more than once. Just thinking about it probably increases the risk of carpal tunnel syndrome.*” Book structure (Item 18). Problem of code clarity in C++. STL property.*

*Typedef purpose and structure: “*Avoiding such medical tragedies is easy. Introduce a typedef:

typedef

std::unique\_ptr<std::unordered\_map<std::string, std::string>>

UPtrMapSS;*”*

*C++ history and alias declarations definition: “*But typedefs are soooo C++98. They work in C++11, sure, but C++11 also offers alias declarations:

using UPtrMapSS =

std::unique\_ptr<std::unordered\_map<std::string, std::string>>;*”*

*Book question: “*Given that the typedef and the alias declaration do exactly the same thing, it’s reasonable to wonder whether there is a solid technical reason for preferring one over the other.*”*

*Advantage of alias declaration: shortness for function pointers: “*There is, but before I get to it, I want to mention that many people find the alias declaration easier to swallow when dealing with types involving function pointers:

// FP is a synonym for a pointer to a function taking an int and

// a const std::string& and returning nothing

typedef void (\*FP)(int, const std::string&); // typedef

// same meaning as above

using FP = void (\*)(int, const std::string&); // alias

// declaration

Of course, neither form is particularly easy to choke down, and few people spend much time dealing with synonyms for function pointer types, anyway, so this is hardly a compelling reason to choose alias declarations over typedefs.*”*

*Alias declarations are better for templates: “*But a compelling reason does exist: templates. In particular, alias declarations may be templatized (in which case they’re called alias templates), while typedefs cannot. This gives C++11 programmers a straightforward mechanism for expressing things that in C++98 had to be hacked together with typedefs nested inside templatized structs.*” Alias declaration structure. C++ concept – alias template.*

*Demonstration example: “*For example, consider defining a synonym for a linked list that uses a custom allocator, MyAlloc. With an alias template, it’s a piece of cake:*” what is allocator?*

*p. 82*

*demonstration example continue: “*template<typename T> // MyAllocList<T>

using MyAllocList = std::list<T, MyAlloc<T>>; // is synonym for

// std::list<T,

// MyAlloc<T>>

MyAllocList<Widget> lw; // client code

With a typedef, you pretty much have to create the cake from scratch:

template<typename T> // MyAllocList<T>::type

struct MyAllocList { // is synonym for

typedef std::list<T, MyAlloc<T>> type; // std::list<T,

}; // MyAlloc<T>>

MyAllocList<Widget>::type lw; // client code

*” Why is it not possible to write “typedef* MyAllocList std::list<T, MyAlloc<T>>*”*

*Demonstration example: “*It gets worse. If you want to use the typedef inside a template for the purpose of creating a linked list holding objects of a type specified by a template parameter, you have to precede the typedef name with typename:

template<typename T>

class Widget { // Widget<T> contains

private: // a MyAllocList<T>

typename MyAllocList<T>::type list; // as a data member

…

};

Here, MyAllocList<T>::type refers to a type that’s dependent on a template type parameter (T). MyAllocList<T>::type is thus a dependent type, and one of C++’s many endearing rules is that the names of dependent types must be preceded by type name.*”. using context of typename – dependent type of class.*

*Demonstration example continue: “*If MyAllocList is defined as an alias template, this need for typename vanishes (as does the cumbersome “::type” suffix):

template<typename T>

using MyAllocList = std::list<T, MyAlloc<T>>; // as before

template<typename T>

class Widget {

private:

MyAllocList<T> list; // no "typename",

… // no "::type"

};*”*

*Dependent type. Compiler structure: “*To you, MyAllocList<T> (i.e., use of the alias template) may look just as dependent on the template parameter T as MyAllocList<T>::type (i.e., use of the nested Typedef), but you’re not a compiler. When compilers process the Widget template and encounter the use of MyAllocList<T> (i.e., use of the alias template), they know that MyAllocList<T> is the name of a type, because MyAllocList is an alias template: it must name a type. MyAllocList<T> is thus a non-dependent type, and a typename specifier is neither required nor permitted.*” Alias template purpose and structure. Using context of typename.*

*p. 83*

When compilers…

*13.12.2016*

*Dangerous of nested typedef: “*When compilers see MyAllocList<T>::type

(i.e., use of the nested typedef) in the Widget template, on the other hand, they can’t know for sure that it names a type, because there might be a specialization of MyAllocList that they haven’t yet seen Where MyAllocList<T>::type refers to something other than a type. That sounds crazy, but don’t blame compilers for this possibility. It’s the humans who have been known to produce such code.*”*

*Book example of nested typedef dangerous:*

*“*For example, some misguided soul may have concocted something like this:

class Wine { … };

template<> // MyAllocList specialization

class MyAllocList<Wine> { // for when T is Wine

private:

enum class WineType // see Item 10 for info on

{ White, Red, Rose }; // "enum class"

WineType type; // in this class, type is

… // a data member!

};

As you can see, MyAllocList<Wine>::type doesn’t refer to a type.*” Book structure. (Enum 10). What is enum classes? У дадзеным выпадку ідэнтыфікатар тайп звяртаецца не да назвы тыпа, а да члена класа. Conflict of type name and class member name – problem in C++.*

*For explanation of typename keyword: “*As you can see, MyAllocList<Wine>::type doesn’t refer to a type. If Widget were to be instantiated with Wine, MyAllocList<T>::type inside the Widget template would refer to a data member, not a type. Inside the Widget template, then, whether MyAllocList<T>::type refers to a type is honestly dependent on what T is, and that’s why compilers insist on your asserting that it is a type by preceding it with typename.*” I understood it.*

*Template metaprogramming (TMP) – C++ feature.*

*TMP programming situation: “*If you’ve done any template metaprogramming (TMP), you’ve almost certainly bumped up against the need to take template type parameters and create revised types from them.*”*

*Type conversion cases: “*you might want to strip off any constor reference-qualifiers that T contains*”, “*you might want to add const to a type*”, “*turn it into an lvalue reference*”. What is lvalue reference?*

*Book structure: TMP examples: Items 23 and 27.*

*C++ quality marker: “*If you haven’t done any TMP, that’s too bad, because if you want to be a truly effective C++ programmer, you need to be familiar with at least the basics of this facet of C++. You can see examples of TMP in action, including the kinds of type transformations I just mentioned, in Items 23 and 27.*” I need to better know it!*

*p. 84*

*C++ history and type traits mechanism: “*C++11 gives you the tools to perform these kinds of transformations in the form of type traits, an assortment of templates inside the header <type\_traits>. There are dozens of type traits in that header, and not all of them perform type transformations, but the ones that do offer a predictable interface. Given a type T to which you’d like to apply a transformation, the resulting type is std::transformation <T>::type. For example:

std::remove\_const<T>::type // yields T from const T

std::remove\_reference<T>::type // yields T from T& and T&&

std::add\_lvalue\_reference<T>::type // yields T& from T*” Structure of type traits mechanism.*

*type transformation is the same as type conversion.*

*Author says that it is better to read specification of type traits.*

*Typename using context and type traits: “*My motivation here isn’t to give you a tutorial on type traits, anyway. Rather, note that application of these transformations entails writing “::type” at the end of each use. If you apply them to a type parameter inside a template (which is virtually always how you employ them in real code), you’d also have to precede each use with type name. The reason for both of these syntactic speed bumps is that the C++11 type traits are implemented as nested typedefs inside templatized structs. That’s right, they’re implemented using the type synonym technology I’ve been trying to convince you is inferior to alias templates!*” advantage of alias declaration over typedef. What is type synonym technology?*

*C++ history. Alias templates advantage: “*There’s a historical reason for that, but we’ll skip over it (it’s dull, I promise), because the Standardization Committee belatedly recognized that alias templates are the better way to go, and they included such templates in C++14 for all the C++11 type transformations. The aliases have a common form: for each C++11 transformation std::transformation<T>::type, there’s a corresponding C++14 alias template named std::transformation\_t. Examples will clarify what I mean:

std::remove\_const<T>::type // C++11: const T → T

std::remove\_const\_t<T> // C++14 equivalent

std::remove\_reference<T>::type // C++11: T&/T&& → T

std::remove\_reference\_t<T> // C++14 equivalent

std::add\_lvalue\_reference<T>::type // C++11: T → T&

std::add\_lvalue\_reference\_t<T> // C++14 equivalent

*” This mechanism is realized by alias templates.*

*Метад разумення: імітаваць патэрн. Паразважаць. Гэта могуць рабіць дзеці. Я заву гэта – выконваць па ўзоры. Аспекты траз.*

*p. 85*

*It is necessary to better understand what does mean this code:*

*“*If you happen to have access to an electronic copy of the C++14 Standard, it’s easier still, because all that’s required is some copying and pasting. Here, I’ll get you started:

template <class T>

using remove\_const\_t = typename remove\_const<T>::type;

template <class T>

using remove\_reference\_t = typename remove\_reference<T>::type;

template <class T>

using add\_lvalue\_reference\_t =

typename add\_lvalue\_reference<T>::type;

See? Couldn’t be easier.*” Гэта тое, што ў стандарце? Паразважаць. Механізм у С++ 14 замяняе слова typename з пэўным канструктам сінонімам? Паразважаць.*

*Book structure. Resume:*

*“*

Things to Remember • typedefs don’t support templatization, but alias declarations do.

• Alias templates avoid the “::type” suffix and, in templates, the “typename” prefix often required to refer to typedefs.

• C++14 offers alias templates for all the C++11 type traits transformations.*” intbu why typedefs don’t support templatization and what is it?*

**Item 10: Prefer scoped enums to unscoped enums**

*What are scoped enums and what are unscoped enums? Intbu.*

*C++ feature “scope”. Braces purpose in the context of scope: “*As a general rule, declaring a name inside curly braces limits the visibility of that name to the scope defined by the braces.*”*

*C++ history and historical lack: “*Not so for the enumerators declared in C++98-style enums. The names of such enumerators belong to the scope containing the enum, and that means that nothing else in that scope may have the same name:

enum Color { black, white, red }; // black, white, red are

// in same scope as Color

auto white = false; // error! white already

// declared in this scope*” disadvantage and definition of unscoped enumerations.*

*“”*

*C++ history. Scoped and unscoped enumerations: “*The fact that these enumerator names leak into the scope containing their enum definition gives rise to the official term for this kind of enum: unscoped. Their new C++11 counterparts, scoped enums, don’t leak names in this way:

enum class Color { black, white, red }; // black, white, red

// are scoped to Color

auto white = false; // fine, no other

// "white" in scope

Color c = white; // error! no enumerator named

// "white" is in this scope

Color c = Color::white; // fine

auto c = Color::white; // also fine (and in accord

// with Item 5's advice)*” auto example. Book structure: Mayers uses in code through book all recommendations which he done.*

*p. 86*

*scoped enums synonym: “*Because scoped enums are declared via “enum class”, they’re sometimes referred to as enum classes.*”*

*C++ problem: namespace pollution. One of its components is unscoped enums.*

*What is strength of typing?*

*Recommendation of book and advantage of scoped enums over unscoped enums: “*The reduction in namespace pollution offered by scoped enums is reason enough to prefer them over their unscoped siblings, but scoped enums have a second compelling advantage: their enumerators are much more strongly typed. Enumerators for unscoped enums implicitly convert to integral types (and, from there, to floating-point types).*” Type strength is possibility of implicit type conversion.*

*Example of unscoped classes weakness typing: “*Semantic travesties such as the following are therefore completely valid:

enum Color { black, white, red }; // unscoped enum

std::vector<std::size\_t> // func. returning

primeFactors(std::size\_t x); // prime factors of x

Color c = red;

…

if (c < 14.5) { // compare Color to double (!)

auto factors = // compute prime factors

primeFactors(c); // of a Color (!)

…

}*”*

*C++ nuance: implicit type conversion is origin of erros. Example of it is upper.*

*Advantage of scoped enums: “*Throw a simple “class” after “enum”, however, thus transforming an unscoped enum into a scoped one, and it’s a very different story. There are no implicit conversions from enumerators in a scoped enum to any other type:

enum class Color { black, white, red }; // enum is now scoped

Color c = Color::red; // as before, but

… // with scope qualifier

if (c < 14.5) { // error! can't compare

// Color and double

auto factors = // error! can't pass Color to

primeFactors(c); // function expecting std::size\_t

…

}*”*

*p. 87*

*C++ structure: type system.*

*Purpose of static\_cast: “*If you honestly want to perform a conversion from Color to a different type, do what you always do to twist the type system to your wanton desires—use a cast:

if (static\_cast<double>(c) < 14.5) { // odd code, but

// it's valid

auto factors = // suspect, but

primeFactors(static\_cast<std::size\_t>(c)); // it compiles

…

}*”*

*C++ feature: forward declaration: “*It may seem that scoped enums have a third advantage over unscoped enums, because scoped enums may be forward-declared, i.e., their names may be declared without specifying their enumerators:

enum Color; // error!

enum class Color; // fine*” Я са свайго вопыту магу сказаць, што гэта вельмі добра! Scoped enumeration advantage.*

*C++ realization, enum, compiler structure: “*This is misleading. In C++11, unscoped enums may also be forward-declared, but only after a bit of additional work. The work grows out of the fact that every enum in C++ has an integral underlying type that is determined by compilers. For an unscoped enum like Color,

enum Color { black, white, red };

compilers might choose char as the underlying type, because there are only three values to represent. However, some enums have a range of values that is much larger, e.g.:

enum Status { good = 0,

failed = 1,

incomplete = 100,

corrupt = 200,

indeterminate = 0xFFFFFFFF

};

Here the values to be represented range from 0 to 0xFFFFFFFF. Except on unusual machines (where a char consists of at least 32 bits), compilers will have to select an integral type larger than char for the representation of Status values.*”*

*Unscoped enums in C++11 can be forward declared.*

*P. 88*

*Compiler structure: “*To make efficient use of memory, compilers often want to choose the smallest underlying type for an enum that’s sufficient to represent its range of enumerator values. In some cases, compilers will optimize for speed instead of size, and in that case, they may not choose the smallest permissible underlying type, but they certainly want to be able to optimize for size. To make that possible, C++98 supports only enum definitions (where all enumerators are listed); enum declarations are not allowed. That makes it possible for compilers to select an underlying type for each enum prior to the enum being used.*” C++ history and background of this C++ feature.*

*C++ cocenpt connected with compilation: compilation dependencies. Increase of them can be called by absence of enum forward declarations: “*But the inability to forward-declare enums has drawbacks. The most notable is probably the increase in compilation dependencies. Consider again the Status enum:

enum Status { good = 0,

failed = 1,

incomplete = 100,

corrupt = 200,

indeterminate = 0xFFFFFFFF

};

This is the kind of enum that’s likely to be used throughout a system, hence included in a header file that every part of the system is dependent on. If a new status value is then introduced,

enum Status { good = 0,

failed = 1,

incomplete = 100,

corrupt = 200,

audited = 500,

indeterminate = 0xFFFFFFFF

};

it’s likely that the entire system will have to be recompiled, even if only a single subsystem—possibly only a single function!—uses the new enumerator. This is the kind of thing that people hate. And it’s the kind of thing that the ability to forward-declare enums in C++11 eliminates.*” Aspects of compilation. Recompilation situation: when some part of code refers to header with code change. It is necessary to better understand this advantage of forward declarations.*

*Example of absence of this problem: “*For example, here’s a perfectly valid declaration of a scoped enum and a function that takes one as a parameter:

enum class Status; // forward declaration

void continueProcessing(Status s); // use of fwd-declared enum

The header containing these declarations requires no recompilation if Status’s definition is revised. Furthermore, if Status is modified (e.g., to add the audited enumerator), but continueProcessing’s behavior is unaffected (e.g., because continueProcessing doesn’t use audited), continueProcessing’s implementation need not be recompiled, either.*” Cases where there is no necessity of recompilation.*

*p. 89*

*But if compilers…*

*14.12.2016*

*Compiler and scoped enums structure: “*But if compilers need to know the size of an enum before it’s used, how can C++11’s enums get away with forward declarations when C++98’s enums can’t? The answer is simple: the underlying type for a scoped enum is always known, and for unscoped enums, you can specify it. By default, the underlying type for scoped enums is int:

enum class Status; // underlying type is int

If the default doesn’t suit you, you can override it:

enum class Status: std::uint32\_t; // underlying type for

// Status is std::uint32\_t

// (from <cstdint>)

Either way, compilers know the size of the enumerators in a scoped enum.*” Default type declaration of enumerators in scoped enum.*

*Unscoped enums also has possibility of default type declaration:*

*“*To specify the underlying type for an unscoped enum, you do the same thing as for a scoped enum, and the result may be forward-declared:

enum Color: std::uint8\_t; // fwd decl for unscoped enum;

// underlying type is

// std::uint8\_t*”*

*Default type declaration can be with scoped enum definition: “*Underlying type specifications can also go on an enum’s definition:

enum class Status: std::uint32\_t { good = 0,

failed = 1,

incomplete = 100,

corrupt = 200,

audited = 500,

indeterminate = 0xFFFFFFFF

};*”*

*Forward declaration is feature which show difference between declaration and definition.*

*Class is the thing which shows difference between definition and initialization.*

*Std::tuples structure. What is it? Using context of unscoped enums. Advantages of scoped enums: “*In view of the fact that scoped enums avoid namespace pollution and aren’t susceptible to nonsensical implicit type conversions, it may surprise you to hear that there’s at least one situation where unscoped enums may be useful. That’s when referring to fields within C++11’s std::tuples. For example, suppose we have a tuple holding values for the name, email address, and reputation value for a user at a social networking website:

using UserInfo = // type alias; see Item 9

std::tuple<std::string, // name

std::string, // email

std::size\_t> ; // reputation*”*

*p. 90*

*example continue: “*Though the comments indicate what each field of the tuple represents, that’s probably not very helpful when you encounter code like this in a separate source file:

UserInfo uInfo; // object of tuple type

…

auto val = std::get<1>(uInfo); // get value of field 1*” example of auto using. std::get purpose.*

*Problem of cognitive orientation in code: “*As a programmer, you have a lot of stuff to keep track of. Should you really be expected to remember that field 1 corresponds to the user’s email address? I not.*” Programmer work.*

*Example continue: “*As a programmer, you have a lot of stuff to keep track of. Should you really be expected to remember that field 1 corresponds to the user’s email address? I think not. Using an unscoped enum to associate names with field numbers avoids the need to:

enum UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo; // as before

…

auto val = std::get<uiEmail>(uInfo); // ah, get value of

// email field*” advantage of unscoped enum.*

*Example continue: “*The corresponding code with scoped enums is substantially more verbose:

enum class UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo; // as before

…

auto val =

std::get<static\_cast<std::size\_t>(UserInfoFields::uiEmail)>

(uInfo);

The verbosity can be reduced by writing a function that takes an enumerator and returns its corresponding std::size\_t value, but it’s a bit tricky. std::get is a template, and the value you provide is a template argument (notice the use of angle brackets, not parentheses), so the function that transforms an enumerator into a std::size\_t has to produce its result during compilation.*” Compilation structure. Code which must be defined during compilation – compilation time. C++ structure and code classification.*

*“*As Item 15 explains, that means it must be a constexpr function.*” what is constexpr? Book structure (Item 15).*

*p. 91*

*Context of using type traits (intbu) (Item 9) and noexcept (Item 14) “*In fact, it should really be a constexpr function template, because it should work with any kind of enum. And if we’re going to make that generalization, we should generalize the return type, too. Rather than returning std::size\_t, we’ll return the enum’s underlying type. It’s available via the std::underlying\_type type trait. (See Item 9 for information on type traits.) Finally, we’ll declare it noexcept (see Item 14), because we know it will never yield an exception.*”*

*Intbu the following code: “*The result is a function template toUType that takes an arbitrary enumerator and can return its value as a compiletime constant:

template<typename E>

constexpr typename std::underlying\_type<E>::type

toUType(E enumerator) noexcept

{

return

static\_cast<typename

std::underlying\_type<E>::type>(enumerator);

}

In C++14, toUType can be simplified by replacing typename std::underly ing\_type<E>::type with the sleeker std::underlying\_type\_t (see Item 9):

template<typename E> // C++14

constexpr std::underlying\_type\_t<E>

toUType(E enumerator) noexcept

{

return static\_cast<std::underlying\_type\_t<E>>(enumerator);

}

The even-sleeker auto return type (see Item 3) is also valid in C++14:

template<typename E> // C++14

constexpr auto

toUType(E enumerator) noexcept

{

return static\_cast<std::underlying\_type\_t<E>>(enumerator);

}

Regardless of how it’s written, toUType permits us to access a field of the tuple like this:

auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);

*” Auto example and alias template example. Method to know type of tuple enumerator.*

*Some balance of programming: number of code – number of dangerous: “*It’s still more to write than use of the unscoped enum, but it also avoids namespace pollution and inadvertent conversions involving enumerators. In many cases, you may decide that typing a few extra characters is a reasonable price to pay for the ability to avoid the pitfalls of an enum technology that dates to a time when the state of the art in digital telecommunications was the 2400-baud modem.*”*

*p. 92*

*Book structure. Resume.*

*“*Things to Remember • C++98-style enums are now known as unscoped enums.

• Enumerators of scoped enums are visible only within the enum. They convert to other types only with a cast.

• Both scoped and unscoped enums support specification of the underlying type. The default underlying type for scoped enums is int. Unscoped enums have no default underlying type.

• Scoped enums may always be forward-declared. Unscoped enums may be forward-declared only if their declaration specifies an underlying type.*” Structure of scoped and unscoped enums.*

**Item 11: Prefer deleted functions to private undefined ones**

*Good code practice and programming situation concerning functions: “*If you’re providing code to other developers, and you want to prevent them from calling a particular function, you generally just don’t declare the function. No function declaration, no function to call. Easy, peasy.*”*

*C++ structure: “*But sometimes C++ declares functions for you, and if you want to prevent clients from calling those functions, the peasy isn’t quite so easy any more.*” What does it mean? It means C++ feature “special member functions”: “*The situation arises only for the “special member functions,” i.e., the member functions that C++ automatically generates when they’re needed.*” Book structure (Item 17).*

*Book structure and these functions examples: “*Item 17 discusses these functions in detail, but for now, we’ll worry only about the copy constructor and the copy assignment operator.*” Copy constructor and assignment operator are special member functions?*

*Book structure ans C++ history: “*This chapter is largely devoted to common practices in C++98 that have been superseded by better practices in C++11, and in C++98, if you want to suppress use of a member function, it’s almost always the copy constructor, the assignment operator, or both.*”*

*Book category: “practice”*

*Way to prevent using of function: “*The C++98 approach to preventing use of these functions is to declare them private and not define them. For example, near the base of the iostreams hierarchy in the C++ Standard Library is the class template basic\_ios. All istream and ostream classes inherit (possibly indirectly) from this class. Copying istreams and ostreams is undesirable, because it’s not really clear what such operations should do. An istream object, for example, represents a stream of input values, some of which may have already been read, and some of which will potentially be read later. If an istream were to be copied, would that entail copying all the values that had already been read as well as all the values that would be read in the future? The easiest way to deal with such questions is to define them out of existence. Prohibiting the copying of streams does just that.*” what does it mean “not define function”??*

*p. 93*

*example of this method: “*To render istream and ostream classes uncopyable, basic\_ios is specified in C++98 as follows (including the comments):

template <class charT, class traits = char\_traits<charT> >

class basic\_ios : public ios\_base {

public:

…

private:

basic\_ios(const basic\_ios& ); // not defined

basic\_ios& operator=(const basic\_ios&); // not defined

};*”*

*Intbu the sense of this text: “*Declaring these functions private prevents clients from calling them. Deliberately failing to define them means that if code that still has access to them (i.e., member functions or friends of the class) uses them, linking will fail due to missing function definitions.*” – каб ніякі код не мог да іх звярнуцца? C++ structure: for each program object there is code that has access to it – this is scope. For explanation of this concept.*

*Deleted function structure and C++ history: “*In C++11, there’s a better way to achieve essentially the same end: use “= delete” to mark the copy constructor and the copy assignment operator as deleted functions. Here’s the same part of basic\_ios as it’s specified in C++11:

template <class charT, class traits = char\_traits<charT> >

class basic\_ios : public ios\_base {

public:

…

basic\_ios(const basic\_ios& ) = delete;

basic\_ios& operator=(const basic\_ios&) = delete;

…

};*” The second way to make function invisible to client.*

*Advantage of second method over first: aspects of compilation and link time (intbu what is compilation and what is link in low level): “*The difference between deleting these functions and declaring them private may seem more a matter of fashion than anything else, but there’s greater substance here than you might think. Deleted functions may not be used in any way, so even code that’s in member and friend functions will fail to compile if it tries to copy basic\_ios objects. That’s an improvement over the C++98 behavior, where such improper usage wouldn’t be diagnosed until link-time.*” This question concerns friend functions and classes.*

*C++ convention: “*By convention, deleted functions are declared public, not private. There’s a reason for that. When client code tries to use a member function, C++ checks accessibility before deleted status. When client code tries to use a deleted private function, some compilers complain only about the function being private, even though the function’s accessibility doesn’t really affect whether it can be used. It’s worth bearing this in mind when revising legacy code to replace private-and-not-defined member functions with deleted ones, because making the new functions public will generally result in better error messages.*” Better error messages of compiler.*

*p. 94*

*advantage of deleted functions and their using context: “*An important advantage of deleted functions is that any function may be deleted, while only member functions may be private.*” This is concerns functions and member functions. С herigage of C++. C++ aspect of type conversion.*

*Example of prevention of ordinary functions calls: “*C++’s C heritage means that pretty much any type that can be viewed as vaguely numerical will implicitly convert to int, but some calls that would compile might not make sense:

if (isLucky('a')) … // is 'a' a lucky number?

if (isLucky(true)) … // is "true"?

if (isLucky(3.5)) … // should we truncate to 3

// before checking for luckiness?

If lucky numbers must really be integers, we’d like to prevent calls such as these from compiling. One way to accomplish that is to create deleted overloads for the types we want to filter out:

bool isLucky(int number); // original function

bool isLucky(char) = delete; // reject chars

bool isLucky(bool) = delete; // reject bools

bool isLucky(double) = delete; // reject doubles and

// floats*“ C++ practice of preventing to use some arguments as function arguments.*

*С++ type conversion peculiarity: “*(The comment on the double overload that says that both doubles and floats will be rejected may surprise you, but your surprise will dissipate once you recall that, given a choice between converting a float to an int or to a double, C++ prefers the conversion to double. Calling isLucky with a float will therefore call the double overload, not the int one. Well, it’ll try to. The fact that that overload is deleted will prevent the call from compiling.)

*” double, float and int.*

*Example concerns and deleted functions purpose: “*Although deleted functions can’t be used, they are part of your program. As such, they are taken into account during overload resolution. That’s why, with the deleted

function declarations above, the undesirable calls to isLucky will be rejected:

if (isLucky('a')) … // error! call to deleted function

if (isLucky(true)) … // error!

if (isLucky(3.5f)) … // error!*”*

*p. 95*

*Another trick…*

*15.12.2016*

*Deleted functions using context: “*Another trick that deleted functions can perform (and that private member functions can’t) is to prevent use of template instantiations that should be disabled. For example, suppose you need a template that works with built-in pointers (Chapter 4’s advice to prefer smart pointers to raw pointers notwithstanding):*”. Book structure (Chapter 4). Advantage of smart pointers. Book recommendation.*

*Two special cases of pointers: “*template<typename T> void processPointer(T\* ptr);

There are two special cases in the world of pointers. One is void\* pointers, because there is no way to dereference them, to increment or decrement them, etc. The other is char\* pointers, because they typically represent pointers to C-style strings, not pointers to individual characters.*” Intbu especially void\*. C++ concept – C-style strings.*

*Example of this deleted functions application: “*These special cases often call for special handling, and, in the case of the processPointer template, let’s assume the proper handling is to reject calls using those types. That is, it should not be possible to call processPointer with void\* or char\* pointers. That’s easily enforced. Just delete those instantiations:

template<>

void processPointer<void>(void\*) = delete;

template<>

void processPointer<char>(char\*) = delete;

Now, if calling processPointer with a void\* or a char\* is invalid, it’s probably also invalid to call it with a const void\* or a const char\*, so those instantiations will typically need to be deleted, too:

template<>

void processPointer<const void>(const void\*) = delete;

template<>

void processPointer<const char>(const char\*) = delete;

And if you really want to be thorough, you’ll also delete the const volatile void\* and const volatile char\* overloads, and then you’ll get to work on the overloads for pointers to the other standard character types: std::wchar\_t, std::char16\_t, and std::char32\_t.*” This is enough labour intensive process.*

*C++ feature: access level of class memebers and templates: “*Interestingly, if you have a function template inside a class, and you’d like to disable some instantiations by declaring them private (à la classic C++98 convention), you can’t, because it’s not possible to give a member function template specialization a*”*

*p. 96*

*Example of it: “*If processPointer were a member function template inside Widget, for example, and you wanted to disable calls for void\* pointers, this would be the C++98 approach, though it would not compile:

class Widget {

public:

…

template<typename T>

void processPointer(T\* ptr)

{ … }

private:

template<> // error!

void processPointer<void>(void\*);

};

The problem is that template specializations must be written at namespace scope, not class scope. This issue doesn’t arise for deleted functions, because they don’t need a different access level. They can be deleted outside the class (hence at namespace scope):

class Widget {

public:

…

template<typename T>

void processPointer(T\* ptr)

{ … }

…

};

template<> // still

void Widget::processPointer<void>(void\*) = delete; // public,

// but

// deleted

The truth is that the C++98 practice of declaring functions private and not defining them was really an attempt to achieve what C++11’s deleted functions actually accomplish. As an emulation, the C++98 approach is not as good as the real thing. It doesn’t work outside classes, it doesn’t always work inside classes, and when it does work, it may not work until link-time. So stick to deleted functions.*” Deleted functions structure and advantage. Book recommendation.*

*Book structure. Resume: “*Things to Remember • Prefer deleted functions to private undefined ones.

• Any function may be deleted, including non-member functions and template instantiations.*”*

**Item 12: Declare overriding functions override**

*OOP in C++. Three aspects: “*The world of object-oriented programming in C++ revolves around classes, inheritance, and virtual functions. Among the most fundamental ideas in this world is that virtual function implementations in derived classes override the implementations of their base class counterparts. It’s disheartening, then, to realize just how easily virtual function overriding can go wrong.*” Dangerous of virtual functions overriding.*

*Book example: “*It’s disheartening, then, to realize just how easily virtual function overriding can go wrong. It’s almost as if this part of the language were designed with the idea that Murphy’s Law wasn’t just to be obeyed, it was to be honored. Because “overriding” sounds a lot like “overloading,” yet is completely unrelated, let me make clear that virtual function overriding is what makes it possible to invoke a derived class function through a base class interface:

class Base {

public:

virtual void doWork(); // base class virtual function

…

};

class Derived: public Base {

public:

virtual void doWork(); // overrides Base::doWork

… // ("virtual" is optional

}; // here)

std::unique\_ptr<Base> upb = // create base class pointer

std::make\_unique<Derived>(); // to derived class object;

// see Item 21 for info on

… // std::make\_unique

upb->doWork(); // call doWork through base

// class ptr; derived class

// function is invoked*” intbu sense of this example. Book structure (Item 21). What is std::make\_unique?*

*p. 98*

*“*For overriding to occur, several requirements must be met:

The base class function must be virtual.

• The base and derived function names must be identical (except in the case of destructors).

• The parameter types of the base and derived functions must be identical.

• The constness of the base and derived functions must be identical.

• The return types and exception specifications of the base and derived functions must be compatible.

*” What do concepts “identical” and “compatible” mean? Function overriding conditions.*

*C++ history and additional condition: “*To these constraints, which were also part of C++98, C++11 adds one more:

• The functions’ reference qualifiers must be identical. Member function reference qualifiers are one of C++11’s less-publicized features, so don’t be surprised if you’ve never heard of them. They make it possible to limit use of a member function to lvalues only or to rvalues only. Member functions need not be virtual to use them:

class Widget {

public:

…

void doWork() &; // this version of doWork applies

// only when \*this is an lvalue

void doWork() &&; // this version of doWork applies

}; // only when \*this is an rvalue

…

Widget makeWidget(); // factory function (returns rvalue)

Widget w; // normal object (an lvalue)

…

w.doWork(); // calls Widget::doWork for lvalues

// (i.e., Widget::doWork &)

makeWidget().doWork(); // calls Widget::doWork for rvalues

// (i.e., Widget::doWork &&)*” feature of class concerning function application and is connected with rvalues and lvalues. C++ feature and concept: reference qualifiers. What are using contexts of them? Example of rvalue and lvalue.*

*Function overriding condition: “*I’ll say more about member functions with reference qualifiers later, but for now, simply note that if a virtual function in a base class has a reference qualifier, derived class overrides of that function must have exactly the same reference qualifier. If they don’t, the declared functions will still exist in the derived class, but they won’t override anything in the base class.*”*

*p. 99*

*C++ dangerous errors: overriding errors: “*All these requirements for overriding mean that small mistakes can make a big difference. Code containing overriding errors is typically valid, but its meaning isn’t what you intended. You therefore can’t rely on compilers notifying you if you do something wrong.*”*

*Example of such error: “*For example, the following code is completely legal and, at first sight, looks reasonable, but it contains no virtual function overrides—not a single derived class function that is tied to a base class function. Can you identify the problem in each case, i.e., why each derived class function doesn’t override the base class function with the same name?

class Base {

public:

virtual void mf1() const;

virtual void mf2(int x);

virtual void mf3() &;

void mf4() const;

};

class Derived: public Base {

public:

virtual void mf1();

virtual void mf2(unsigned int x);

virtual void mf3() &&;

void mf4() const;

};

Need some help?

• mf1 is declared const in Base, but not in Derived.

• mf2 takes an int in Base, but an unsigned int in Derived.

• mf3 is lvalue-qualified in Base, but rvalue-qualified in Derived.

• mf4 isn’t declared virtual in Base.

*” Fortunately I solved this problem. It’s good.*

*C++ compiler details: “*You may think, “Hey, in practice, these things will elicit compiler warnings, so I don’t need to worry.” Maybe that’s true. But maybe it’s not. With two of the compilers I checked, the code was accepted without complaint, and that was with all warnings enabled. (Other compilers provided warnings about some of the issues, but not all of them.)*”*

*p. 100*

*Purpose of override concept. Kind of error and problem. Example of concept: “*Because declaring derived class overrides is important to get right, but easy to get wrong, C++11 gives you a way to make explicit that a derived class function is supposed to override a base class version: declare it override. Applying this to the example above would yield this derived class:

class Derived: public Base {

public:

virtual void mf1() override;

virtual void mf2(unsigned int x) override;

virtual void mf3() && override;

virtual void mf4() const override;

};

This won’t compile, of course, because when written this way, compilers will kvetch about all the overriding-related problems. That’s exactly what you want, and it’s why you should declare all your overriding functions override. The code using override that does compile looks as follows (assuming that the goal is for all functions in Derived to override virtuals in Base):

class Base {

public:

virtual void mf1() const;

virtual void mf2(int x);

virtual void mf3() &;

virtual void mf4() const;

};

class Derived: public Base {

public:

virtual void mf1() const override;

virtual void mf2(int x) override;

virtual void mf3() & override;

void mf4() const override; // adding "virtual" is OK,

}; // but not necessary

Note that in this example, part of getting things to work involves declaring mf4 virtual in Base.*”*

*Overriding problems appearing context structure: “*Most overriding-related errors occur in derived classes, but it’s possible for things to be incorrect in base classes, too.*” Book recommendation. Good C++ practice.*

*Аспекты маёй сістэмы: я магу запытацца: добрая практыка выкарыстання ўказацеляў – і г.д. Метам структуры будуць служыць падказкамі для выкарыстання сісвед. А таксама сістэма павінна ўказваць, што неабходна прачытаць для больш добрага засваення. Яна не замяняе кнігу. Але дапамагае лепей арыентавацца ў інфармацыі.*

A policy of using override…

*16.12.2016*

*2 advantages of override: “*A policy of using override on all your derived class overrides can do more than just enable compilers to tell you when would-be overrides aren’t overriding anything. It can also help you gauge the ramifications if you’re contemplating changing the signature of a virtual function in a base class. If derived classes use override everywhere, you can just change the signature, recompile your system, see how much damage you’ve caused (i.e., how many derived classes fail to compile), then decide whether the signature change is worth the trouble.*” Detection of potential mistake is very important part of system. Think about this effect.*

*p. 101*

*C++ feature “contextual keywords”. They are final and override: “*C++ has always had keywords, but C++11 introduces two contextual keywords, over ride and final… These keywords have the characteristic that they are reserved, but only in certain contexts.*” C++ history.*

*Override using context: “*In the case of override, it has a reserved meaning only when it occurs at the end of a member function declaration. That means that if you have legacy code that already uses the name override, you don’t need to change it for C++11:

class Warning { // potential legacy class from C++98

public:

…

void override(); // legal in both C++98 and C++11

… // (with the same meaning)

};

That’s all there is to say about override, but it’s not all there is to say about member function reference qualifiers. I promised I’d provide more information on them later, and now it’s later.*” Book structure. Connection without indication of accurate location. Contextual keywords can be used as function names.*

*Lvalue and rvalue structure: “*If we want to write a function that accepts only lvalue arguments, we declare a nonconst lvalue reference parameter:

void doSomething(Widget& w); // accepts only lvalue Widgets

If we want to write a function that accepts only rvalue arguments, we declare an rvalue reference parameter:

void doSomething(Widget&& w); // accepts only rvalue Widgets*”*

*Member function reference qualifiers structure: “*Member function reference qualifiers simply make it possible to draw the same distinction for the object on which a member function is invoked, i.e. \*this. It’s precisely analogous to the const at the end of a member function declaration, which indicates that the object on which the member function is invoked (i.e., \*this) is const.*” “const qualifier in member functions” structure.*

*Final qualifier structure: “*Applying final to a virtual function prevents the function from being overridden in derived classes. Final may also be applied to a class, in which case the class is prohibited from being used as a base class.*”*

*Using context of reference qualifiers: “*The need for reference-qualified member functions is not common, but it can arise. For example, suppose our Widget class has a std::vector data member, and we offer an accessor function that gives clients direct access to it:

class Widget {

public:

using DataType = std::vector<double>; // see Item 9 for

… // info on "using"

DataType& data() { return values; }

…

private:

DataType values;

};

*” book structure (item 9).*

*Аспекты маёй сістэмы: у кодзе я магу заўважаць кантструкты і выдаваць чалавеку ўсю інфармацыю аб іх. Пасіўна. Патокава. Ён можа глянуць. І, магчыма, нешта будзе палезным. Яму не трэба будзе гэта шукаць спецыяльна. На гэта траціш шмат часу.*

*p. 102*

*example continue and important programming concept “client code”: “*This is hardly the most encapsulated design that’s seen the light of day, but set that aside and consider what happens in this client code:

Widget w;

…

auto vals1 = w.data(); // copy w.values into vals1

The return type of Widget::data is an lvalue reference (a std::vector<double>&, to be precise), and because lvalue references are defined to be lvalues, we’re initializing vals1 from an lvalue. vals1 is thus copy constructed from w.values, just as the comment says. Now suppose we have a factory function that creates Widgets, Widget makeWidget();

and we want to initialize a variable with the std::vector inside the Widget returned from makeWidget:

auto vals2 = makeWidget().data(); // copy values inside the

// Widget into vals2

Again, Widgets::data returns an lvalue reference, and, again, the lvalue reference is an lvalue, so, again, our new object (vals2) is copy constructed from values inside the Widget. This time, though, the Widget is the temporary object returned from makeWidget (i.e., an rvalue), so copying the std::vector inside it is a waste of time. It’d be preferable to move it, but, because data is returning an lvalue reference, the rules of C++ require that compilers generate code for a copy. (There’s some wiggle room for optimization through what is known as the “as if rule,” but you’d be foolish to rely on your compilers finding a way to take advantage of it.)

*” example of lvalue and rvalue (temporary object). Problem for solution of which lvalues and rvalues are introduced “copying of temporary object data”. С++ compiler structure: internal optimization is not reliable.*

*Purpose of reference qualifiers: “*What’s needed is a way to specify that when data is invoked on an rvalue Widget, the result should also be an rvalue. Using reference qualifiers to overload data for lvalue and rvalue Widgets makes that possible:*”*

*p. 103*

*example of it: “*class Widget {

public:

using DataType = std::vector<double>;

…

DataType& data() & // for lvalue Widgets,

{ return values; } // return lvalue

DataType data() && // for rvalue Widgets,

{ return std::move(values); } // return rvalue

…

private:

DataType values;

};

Notice the differing return types from the data overloads. The lvalue reference overload returns an lvalue reference (i.e., an lvalue), and the rvalue reference overload returns a temporary object (i.e., an rvalue). This means that client code now behaves as we’d like:

auto vals1 = w.data(); // calls lvalue overload for

// Widget::data, copy

// constructs vals1

auto vals2 = makeWidget().data(); // calls rvalue overload for

// Widget::data, move

// constructs vals2*” using context of reference qualifiers. For better understanding of rvalues and lvalues. С++ structure.*

*Book recommendation and good C++ practice: “*This is certainly nice, but don’t let the warm glow of this happy ending distract you from the true point of this Item. That point is that whenever you declare a function in a derived class that’s meant to override a virtual function in a base class, be sure to declare that function override.*”*

*Book structure. Resume: “*Things to Remember • Declare overriding functions override.

• Member function reference qualifiers make it possible to treat lvalue and rvalue objects (\*this) differently.*”*

*p. 104*

**Item 13: Prefer const\_iterators to iterators**

*Const\_iterators structure: “*const\_iterators are the STL equivalent of pointers-to-const. They point to values that may not be modified. The standard practice of using const whenever possible dictates that you should use const\_iterators any time you need an iterator, yet have no need to modify what the iterator points to.

*” C++ practice: use const whenever possible. It is necessary to better unserstand this practice.*

*C++ history: “*That’s as true for C++98 as for C++11, but in C++98, const\_iterators had only halfhearted support. It wasn’t that easy to create them, and once you had one, the ways you could use it were limited. For example, suppose you want to search a std::vector<int> for the first occurrence of 1983 (the year “C++” replaced “C with Classes” as the name of the programming language), then insert the value 1998 (the year the first ISO C++ Standard was adopted) at that location. If there’s no 1983 in the vector, the insertion should go at the end of the vector. Using iterators in C++98, that was easy:

std::vector<int> values;

…

std::vector<int>::iterator it =

std::find(values.begin(),values.end(), 1983);

values.insert(it, 1998);

But iterators aren’t really the proper choice here, because this code never modifies what an iterator points to. Revising the code to use const\_iterators should be trivial, but in C++98, it was anything but. Here’s one approach that’s conceptually sound, though still not correct:

typedef std::vector<int>::iterator IterT; // typetypedef

std::vector<int>::const\_iterator ConstIterT; // defs

std::vector<int> values;

…

ConstIterT ci =

std::find(static\_cast<ConstIterT>(values.begin()), // cast

static\_cast<ConstIterT>(values.end()), // cast

1983);

values.insert(static\_cast<IterT>(ci), 1998); // may not

// compile; see

// below*”*

*p. 105*

*example continue: “*The typedefs aren’t required, of course, but they make the casts in the code easier to write. (If you’re wondering why I’m showing typedefs instead of following the advice of Item 9 to use alias declarations, it’s because this example shows C++98 code, and alias declarations are a feature new to C++11.)

The casts in the call to std::find are present because values is a non-const container and in C++98, there was no simple way to get a const\_iterator from a nonconst container. The casts aren’t strictly necessary, because it was possible to get const\_iterators in other ways (e.g., you could bind values to a reference-to-const variable, then use that variable in place of values in your code), but one way or another, the process of getting const\_iterators to elements of a non-const container involved some amount of contorting. Once you had the const\_iterators, matters often got worse, because in C++98, locations for insertions (and erasures) could be specified only by iterators. const\_iterators weren’t acceptable. That’s why, in the code above, I cast the const\_iterator (that I was so careful to get from std::find) into an iterator:

passing a const\_iterator to insert wouldn’t compile.*” C++ history.*

*C++ structure. Problem of type conversion between const\_iterator and iterator: “*To be honest, the code I’ve shown might not compile, either, because there’s no portable conversion from a const\_iterator to an iterator, not even with a static\_cast. Even the semantic sledgehammer known as reinterpret\_cast can’t do the job. (That’s not a C++98 restriction. It’s true in C++11, too. const\_iterators simply don’t convert to iterators, no matter how much it might seem like they should.)*” Aspects of reinterpret\_cast.*

*C++ history: “*There are some portable ways to generate iterators that point where const\_iterators do, but they’re not obvious, not universally applicable, and not worth discussing in this book. Besides, I hope that by now my point is clear: const\_iterators were so much trouble in C++98, they were rarely worth the bother. At the end of the day, developers don’t use const whenever possible, they use it whenever practical, and in C++98, const\_iterators just weren’t very practical.*”*

*C++ history and STL structure: “*All that changed in C++11. Now const\_iterators are both easy to get and easy to use. The container member functions cbegin and cend produce const\_iterators, even for non-const containers, and STL member functions that use iterators to identify positions (e.g., insert and erase) actually use const\_iterators. Revising the original C++98 code that uses iterators to use const\_iterators in C++11 is truly trivial:

std::vector<int> values; // as before

…

auto it = // use cbegin

std::find(values.cbegin(),values.cend(), 1983); // and cend

values.insert(it, 1998);

Now that’s code using const\_iterators that’s practical!*”*

*p. 106*

*C++ good practice and book recommendation: “*Now that’s code using const\_iterators that’s practical!*”*

*what is maximally generic code? Why does it uses non-member functions rather that assuming the existence of member versions?: “*About the only situation in which C++11’s support for const\_iterators comes up a bit short is when you want to write maximally generic library code. Such code takes into account that some containers and container-like data structures offer begin and end (plus cbegin, cend, rbegin, etc.) as non-member functions, rather than members. This is the case for built-in arrays, for example, and it’s also the case for some third-party libraries with interfaces consisting only of free functions. Maximally generic code thus uses non-member functions rather than assuming the existence of member versions.*”*

*Example of it: “*For example, we could generalize the code we’ve been working with into a findAnd Insert template as follows:

template<typename C, typename V>

void findAndInsert(C& container, // in container, find

const V& targetVal, // first occurrence

const V& insertVal) // of targetVal, then

{ // insert insertVal

using std::cbegin; // there

using std::cend;

auto it = std::find(cbegin(container), // non-member cbegin

cend(container), // non-member cend

targetVal);

container.insert(it, insertVal);

}*” Why this code is more generic?*

*C++ history. C++11 const\_iterators limitations: “*This works fine in C++14, but, sadly, not in C++11. Through an oversight during standardization, C++11 added the non-member functions begin and end, but it failed to add cbegin, cend, rbegin, rend, crbegin, and crend. C++14 rectifies that oversight.*” difference C++1 and C++14.*

*I understand! Code is not generic, because this method (for example cbegin) must be in the given class and it must be declared there. Non-member function requires less work and more understanding.*

*Example of generic code in C++11: “*If you’re using C++11, you want to write maximally generic code, and none of the libraries you’re using provides the missing templates for non-member cbegin and friends, you can throw your own implementations together with ease. For example, here’s an implementation of non-member cbegin:

template <class C>

auto cbegin(const C& container)->decltype(std::begin(container))

{

return std::begin(container); // see explanation below

}*”*

*p. 107*

*example continue: “*You’re surprised to see that non-member cbegin doesn’t call member cbegin, aren’t you? So was I. But follow the logic. This cbegin template accepts any type of argument representing a container-like data structure, C, and it accesses this argument through its reference-to-const parameter, container. If C is a conventional container type (e.g., a std::vector<int>), container will be a reference to a const version of that container (e.g., a const std::vector<int>&). Invoking the nonmember begin function (provided by C++11) on a const container yields a const\_iterator, and that iterator is what this template returns. The advantage of implementing things this way is that it works even for containers that offer a begin member function (which, for containers, is what C++11’s non-member begin calls), but fail to offer a cbegin member. You can thus use this non-member cbegin on containers that directly support only begin. This template also works if C is a built-in array type. In that case, container becomes a reference to a const array. C++11 provides a specialized version of non-member begin for arrays that returns a pointer to the array’s first element. The elements of a const array are const, so the pointer that non-member begin returns for a const array is a pointer-to-const, and a pointer-to-const is, in fact, a const\_iterator for an array. (For insight into how a template can be specialized for built-in arrays, consult Item 1’s discussion of type deduction in templates that take reference parameters to arrays.)*” Book structure (Item 1). To remind problems of build-in arrays and templates. Kind of arrays – build-in.*

*C++ good practice and book recommendation: “*But back to basics. The point of this Item is to encourage you to use const\_itera tors whenever you can. The fundamental motivation—using const whenever it’s meaningful—predates C++11, but in C++98, it simply wasn’t practical when working with iterators. In C++11, it’s eminently practical, and C++14 tidies up the few bits of unfinished business that C++11 left behind.*” C++ history.*

*Book structure. Resume: “*Things to Remember

• Prefer const\_iterators to iterators.

• In maximally generic code, prefer non-member versions of begin, end, rbegin, etc., over their member function counterparts.*” It is better to work with const things because in this case compiler can detect error of changing such value. This is way to detect potenthial characteristic for people thinking errors.*

*p. 108*

**Item 14: Declare functions noexcept if they won’t emit exceptions**

*C++ history and problems with exception treatment: “*In C++98, exception specifications were rather temperamental beasts. You had to summarize the exception types a function might emit, so if the function’s implementation was modified, the exception specification might require revision, too. Changing an exception specification could break client code, because callers might be dependent on the original exception specification. Compilers typically offered no help in maintaining consistency among function implementations, exception specifications, and client code. Most programmers ultimately decided that C++98 exception specifications weren’t worth the trouble.*” Changing of exceptions can break client code. I don’t like to use them.*

*C++ idiom and noexcept purpose: “*During work on C++11, a consensus emerged that the truly meaningful information about a function’s exception-emitting behavior was whether it had any. Black or white, either a function might emit an exception or it guaranteed that it wouldn’t. This maybe-or-never dichotomy forms the basis of C++11’s exception specifications, which essentially replace C++98’s. (C++98-style exception specifications remain valid, but they’re deprecated.) In C++11, unconditional noexcept is for functions that guarantee they won’t emit exceptions.*” C++ history. Exception feature property “maybe-or-never dichotomy”. Noexcept purpose.*

*C++ good practice. Aspects of function description for clients: “*Whether a function should be so declared is a matter of interface design. The exception-emitting behavior of a function is of key interest to clients. Callers can query a function’s noexcept status, and the results of such a query can affect the exception safety or efficiency of the calling code. As such, whether a function is noexcept is as important a piece of information as whether a member function is const. Failure to declare a function noexcept when you know that it won’t emit an exception is simply poor interface specification.*”. But we can to not notice something. So in future it is possible for noexcept to become wrong.*

*Noexcept advantage: “*But there’s an additional incentive to apply noexcept to functions that won’t produce exceptions: it permits compilers to generate better object code. To understand why, it helps to examine the difference between the C++98 and C++11 ways of saying that a function won’t emit exceptions. Consider a function f that promises callers they’ll never receive an exception. The two ways of expressing that are:

int f(int x) throw(); // no exceptions from f: C++98 style

int f(int x) noexcept; // no exceptions from f: C++11 style

If, at runtime, an exception leaves f, f’s exception specification is violated. With the C++98 exception specification, the call stack is unwound to f’s caller, and, after some actions not relevant here, program execution is terminated. With the C++11 exception specification, runtime behavior is slightly different: the stack is only possibly unwound before program execution is terminated.*” I didn’t understand this example. Intbu. What is “stack is unwound”?*

*p. 109*

*example continue and compiler structure: “*The difference between unwinding the call stack and possibly unwinding it has a surprisingly large impact on code generation. In a noexcept function, optimizers need not keep the runtime stack in an unwindable state if an exception would propagate out of the function, nor must they ensure that objects in a noexcept function are destroyed in the inverse order of construction should an exception leave the function. Functions with “throw()” exception specifications lack such optimization flexibility, as do functions with no exception specification at all. The situation can be summarized this way:

RetType function(params) noexcept; // most optimizable

RetType function(params) throw(); // less optimizable

RetType function(params); // less optimizable

*” this is very misunderstood text. Intbu. Aspects of stack work.*

*Book recommendation: “*This alone is sufficient reason to declare functions noexcept whenever you know they won’t produce exceptions.*”*

*Another example with move: “*This alone is sufficient reason to declare functions noexcept whenever you know they won’t produce exceptions. Assume this code works fine, and you have no interest in modifying it for C++11. However, you do want to take advantage of the fact that C++11’s move semantics can improve the performance of legacy code when move-enabled types are involved. You therefore ensure that Widget has move operations, either by writing them yourself or by seeing to it that the conditions for their automatic generation are fulfilled (see Item 17). When a new element is added to a std::vector, it’s possible that the std::vector lacks space for it, i.e., that the std::vector’s size is equal to its capacity. When that happens, the std::vector allocates a new, larger, chunk of memory to hold its*” C++ feature “move semantic” example. Move semantic realization. Book structure (Item 17). Std::vector realization.*

*p. 110*

*example continue: “*elements, and it transfers the elements from the existing chunk of memory to the new one. In C++98, the transfer was accomplished by copying each element from the old memory to the new memory, then destroying the objects in the old memory. This approach enabled push\_back to offer the strong exception safety guarantee: if an exception was thrown during the copying of the elements, the state of the std::vector remained unchanged, because none of the elements in the old memory were destroyed until all elements had been successfully copied into the new memory. In C++11, a natural optimization would be to replace the copying of std::vector elements with moves. Unfortunately, doing this runs the risk of violating push\_back’s exception safety guarantee. If n elements have been moved from the old memory and an exception is thrown moving element n+1, the push\_back operation can’t run to completion. But the original std::vector has been modified: n of its elements have been moved from. Restoring their original state may not be possible, because attempting to move each object back into the original memory may itself yield an exception. This is a serious problem, because the behavior of legacy code could depend on push\_back’s strong exception safety guarantee. Therefore, C++11 implementations can’t silently replace copy operations inside push\_back with moves unless it’s known that the move operations won’t emit exceptions. In that case, having moves replace copies would be safe, and the only side effect would be improved performance. std::vector::push\_back takes advantage of this “move if you can, but copy if you must” strategy, and it’s not the only function in the Standard Library that does. Other functions sporting the strong exception safety guarantee in C++98 (e.g., std::vector::reserve, std::deque::insert, etc.) behave the same way. All these functions replace calls to copy operations in C++98 with calls to move operations in C++11 only if the move operations are known to not emit exceptions. But how can a function know if a move operation won’t produce an exception? The answer is obvious: it checks to see if the operation is declared noexcept.3*” std::vector structure and safety. Move semantics can reduce safety. This is its pitfall? Noexception optimization connecting with move semantics. C++ strategy “*move if you can, but copy if you must*”. Noexcept using context.*

*Another noexcept using context motivation: “*swap functions comprise another case where noexcept is particularly desirable. swap is a key component of many STL algorithm implementations, and it’s commonly employed in copy assignment operators, too. Its widespread use renders the optimizations that noexcept affords especially worthwhile. Interestingly, whether swaps in the Standard Library are noexcept is sometimes dependent on whether user*” what is swap functions? STL structure.*

*C++ realization: “*The checking is typically rather roundabout. Functions like std::vector::push\_back call std::move\_if\_noexcept, a variation of std::move that conditionally casts to an rvalue (see Item 23), depending on whether the type’s move constructor is noexcept. In turn, std::move\_if\_noexcept consults std::is\_nothrow\_move\_constructible, and the value of this type trait (see Item 9) is set by compilers, based on whether the move constructor has a noexcept (or throw()) designation.*” intbu. Book structure. (Item 23, Item 9).*

*p. 111*

*example continue: “*defined swaps are noexcept. For example, the declarations for the Standard Library’s swaps for arrays and std::pair are:

template <class T, size\_t N>

void swap(T (&a)[N], // see

T (&b)[N]) noexcept(noexcept(swap(\*a, \*b))); // below

template <class T1, class T2>

struct pair {

…

void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&

noexcept(swap(second, p.second)));

…

};

These functions are conditionally noexcept: whether they are noexcept depends on whether the expressions inside the noexcept clauses are noexcept. Given two arrays of Widget, for example, swapping them is noexcept only if swapping individual elements in the arrays is noexcept, i.e., if swap for Widget is noexcept. The author of Widget’s swap thus determines whether swapping arrays of Widget is noexcept. That, in turn, determines whether other swaps, such as the one for arrays of arrays of Widget, are noexcept. Similarly, whether swapping two std::pair objects containing Widgets is noexcept depends on whether swap for Widgets is noexcept. The fact that swapping higher-level data structures can generally be noexcept only if swapping their lower-level constituents is noexcept should motivate you to offer noexcept swap functions whenever you can.*” conditionally noexcept definition. Book recommendation. Rules of noexcept: “*The fact that swapping higher-level data structures can generally be noexcept only if swapping their lower-level constituents is noexcept should motivate you to offer noexcept swap functions whenever you can.*”*

By now…

p. 111

*19.12.2016*

*Using context of noexcept “*By now, I hope you’re excited about the optimization opportunities that noexcept affords. Alas, I must temper your enthusiasm. Optimization is important, but correctness is more important. I noted at the beginning of this Item that noexcept is part of a function’s interface, so you should declare a function noexcept only if you are willing to commit to a noexcept implementation over the long term.*” Function interface is important concept.*

*Noexcept – небяспечны інструмент. Яго выдаленне можа абрушыць кліенсткі код: “*By now, I hope you’re excited about the optimization opportunities that noexcept affords. Alas, I must temper your enthusiasm. Optimization is important, but correctness is more important. I noted at the beginning of this Item that noexcept is part of a function’s interface, so you should declare a function noexcept only if you are willing to commit to a noexcept implementation over the long term. If you declare a function noexcept and later regret that decision, your options are bleak. You can remove noexcept from the function’s declaration (i.e., change its interface), thus running the risk of breaking client code. You can change the implementation such that an exception could escape, yet keep the original (now incorrect) exception specification. If you do that, your program will be terminated if an exception tries to leave the function. Or you can resign yourself to your existing implementation, abandoning whatever kindled your desire to change the implementation in the first place. None of these options is appealing.*”*

*Fact of the book: “*The fact of the matter is that most functions are exception-neutral. Such functions throw no exceptions themselves, but functions they call might emit one. When that happens, the exception-neutral function allows the emitted exception to pass through on its way to a handler further up the call chain. Exception-neutral functions are never noexcept, because they may emit such “just passing through” exceptions. Most functions, therefore, quite properly lack the noexcept designation*” Book recommendation and noexcept using context.*

*20.12.2016*

*p. 112*

*Function kind and noexcept using context: “*Some functions, however, have natural implementations that emit no exceptions, and for a few more—notably the move operations and swap—being noexcept can have such a significant payoff, it’s worth implementing them in a noexcept manner if at all possible.4 When you can honestly say that a function should never emit exceptions, you should definitely declare it noexcept. *” Book recommendation with some notes earlier.*

*Intbu translation: “*Twisting a function’s implementation to permit a noexcept declaration is the tail wagging the dog.*”*

*Kind of exception situation treatment: “*the hoops you’ll jump through to hide that from callers (e.g., catching all exceptions and replacing them with status codes or special return values)*”*

*Bad C++ practice and noexcept using context: ”* If a straightforward function implementation might yield exceptions (e.g., by invoking a function that might throw), the hoops you’ll jump through to hide that from callers (e.g., catching all exceptions and replacing them with status codes or special return values) will not only complicate your function’s implementation, it will typically complicate code at call sites, too. For example, callers may have to check for status codes or special return values. The runtime cost of those complications (e.g., extra branches, larger functions that put more pressure on instruction caches, etc.) could exceed any speedup you’d hope to achieve via noexcept, plus you’d be saddled with source code that’s more difficult to comprehend and maintain. That’d be poor software engineering.*” important moments of code quality: to maintain, to comprehend, to optimize run time behavior. The first two are to be counterpart of the last.*

*STL structure: “*The interface specifications for move operations on containers in the Standard Library lack noexcept. However, implementers are permitted to strengthen exception specifications for Standard Library functions, and, in practice, it is common for at least some container move operations to be declared noexcept. That practice exemplifies this Item’s advice. Having found that it’s possible to write container move operations such that exceptions aren’t thrown, implementers often declare the operations noexcept, even though the Standard does not require them to do so.*” intbu in practice.*

*C++ history and practice. Deallocation operations and destructors aspects. Noexcept using context: “*For some functions, being noexcept is so important, they’re that way by default. In C++98, it was considered bad style to permit the memory deallocation functions (i.e., operator delete and operator delete[]) and destructors to emit exceptions, and in C++11, this style rule has been all but upgraded to a language rule. By default, all memory deallocation functions and all destructors—both user-defined and compilergenerated — are implicitly noexcept. There’s thus no need to declare them noexcept. (Doing so doesn’t hurt anything, it’s just unconventional.)*”*

*Destructor structure and noexcept(false) construction using context: “*The only time a destructor is not implicitly noexcept is when a data member of the class (including inherited members and those contained inside other data members) is of a type that expressly states that its destructor may emit exceptions (e.g., declares it “noexcept(false)”). Such destructors are uncommon. There are none in the Standard Library, and if the destructor for an object being used by the Standard Library (e.g., because it’s in a container or was passed to an algorithm) emits an exception, the behavior of the program is undefined.*” Situation of undefined behavior.*

*p. 113*

*function types: with wide contacts and with narrow contracts: “*It’s worth noting that some library interface designers distinguish functions with wide contracts from those with narrow contracts. A function with a wide contract has no preconditions. Such a function may be called regardless of the state of the program, and it imposes no constraints on the arguments that callers pass it.5 Functions with wide contracts never exhibit undefined behavior. Functions without wide contracts have narrow contracts. For such functions, if a precondition is violated, results are undefined.*” For better understanding of concept “precondition”.*

*Noexcept using contex and function with wide contracts: “*If you’re writing a function with a wide contract and you know it won’t emit exceptions, following the advice of this Item and declaring it noexcept is easy. For functions with narrow contracts, the situation is trickier. For example, suppose you’re writing a function f taking a std::string parameter, and suppose f’s natural implementation never yields an exception. That suggests that f should be declared noexcept.*”*

*Function with narrow conracts aspects. It is necessary to indicate responsibility of preconditions satisfying: if inside function or by user. : “*Now suppose that f has a precondition: the length of its std::string parameter doesn’t exceed 32 characters. If f were to be called with a std::string whose length is greater than 32, behavior would be undefined, because a precondition violation by definition results in undefined behavior. f is under no obligation to check this precondition, because functions may assume that their preconditions are satisfied. (Callers are responsible for ensuring that such assumptions are valid.) Even with a precondition, then, declaring f noexcept seems appropriate:

void f(const std::string& s) noexcept; // precondition:

// s.length() <= 32*”*

*Exception using context: for debugging: “*But suppose that f’s implementer chooses to check for precondition violations. Checking isn’t required, but it’s also not forbidden, and checking the precondition could be useful, e.g., during system testing. Debugging an exception that’s been thrown is generally easier than trying to track down the cause of undefined behavior. But how should a precondition violation be reported such that a test harness or a client error handler could detect it? A straightforward approach would be to throw a “precondition was violated” exception, but if f is declared noexcept, that would be impossible; throwing an exception would lead to program termination. For this rea*” noexcept using context.*

*Undefined behavior of program is not the same as undefined behavior of function: “*“Regardless of the state of the program” and “no constraints” doesn’t legitimize programs whose behavior is already undefined. For example, std::vector::size has a wide contract, but that doesn’t require that it behave reasonably if you invoke it on a random chunk of memory that you’ve cast to a std::vector. The result of the cast is undefined, so there are no behavioral guarantees for the program containing the cast.*”*

*p. 114*

*Noexcept using context and library quality: “*For this reason, library designers who distinguish wide from narrow contracts generally reserve noexcept for functions with wide contracts.*”*

*Compiler limitation: “*As a final point, let me elaborate on my earlier observation that compilers typically offer no help in identifying inconsistencies between function implementations and their exception specifications. Consider this code, which is perfectly legal:

void setup(); // functions defined elsewhere

void cleanup();

void doWork() noexcept

{

setup(); // set up work to be done

… // do the actual work

cleanup(); // perform cleanup actions

}

Here, doWork is declared noexcept, even though it calls the non-noexcept functions setup and cleanup. This seems contradictory, but it could be that setup and cleanup document that they never emit exceptions, even though they’re not declared that way.*”*

*Historical reasons for functions not to be noexcept. C++ history: “*Here, doWork is declared noexcept, even though it calls the non-noexcept functions setup and cleanup. This seems contradictory, but it could be that setup and cleanup document that they never emit exceptions, even though they’re not declared that way. There could be good reasons for their non-noexcept declarations. For example, they might be part of a library written in C. (Even functions from the C Standard Library that have been moved into the std namespace lack exception specifications, e.g., std::strlen isn’t declared noexcept.) Or they could be part of a C++98 library that decided not to use C++98 exception specifications and hasn’t yet been revised for C++11.*”*

*C++ structure and explanation of such compiler property: “*Because there are legitimate reasons for noexcept functions to rely on code lacking the noexcept guarantee, C++ permits such code, and compilers generally don’t issue warnings about it.*”*

*Function interface – C++ concept.*

*Book structure. Resume: “*Things to Remember

• noexcept is part of a function’s interface, and that means that callers may depend on it.

• noexcept functions are more optimizable than non-noexcept functions.

• noexcept is particularly valuable for the move operations, swap, memory deallocation functions, and destructors.

• Most functions are exception-neutral rather than noexcept.*” The last punkt is C++ program statistical property and structure of function type using context.*

*p. 115*

### **Item 15: Use constexpr whenever possible**

*Book recommendation: “***Use constexpr whenever possible***”. C++ practice.*

*Constexp structure: “*When applied to objects, it’s essentially a beefed-up form of const, but when applied to functions, it has a quite different meaning.*” Its meaning depend on entity to which it is being applied.*

*Advantage of concept: “*Cutting through the confusion is worth the trouble, because when constexpr corresponds to what you want to express, you definitely want to use it.*”*

*Constexpr conception: “*Conceptually, constexpr indicates a value that’s not only constant, it’s known during compilation.*”*

*Constexpr structure: “*The concept is only part of the story, though, because when constexpr is applied to functions, things are more nuanced than this suggests. Lest I ruin the surprise ending, for now I’ll just say that you can’t assume that the results of constexpr functions are const, nor can you take for granted that their values are known during compilation. Perhaps most intriguingly, these things are features. It’s good that constexpr functions need not produce results that are const or known during compilation!*” kinds of values: known during compilation, const.*

*Constexpr objects: “*But let’s begin with constexpr objects. Such objects are, in fact, const, and they do, in fact, have values that are known at compile time. (Technically, their values are determined during translation, and translation consists not just of compilation but also of linking. Unless you write compilers or linkers for C++, however, this has no effect on you, so you can blithely program as if the values of constexpr objects were determined during compilation.)*” C++ language realization: translation, compilation, linking. It is necessary to better understand.*

*Embedded systems important moments: “*Values known during compilation are privileged. They may be placed in read-only memory, for example, and, especially for developers of embedded systems, this can be a feature of considerable importance.*” Advantage of const values.*

*Advantage of const int values. C++ context: requires an intergral expression. Its concretizations: “*Of broader applicability is that integral values that are constant and known during compilation can be used in contexts where C++ requires an integral constant expression. Such contexts include specification of array sizes, integral template arguments (including lengths of std::array objects), enumerator values, alignment specifiers, and more.*”*

*Constexpr using context: “*If you want to use a variable for these kinds of things, you certainly want to declare it constexpr, because then compilers will ensure that it has a compile-time value:

int sz; // non-constexpr variable

…

constexpr auto arraySize1 = sz; // error! sz's value not

// known at compilation

std::array<int, sz> data1; // error! same problem

constexpr auto arraySize2 = 10; // fine, 10 is a

// compile-time constant

std::array<int, arraySize2> data2; // fine, arraySize2

// is constexpr*”*

*p. 116*

*Differenct between const and constexpr. Their structure: “*Note that const doesn’t offer the same guarantee as constexpr, because const objects need not be initialized with values known during compilation:

int sz; // as before

…

const auto arraySize = sz; // fine, arraySize is

// const copy of sz

std::array<int, arraySize> data; // error! arraySize's value

// not known at compilation

Simply put, all constexpr objects are const, but not all const objects are con stexpr. If you want compilers to guarantee that a variable has a value that can be used in contexts requiring compile-time constants, the tool to reach for is constexpr, not const.*”*

*Kinds of values: conmpile-time constants, run-time values.*

*Usage scenarios is common category of programming languages sysved.*

*Using constexpr with functions: “*Usage scenarios for constexpr objects become more interesting when constexpr functions are involved. Such functions produce compile-time constants when they are called with compile-time constants. If they’re called with values not known until runtime, they produce runtime values. This may sound as if you don’t know what they’ll do, but that’s the wrong way to think about it. The right way to view it is this: constexpr functions can be used in contexts that demand compile-time constants. If the values of the arguments you pass to a constexpr function in such a context are known during compilation, the result will be computed during compilation. If any of the arguments’ values is not known during compilation, your code will be rejected.

• When a constexpr function is called with one or more values that are not known during compilation, it acts like a normal function, computing its result at runtime. This means you don’t need two functions to perform the same operation, one for compile-time constants and one for all other values. The constexpr function does it all.*” advantage of constexpr.*

*Book example of such function using context: “*Suppose we need a data structure to hold the results of an experiment that can be run in a variety of ways. For example, the lighting level can be high, low, or off during the course of the experiment, as can the fan speed and the temperature, etc. If there are n environmental conditions relevant to the experiment, each of which has three possi*”*

*p. 117*

*example continue: “*ble states, the number of combinations is 3n. Storing experimental results for all combinations of conditions thus requires a data structure with enough room for 3n values. Assuming each result is an int and that n is known (or can be computed) during compilation, a std::array could be a reasonable data structure choice. But we’d need a way to compute 3n during compilation. The C++ Standard Library provides std::pow, which is the mathematical functionality we need, but, for our purposes, there are two problems with it. First, std::pow works on floating-point types, and we need an integral result. Second, std::pow isn’t constexpr (i.e., isn’t guaranteed to return a compile-time result when called with compile-time values), so we can’t use it to specify a std::array’s size.*” std::pow features. Aspects of combinatorics.*

*Example continue: “*Fortunately, we can write the pow we need. I’ll show how to do that in a moment, but first let’s look at how it could be declared and used:

constexpr // pow's a constexpr func

int pow(int base, int exp) noexcept // that never throws

{

… // impl is below

}

constexpr auto numConds = 5; // # of conditions

std::array<int, pow(3, numConds)> results; // results has

// 3^numConds

// elements

*” well-understood example.*

*Constexpr structure: “*Recall that the constexpr in front of pow doesn’t say that pow returns a const value, it says that if base and exp are compile-time constants, pow’s result may be used as a compile-time constant. If base and/or exp are not compile-time constants, pow’s result will be computed at runtime.*”*

*Example continue: “*That means that pow can not only be called to do things like compile-time-compute the size of a std::array, it can also be called in runtime contexts such as this:

auto base = readFromDB("base"); // get these values

auto exp = readFromDB("exponent"); // at runtime

auto baseToExp = pow(base, exp); // call pow function

// at runtime*”*

*C++ history and constexpr structure: restrictions: “*Because constexpr functions must be able to return compile-time results when called with compile-time values, restrictions are imposed on their implementations. The restrictions differ between C++11 and C++14. In C++11, constexpr functions may contain no more than a single executable statement: a return. That sounds more limiting than it is, because two tricks can be used*”*

*p. 118*

*C++ history and constexpr structure: restrictions: “*to extend the expressiveness of constexpr functions beyond what you might think. First, the conditional “?:” operator can be used in place of if-else statements, and second, recursion can be used instead of loops. Pow can therefore be implemented like this:

constexpr int pow(int base, int exp) noexcept

{

return (exp == 0 ? 1 : base \* pow(base, exp - 1));

}

This works, but it’s hard to imagine that anybody except a hard-core functional programmer would consider it pretty. In C++14, the restrictions on constexpr functions are substantially looser, so the following implementation becomes possible:

constexpr int pow(int base, int exp) noexcept // C++14

{

auto result = 1;

for (int i = 0; i < exp; ++i) result \*= base;

return result;

}

constexpr functions are limited to taking and returning literal types, which essentially means types that can have values determined during compilation.*” Literal types definition. Example of hard-core.*

*Literal types concretizations: “*In C++11, all built-in types except void qualify, but user-defined types may be literal, too, because constructors and other member functions may be constexpr:

class Point {

public:

constexpr Point(double xVal = 0, double yVal = 0) noexcept

: x(xVal), y(yVal)

{}

constexpr double xValue() const noexcept { return x; }

constexpr double yValue() const noexcept { return y; }

void setX(double newX) noexcept { x = newX; }

void setY(double newY) noexcept { y = newY; }

private:

double x, y;

};

Here, the Point constructor can be declared constexpr, because if the arguments passed to it are known during compilation, the value of the data members of the constructed Point can also be known during compilation. Points so initialized could thus be constexpr:

constexpr Point p1(9.4, 27.7); // fine, "runs" constexpr

// ctor during compilation

constexpr Point p2(28.8, 5.3); // also fine*” const objects example. Constexpr using context.*

*Example continue: “*Similarly, the getters xValue and yValue can be constexpr, because if they’re invoked on a Point object with a value known during compilation (e.g., a constexpr Point object), the values of the data members x and y can be known during compilation. That makes it possible to write constexpr functions that call Point’s getters and to initialize constexpr objects with the results of such functions:

constexpr

Point midpoint(const Point& p1, const Point& p2) noexcept

{

return { (p1.xValue() + p2.xValue()) / 2, // call constexpr

(p1.yValue() + p2.yValue()) / 2 }; // member funcs

}

constexpr auto mid = midpoint(p1, p2); // init constexpr

// object w/result of

// constexpr function*”*

*Const objects advantage: “*This is very exciting. It means that the object mid, though its initialization involves calls to constructors, getters, and a non-member function, can be created in readonly memory! It means you could use an expression like mid.xValue() \* 10 in an argument to a template or in an expression specifying the value of an enumerator!*” kind of memory – read-only memory. Const object using context: enumeration, template argument.*

*Modern C++ feature: “*It means that the traditionally fairly strict line between work done during compilation and work done at runtime begins to blur, and some computations traditionally done at runtime can migrate to compile time. The more code taking part in the migration, the faster your software will run. (Compilation may take longer, however.)*” Advantage of compilation time values.*

*Floating-point types using context: restrictions: “*Because Point::xValue returns double, the type of mid.xValue() \* 10 is also double. Floating-point types can’t be used to instantiate templates or to specify enumerator values, but they can be used as part of larger expressions that yield integral types. For example, static\_cast<int>(mid.xValue() \* 10) could be used to instantiate a template or to specify an enumerator value.*” templates, enumerations. Trick to bypass this.*

In C++11, two…

p. 119 down.

*21.12.2016*

*С++ history and difference between C++11 and C++14: “*In C++11, two restrictions prevent Point’s member functions setX and setY from being declared constexpr. First, they modify the object they operate on, and in C++11, constexpr member functions are implicitly const. Second, they have void return types, and void isn’t a literal type in C++11. Both these restrictions are lifted in C++14, so in C++14, even Point’s setters can be constexpr:*” constexpr evolution and history.*

*p. 120*

*example of constexpr using in C++14: “*

class Point {

public:

…

constexpr void setX(double newX) noexcept // C++14

{ x = newX; }

constexpr void setY(double newY) noexcept // C++14

{ y = newY; }

…

};

That makes it possible to write functions like this:

// return reflection of p with respect to the origin (C++14)

constexpr Point reflection(const Point& p) noexcept

{

Point result; // create non-const Point

result.setX(-p.xValue()); // set its x and y values

result.setY(-p.yValue());

return result; // return copy of it

}

Client code could look like this:

constexpr Point p1(9.4, 27.7); // as above

constexpr Point p2(28.8, 5.3);

constexpr auto mid = midpoint(p1, p2);

constexpr auto reflectedMid = // reflectedMid's value is

reflection(mid); // (-19.1 -16.5) and known

// during compilation*”*

*Advantage of constexpr and book recommendation. Usefulness for objects and functions: “*The advice of this Item is to use constexpr whenever possible, and by now I hope it’s clear why: both constexpr objects and constexpr functions can be employed in a wider range of contexts than non-constexpr objects and functions. By using constexpr whenever possible, you maximize the range of situations in which your objects and functions may be used.*”*

*Constexpr is part of function or class interface: “*It’s important to note that constexpr is part of an object’s or function’s interface. Constexpr proclaims “I can be used in a context where C++ requires a constant expression.” If you declare an object or function constexpr, clients may use it in such contexts. If you later decide that your use of constexpr was a mistake and you remove it, you may cause arbitrarily large amounts of client code to stop compiling.*”*

*Аспекты маёй сістэмы: а калі такое меркаванне наконт constexpr у будучыні зменшыцца? То што? Трэба прадугледзець у сістэме магчымасць змяняць сувязі, адмяняць іх. Паразважаць.*

*p. 121*

*I/O using context and constexpr limitations: “*(The simple act of adding I/O to a function for debugging or performance tuning could lead to such a problem, because I/O statements are generally not permitted in constexpr functions.) Part of “whenever possible” in “Use constexpr whenever possible” is your willingness to make a long-term commitment to the constraints it imposes on the objects and functions you apply it to.*” nuance of constexpr. Intbu do such limitations exist? Constexpr functions can be used in ordinary context. Intbu?*

*Book structure. Resume. “*Things to Remember

• constexpr objects are const and are initialized with values known during compilation.

• constexpr functions can produce compile-time results when called with arguments whose values are known during compilation.

• constexpr objects and functions may be used in a wider range of contexts than non-constexpr objects and functions.

• constexpr is part of an object’s or function’s interface.*”*

### **Item 16: Make const member fucntions thread safe**

*Book recommendation.*

*Example of problem. Kind of problem “thread safety” and its concretization: “*If we’re working in a mathematical domain, we might find it convenient to have a class representing polynomials. Within this class, it would probably be useful to have a function to compute the root(s) of a polynomial, i.e., values where the polynomial evaluates to zero. Such a function would not modify the polynomial, so it’d be natural to declare it const:

class Polynomial {

public:

using RootsType = // data structure holding values

std::vector<double>; // where polynomial evals to zero

… // (see Item 9 for info on "using")

RootsType roots() const;

…

};

Computing the roots of a polynomial can be expensive, so we don’t want to do it if we don’t have to. And if we do have to do it, we certainly don’t want to do it more than once. We’ll thus cache the root(s) of the polynomial if we have to compute*” C++ concept – const member functions.*

*p. 122*

*example continue: “*them, and we’ll implement roots to return the cached value. Here’s the basic approach:

class Polynomial {

public:

using RootsType = std::vector<double>;

RootsType roots() const

{

if (!rootsAreValid) { // if cache not valid

… // compute roots,

// store them in rootVals

rootsAreValid = true;

}

return rootVals;

}

private:

mutable bool rootsAreValid{ false }; // see Item 7 for info

mutable RootsType rootVals{}; // on initializers

};*” book structure (Item 7 about initializers). To remind what is mutable.*

*“*Conceptually, roots doesn’t change the Polynomial object on which it operates, but, as part of its caching activity, it may need to modify rootVals and rootsAreValid. That’s a classic use case for mutable, and that’s why it’s part of the declarations for these data members.*” using context of mutable data members. These members can be modified in const functions?*

*Example continue: “*Imagine now that two threads simultaneously call roots on a Polynomial object:

Polynomial p;

…

/\*----- Thread 1 ----- \*/ /\*------- Thread 2 ------- \*/

auto rootsOfP = p.roots(); auto valsGivingZero = p.roots();

This client code is perfectly reasonable. roots is a const member function, and that means it represents a read operation. Having multiple threads perform a read operation without synchronization is safe. At least it’s supposed to be. In this case, it’s not, because inside roots, one or both of these threads might try to modify the data members rootsAreValid and rootVals.*” Problem indication. Thread safety situation. This is not valid when we have const functions and mutable members. Mutable is useful for caching. Its using context.*

*Synchronization using context: “*That means that this code could have different threads reading and writing the same memory without synchronization, and that’s the definition of a data race. This code has undefined behavior.*”*

*p. 123*

*const doesn’t mean thread safe: “*The problem is that roots is declared const, but it’s not thread safe. The const declaration is as correct in C++11 as it would be in C++98 (retrieving the roots of a polynomial doesn’t change the value of the polynomial), so what requires rectification is the lack of thread safety.*”*

*Mutex using context: “*The easiest way to address the issue is the usual one: employ a mutex:

class Polynomial {

public:

using RootsType = std::vector<double>;

RootsType roots() const

{

std::lock\_guard<std::mutex> g(m); // lock mutex

if (!rootsAreValid) { // if cache not valid

… // compute/store roots

rootsAreValid = true;

}

return rootVals;

} // unlock mutex

private:

mutable std::mutex m;

mutable bool rootsAreValid{ false };

mutable RootsType rootVals{};

};

The std::mutex m is declared mutable, because locking and unlocking it are nonconst member functions, and within roots (a const member function), m would otherwise be considered a const object.*” intbu what is mutex. What is std::lock\_guard and std::mutex? Mutex using technique.*

*Kind of type – move-only type. Std::mutex structure. Aspects of using such types: “*It’s worth noting that because std::mutex is a move-only type (i.e., a type that can be moved, but not copied), a side effect of adding m to Polynomial is that Polynomial loses the ability to be copied. It can still be moved, however.*” how can it be realized?*

*Context using of std::atomic and std::mutex: “*In some situations, a mutex is overkill. For example, if all you’re doing is counting how many times a member function is called, a std::atomic counter (i.e, one where other threads are guaranteed to see its operations occur indivisibly—see Item 40) will often be a less expensive way to go. (Whether it actually is less expensive depends on the hardware you’re running on and the implementation of mutexes in your Standard Library.)*” Book structure (Item 40). Mutex and atomic speed depends on hardware in non-linear manner? Intbu.*

*p. 124*

*std::atomic example: “*Here’s how you can employ a std::atomic to count calls:

class Point { // 2D point

public:

…

double distanceFromOrigin() const noexcept // see Item 14

{ // for noexcept

++callCount; // atomic increment

return std::sqrt((x \* x) + (y \* y));

}

private:

mutable std::atomic<unsigned> callCount{ 0 };

double x, y;

};

Like std::mutexes, std::atomics are move-only types, so the existence of call Count in Point means that Point is also move-only.*” std::atomic property. It is move-only type.*

*Std::atomic and std::mutex using context: “*Because operations on std::atomic variables are often less expensive than mutex acquisition and release, you may be tempted to lean on std::atomics more heavily than you should. For example, in a class caching an expensive-to-compute int, you might try to use a pair of std::atomic variables instead of a mutex:

class Widget {

public:

…

int magicValue() const

{

if (cacheValid) return cachedValue;

else {

auto val1 = expensiveComputation1();

auto val2 = expensiveComputation2();

cachedValue = val1 + val2; // uh oh, part 1

cacheValid = true; // uh oh, part 2

return cachedValue;

}

}

private:

mutable std::atomic<bool> cacheValid{ false };

mutable std::atomic<int> cachedValue;

};*”*

*Example continue: “*This will work, but sometimes it will work a lot harder than it should. Consider:

• A thread calls Widget::magicValue, sees cacheValid as false, performs the two expensive computations, and assigns their sum to cachedValue.

• At that point, a second thread calls Widget::magicValue, also sees cacheValid as false, and thus carries out the same expensive computations that the first thread has just finished. (This “second thread” may in fact be several other threads.)*” std::atomic implementation. Using context of std::mutex and std::atomic.*

*Example continue: “*Such behavior is contrary to the goal of caching. Reversing the order of the assignments to cachedValue and CacheValid eliminates that problem, but the result is even worse:

class Widget {

public:

…

int magicValue() const

{

if (cacheValid) return cachedValue;

else {

auto val1 = expensiveComputation1();

auto val2 = expensiveComputation2();

cacheValid = true; // uh oh, part 1

return cachedValue = val1 + val2; // uh oh, part 2

}

}

…

};

Imagine that cacheValid is false, and then:

• One thread calls Widget::magicValue and executes through the point where cacheValid is set to true.

• At that moment, a second thread calls Widget::magicValue and checks cache Valid. Seeing it true, the thread returns cachedValue, even though the first thread has not yet made an assignment to it. The returned value is therefore incorrect.*” example of thread-safety violation using std::atomic.*

*p. 126*

*using context of std::atomic and std::mutex: “*There’s a lesson here. For a single variable or memory location requiring synchronization, use of a std::atomic is adequate, but once you get to two or more variables or memory locations that require manipulation as a unit, you should reach for a mutex.*”*

*Std::mutex example: “*For Widget::magicValue, that would look like this:

class Widget {

public:

…

int magicValue() const

{

std::lock\_guard<std::mutex> guard(m); // lock m

if (cacheValid) return cachedValue;

else {

auto val1 = expensiveComputation1();

auto val2 = expensiveComputation2();

cachedValue = val1 + val2;

cacheValid = true;

return cachedValue;

}

} // unlock m

…

private:

mutable std::mutex m;

mutable int cachedValue; // no longer atomic

mutable bool cacheValid{ false }; // no longer atomic

};*” intbu the principle of its functioning.*

*Const functions thread safety using context: “*Now, this Item is predicated on the assumption that multiple threads may simultaneously execute a const member function on an object. If you’re writing a сonst member function where that’s not the case—where you can guarantee that there will never be more than one thread executing that member function on an object—the thread safety of the function is immaterial. For example, it’s unimportant whether member functions of classes designed for exclusively single-threaded use are thread safe. In such cases, you can avoid the costs associated with mutexes and std::atomics, as well as the side effect of their rendering the classes containing them move-only.*” Disadvantages of std::mutex and std::atomic.*

*Frequency of such using context meeting: “*However, such threading-free scenarios are increasingly uncommon, and they’re likely to become rarer still. The safe bet is that const member functions will be subject to concurrent execution, and that’s why you should ensure that your const member functions are thread safe.*” Good C++ practice and book recommendation.*

*p. 127*

*book structure. Resume: “*Things to Remember

• Make const member functions thread safe unless you’re certain they’ll never be used in a concurrent context.

• Use of std::atomic variables may offer better performance than a mutex, but they’re suited for manipulation of only a single variable or memory location.*” comparison of std::mutex and std::atomic.*

## **Item 17: Understand special member function generation.**

*Intbu translation: “*In official C++ parlance, the special member functions are the ones that C++ is willing to generate on its own.*” Special member functions: feature of C++.*

*Their kinds and C++ history: “*C++98 has four such functions: the default constructor, the destructor, the copy constructor, and the copy assignment operator. There’s fine print, of course. These functions are generated only if they’re needed, i.e., if some code uses them without their being expressly declared in the class. A default constructor is generated only if the class declares no constructors at all. (This prevents compilers from creating a default constructor for a class where you’ve specified that constructor arguments are required.) Generated special member functions are implicitly public and inline, and they’re nonvirtual unless the function in question is a destructor in a derived class inheriting from a base class with a virtual destructor. In that case, the compiler-generated destructor for the derived class is also virtual.*” special functions structure: default constructor, default destructor. Their properties: inline public non-virtual (except destructor - intbu).*

*)))))) “*But you already know these things. Yes, yes, ancient history: Mesopotamia, the Shang dynasty, FORTRAN, C++98.*”*

*Compiler structure, C++ history and special member functions evolution: “*But times have changed, and the rules for special member function generation in C++ have changed with them. It’s important to be aware of the new rules, because few things are as central to effective C++ programming as knowing when compilers silently insert member functions into your classes.*” importance for modern C++.*

*Another 2 special member function kinds: move constructor and the move assignment operator: “*As of C++11, the special member functions club has two more inductees: the move constructor and the move assignment operator. Their signatures are:

class Widget {

public:

…

Widget(Widget&& rhs); // move constructor

Widget& operator=(Widget&& rhs); // move assignment operator

…

};*” Their structure.*

*p. 128*

*this special member function examples: “*The rules governing their generation and behavior are analogous to those for their copying siblings. The move operations are generated only if they’re needed, and if they are generated, they perform “memberwise moves” on the non-static data members of the class. That means that the move constructor move-constructs each nonstatic data member of the class from the corresponding member of its parameter rhs, and the move assignment operator move-assigns each non-static data member from its parameter. The move constructor also move-constructs its base class parts (if there are any), and the move assignment operator move-assigns its base class parts.*” class structure: non-static memebers, static members, base class parts.*

*Move-constructing realization: “*Now, when I refer to a move operation move-constructing or move-assigning a data member or base class, there is no guarantee that a move will actually take place. “Memberwise moves” are, in reality, more like memberwise move requests, because types that aren’t move-enabled (i.e., that offer no special support for move operations, e.g., most C++98 legacy classes) will be “moved” via their copy operations. The heart of each memberwise “move” is application of std::move to the object to be moved from, and the result is used during function overload resolution to determine whether a move or a copy should be performed. Item 23 covers this process in detail. For this Item, simply remember that a memberwise move consists of move operations on data members and base classes that support move operations, but a copy operation for those that don’t.*” Std::move purpose. Intbu the sense of this centence. Move-constructing realization. Book structure (Item 23).*

*Move operations structure: “*As is the case with the copy operations, the move operations aren’t generated if you declare them yourself.*”*

*Difference between copy operations and move operations: “*However, the precise conditions under which they are generated differ a bit from those for the copy operations.*”*

*Copy operations property: “*The two copy operations are independent: declaring one doesn’t prevent compilers from generating the other. So if you declare a copy constructor, but no copy assignment operator, then write code that requires copy assignment, compilers will generate the copy assignment operator for you. Similarly, if you declare a copy assignment operator, but no copy constructor, yet your code requires copy construction, compilers will generate the copy constructor for you. That was true in C++98, and it’s still true in C++11.*” cope special member function properties.*

*Move operations property: “*The two move operations are not independent. If you declare either, that prevents compilers from generating the other. The rationale is that if you declare, say, a move constructor for your class, you’re indicating that there’s something about how move construction should be implemented that’s different from the default memberwise move that compilers would generate. And if there’s something wrong with memberwise move construction, there’d probably be something wrong with memberwise move assignment, too. So declaring a move constructor prevents a move assignment operator from being generated, and declaring a move assignment operator prevents compilers from generating a move constructor.*” why the same rules are not valid for copy constructors. Intbu.*

*Move operation using context and connection with copy operations: “*Furthermore, move operations won’t be generated for any class that explicitly declares a copy operation. The justification is that declaring a copy operation (construction or assignment) indicates that the normal approach to copying an object (memberwise copy) isn’t appropriate for the class, and compilers figure that if memberwise copy isn’t appropriate for the copy operations, memberwise move probably isn’t appropriate for the move operations.*” move operations are not generated if you explicitely define copy operations.*

*copy operations using context. Connection with move operation: “*struction or assignment) indicates that the normal approach to copying an object (memberwise copy) isn’t appropriate for the class, and compilers figure that if memberwise copy isn’t appropriate for the copy operations, memberwise move probably isn’t appropriate for the move operations. After all, if memberwise move isn’t the proper way to move an object, there’s no reason to expect that memberwise copy is the proper way to copy it. This may sound like it could break C++98 code, because the conditions under which the copy operations are enabled are more constrained in C++11 than in C++98, but this is not the case. C++98 code can’t have move operations, because there was no such thing as “moving” objects in C++98. The only way a legacy class can have user-declared move operations is if they were added for C++11, and classes that are modified to take advantage of move semantics have to play by the C++11 rules for special member function generation.*” book structure (Item 11). Deleting operations. This is example of backward compatibility? Intbu.*

Perhaps…

p. 129

*22.12.2016*

*Rule of Three. C++ practice. Class structure: “*Perhaps you’ve heard of a guideline known as the Rule of Three. The Rule of Three states that if you declare any of a copy constructor, copy assignment operator, or destructor, you should declare all three. It grew out of the observation that the need to take over the meaning of a copy operation almost always stemmed from the class performing some kind of resource management, and that almost always implied that (1) whatever resource management was being done in one copy operation probably needed to be done in the other copy operation and (2) the class destructor would also be participating in management of the resource (usually releasing it). The classic resource to be managed was memory, and this is why all Standard Library classes that manage memory (e.g., the STL containers that perform dynamic memory management) all declare “the big three”: both copy operations and a destructor.*” STL structure. Memory is type of resourse. STL classes that manage memory.*

*C++ history and example of influence of compatibility on product quality: “*A consequence of the Rule of Three is that the presence of a user-declared destructor indicates that simple memberwise copy is unlikely to be appropriate for the copying operations in the class. That, in turn, suggests that if a class declares a destructor, the copy operations probably shouldn’t be automatically generated, because they wouldn’t do the right thing. At the time C++98 was adopted, the significance of this line of reasoning was not fully appreciated, so in C++98, the existence of a userdeclared destructor had no impact on compilers’ willingness to generate copy operations. That continues to be the case in C++11, but only because restricting the conditions under which the copy operations are generated would break too much legacy code.*” Rules of Three structure. Structure of special member functions.*

*C++11 structure. Special member functions structure: “*The reasoning behind the Rule of Three remains valid, however, and that, combined with the observation that declaration of a copy operation precludes the implicit generation of the move operations, motivates the fact that C++11 does not generate move operations for a class with a user-declared destructor.*”*

*p. 130*

*condition of move operations generation: “*So move operations are generated for classes (when needed) only if these three things are true:

• No copy operations are declared in the class.

• No move operations are declared in the class.

• No destructor is declared in the class.*”*

*Conditions of copy operations generation: “*At some point, analogous rules may be extended to the copy operations, because C++11 deprecates the automatic generation of copy operations for classes declaring copy operations or a destructor.*”*

*C++ good practice. Book recommendation: “*This means that if you have code that depends on the generation of copy operations in classes declaring a destructor or one of the copy operations, you should consider upgrading these classes to eliminate the dependence.”

*Default keyword using context and purpose: “*Provided the behavior of the compiler-generated functions is correct (i.e, if memberwise copying of the class’s non-static data members is what you want), your job is easy, because C++11’s “=default” lets you say that explicitly:

class Widget {

public:

…

~Widget(); // user-declared dtor

… // default copy ctor

Widget(const Widget&) = default; // behavior is OK

Widget& // default copy assign

operator=(const Widget&) = default; // behavior is OK

…

};*”*

*Polymorphic base classes: “*This approach is often useful in polymorphic base classes, i.e., classes defining interfaces through which derived class objects are manipulated. Polymorphic base classes normally have virtual destructors, because if they don’t, some operations (e.g., the use of delete or typeid on a derived class object through a base class pointer or reference) yield undefined or misleading results. Unless a class inherits a destructor that’s already virtual, the only way to make a destructor virtual is to explicitly declare it that way. Often, the default implementation would be correct, and “=default” is a good way to express that. However, a user-declared destructor suppresses generation of the move operations, so if movability is to be supported, “=default” often finds a second application.*” Intbu what is polymorphic base classes? Default using context.*

*Intbu this default using context: “*Declaring the move operations disables the copy operations, so if copyability is also desired, one more round of “=default” does the job:

class Base {

public:

virtual ~Base() = default; // make dtor virtual

Base(Base&&) = default; // support moving

Base& operator=(Base&&) = default;

Base(const Base&) = default; // support copying

Base& operator=(const Base&) = default;

…

};*” this is indication that you want to use default move and copy operations even if you defined destructor? Is it so?*

*p. 131*

*book recommendation. Intbu: “*In fact, even if you have a class where compilers are willing to generate the copy and move operations and where the generated functions would behave as you want, you may choose to adopt a policy of declaring them yourself and using “= default” for their definitions. It’s more work, but it makes your intentions clearer, and it can help you sidestep some fairly subtle bugs.*”*

*Example of these bugs: “*For example, suppose you have a class representing a string table, i.e., a data structure that permits fast lookups of string values via an integer ID:

class StringTable {

public:

StringTable() {}

… // functions for insertion, erasure, lookup,

// etc., but no copy/move/dtor functionality

private:

std::map<int, std::string> values;

};

Assuming that the class declares no copy operations, no move operations, and no destructor, compilers will automatically generate these functions if they are used. That’s very convenient. But suppose that sometime later, it’s decided that logging the default construction and the destruction of such objects would be useful. Adding that functionality is easy:

class StringTable {

public:

StringTable()

{ makeLogEntry("Creating StringTable object"); } // added

~StringTable() // also

{ makeLogEntry("Destroying StringTable object"); } // added

… // other funcs as before

private:

std::map<int, std::string> values; // as before

};*” Logging is using context of user-defined constructors and destructors.*

*p. 132*

*Example continue: “*This looks reasonable, but declaring a destructor has a potentially significant side effect: it prevents the move operations from being generated. However, creation of the class’s copy operations is unaffected. The code is therefore likely to compile, run, and pass its functional testing. That includes testing its move functionality, because even though this class is no longer move-enabled, requests to move it will compile and run. Such requests will, as noted earlier in this Item, cause copies to be made. Which means that code “moving” StringTable objects actually copies them, i.e., copies the underlying std::map<int, std::string> objects. And copying a std ::map<int, std::string> is likely to be orders of magnitude slower than moving it. The simple act of adding a destructor to the class could thereby have introduced a significant performance problem! Had the copy and move operations been explicitly defined using “= default”, the problem would not have arisen.*” C++ subtle bug concerning move operation and user-defined copy operations and destructor. Advantage and usefulness of default keyword.*

*C++ history and structure: “*you may wonder when I’ll turn my attention to the two other special member functions, the default constructor and the destructor. That time is now, but only for this sentence, because almost nothing has changed for these member functions: the rules in C++11 are nearly the same as in C++98.*”*

*Default constructor governing rules: “*Default constructor: Same rules as C++98. Generated only if the class contains no user-declared constructors.*”*

*Destructor governing rules: “*Destructor: Essentially same rules as C++98; sole difference is that destructors are noexcept by default (see Item 14). As in C++98, virtual only if a base class destructor is virtual.*” what is virtual destructor. Intbu.*

*Copy constructor: “*Copy constructor: Same runtime behavior as C++98: memberwise copy construction of non-static data members. Generated only if the class lacks a userdeclared copy constructor. Deleted if the class declares a move operation. Generation of this function in a class with a user-declared copy assignment operator or destructor is deprecated.*” For better understanding worf “depricating”.*

*Copy assignment operator governing rules: “*Copy assignment operator: Same runtime behavior as C++98: memberwise copy assignment of non-static data members. Generated only if the class lacks a user-declared copy assignment operator. Deleted if the class declares a move operation. Generation of this function in a class with a user-declared copy constructor or destructor is deprecated.*”*

*p. 133*

*move constructor and move assignment opearator governing rules: “*Move constructor and move assignment operator: Each performs memberwise moving of non-static data members. Generated only if the class contains no userdeclared copy operations, move operations, or destructor. *”*

*threre was also definition of these concepts.*

*Templates and special member functions: “*Note that there’s nothing in the rules about the existence of a member function template preventing compilers from generating the special member functions. That means that if Widget looks like this,

class Widget {

…

template<typename T> // construct Widget

Widget(const T& rhs); // from anything

template<typename T> // assign Widget

Widget& operator=(const T& rhs); // from anything

…

};

compilers will still generate copy and move operations for Widget (assuming the usual conditions governing their generation are fulfilled), even though these templates could be instantiated to produce the signatures for the copy constructor and copy assignment operator. (That would be the case when T is Widget.)*”*

*Importance of this topic: “*In all likelihood, this will strike you as an edge case barely worth acknowledging, but there’s a reason I’m mentioning it. Item 26 demonstrates that it can have important consequences.*” book structure (Item 26).*

*Book structure. Resume: “*Things to Remember

• The special member functions are those compilers may generate on their own: default constructor, destructor, copy operations, and move operations.

• Move operations are generated only for classes lacking explicitly declared move operations, copy operations, and a destructor.

• The copy constructor is generated only for classes lacking an explicitly declared copy constructor, and it’s deleted if a move operation is declared. The copy assignment operator is generated only for classes lacking an explicitly declared copy assignment operator, and it’s deleted if a move operation is declared. Generation of the copy operations in classes with an explicitly declared destructor is deprecated.

• Member function templates never suppress generation of special member functions.*” move-enabled is important property of class.*

*p. 135*

## **CHAPTER 4 Smart Pointers**

*Six disadvantages of raw pointers: “*we might try to enumerate the reasons why a raw pointer is hard to love:

1. Its declaration doesn’t indicate whether it points to a single object or to an array. 2. Its declaration reveals nothing about whether you should destroy what it points to when you’re done using it, i.e., if the pointer owns the thing it points to.

3. If you determine that you should destroy what the pointer points to, there’s no way to tell how. Should you use delete, or is there a different destruction mechanism (e.g., a dedicated destruction function the pointer should be passed to)?

4. If you manage to find out that delete is the way to go, Reason 1 means it may not be possible to know whether to use the single-object form (“delete”) or the array form (“delete []”). If you use the wrong form, results are undefined.

5. Assuming you ascertain that the pointer owns what it points to and you discover how to destroy it, it’s difficult to ensure that you perform the destruction exactly once along every path in your code (including those due to exceptions). Missing a path leads to resource leaks, and doing the destruction more than once leads to undefined behavior.

6. There’s typically no way to tell if the pointer dangles, i.e., points to memory that no longer holds the object the pointer is supposed to point to. Dangling pointers arise when objects are destroyed while pointers still point to them.*” destruction mechanism – is concept for programming. Delete and dedicated destruction function are kinds. Intbu. 2 examples of undefined behavior. Example of resource leak: when you haven’t destroyed resource and it continue exesting and take memory. Definition of dangling pointers.*

*p. 136*

*argumentation of Mayers about raw pointers: “*Raw pointers are powerful tools, to be sure, but decades of experience have demonstrated that with only the slightest lapse in concentration or discipline, these tools can turn on their ostensible masters.*”*

Smart pointers…

*p. 136*

*23.12.2016*

*Smart pointers. They are pattern “wrapper”?: “*Smart pointers are one way to address these issues. Smart pointers are wrappers around raw pointers that act much like the raw pointers they wrap, but that avoid many of their pitfalls. You should therefore prefer smart pointers to raw pointers. Smart pointers can do virtually everything raw pointers can, but with far fewer opportunities for error.*” book recommendation and advantage of smart pointer comparing with raw pointers.*

*C++ structure: kinds of smart pointers: “*There are four smart pointers in C++11: std::auto\_ptr, std::unique\_ptr, std::shared\_ptr, and std::weak\_ptr. All are designed to help manage the lifetimes of dynamically allocated objects, i.e., to avoid resource leaks by ensuring that such objects are destroyed in the appropriate manner at the appropriate time (including in the event of exceptions).*” purposes of smart pointers. They are for resourse liaks avoidance.*

*C++ history: std::auto\_ptr is deprecated: “*std::auto\_ptr is a deprecated leftover from C++98. It was an attempt to standardize what later became C++11’s std::unique\_ptr. Doing the job right required move semantics, but C++98 didn’t have them. As a workaround, std::auto\_ptr co-opted its copy operations for moves. This led to surprising code (copying a std::auto\_ptr sets it to null!) and frustrating usage restrictions (e.g., it’s not possible to store std::auto\_ptrs in containers).*” example of very bad mechanism.*

*Std::unique\_ptr advantage and book recommendation. Using context of it and std::auto\_ptr: “*std::unique\_ptr does everything std::auto\_ptr does, plus more. It does it as efficiently, and it does it without warping what it means to copy an object. It’s better than std::auto\_ptr in every way. The only legitimate use case for std::auto\_ptr is a need to compile code with C++98 compilers. Unless you have that constraint, you should replace std::auto\_ptr with std::unique\_ptr and never look back.*” example of bad feature that is a result of compatibility need. Book recommendation.*

*C++ smart pointers structure: “*The smart pointer APIs are remarkably varied. About the only functionality common to all is default construction. Because comprehensive references for these APIs are widely available, I’ll focus my discussions on information that’s often missing from API overviews, e.g., noteworthy use cases, runtime cost analyses, etc. Mastering such information can be the difference between merely using these smart pointers and using them effectively.*” book structure. Intbu smart pointers APIs.*

### **Item 18: Use std::unique\_ptr for exclusive-ownership resource management**

*Using context of std::unique\_ptr. Frequency of their using: “*When you reach for a smart pointer, std::unique\_ptr should generally be the one closest at hand.*”*

*Std::unique\_ptr structure: “*It’s reasonable to assume that, by default, std::unique\_ptrs are the same size as raw pointers, and for most operations (including dereferencing), they execute exactly the same instructions.*”*

*Their using context: “*This means you can use them even in situations where memory and cycles are tight. If a raw pointer is small enough and fast enough for you, a std::unique\_ptr almost certainly is, too.*” what meand “cycles are tight”?*

*Comparing raw pointers and std::unique\_ptrs. Their properties: “*If a raw pointer is small enough and fast enough for you, a std::unique\_ptr almost certainly is, too.*”*

*p. 137*

*Programming idiom – exclusive membership semantics. Std::unque\_ptr structure and purpose: “*std::unique\_ptr embodies exclusive ownership semantics.*”*

*Std::unique\_ptr structure: “*A non-null std::unique\_ptr always owns what it points to. Moving a std::unique\_ptr transfers ownership from the source pointer to the destination pointer. (The source pointer is set to null.) Copying a std::unique\_ptr isn’t allowed, because if you could copy a std::unique\_ptr, you’d end up with two std::unique\_ptrs to the same resource, each thinking it owned (and should therefore destroy) that resource. std::unique\_ptr is thus a move-only type. Upon destruction, a non-null std::unique\_ptr destroys its resource. By default, resource destruction is accomplished by applying delete to the raw pointer inside the std::unique\_ptr.*” kind of type: move-only type.*

*Using context of ptr: to return type of classes in hierarchy: “*A common use for std::unique\_ptr is as a factory function return type for objects in a hierarchy.”

Example of it: “ Suppose we have a hierarchy for types of investments (e.g., stocks, bonds, real estate, etc.) with a base class Investment.

class Investment { … };

class Stock:

public Investment { … };

class Bond:

public Investment { … };

class RealEstate:

public Investment { … };

A factory function for such a hierarchy typically allocates an object on the heap and returns a pointer to it, with the caller being responsible for deleting the object when it’s no longer needed. That’s a perfect match for std::unique\_ptr, because the caller acquires responsibility for the resource returned by the factory (i.e., exclusive ownership of it), and the std::unique\_ptr automatically deletes what it points to when the std::unique\_ptr is destroyed. A factory function for the Investment hierarchy could be declared like this:

template<typename... Ts> // return std::unique\_ptr

std::unique\_ptr<Investment> // to an object created

makeInvestment(Ts&&... params); // from the given args

Callers could use the returned std::unique\_ptr in a single scope as follows,

{

…*”*

*p. 138*

*example continue: “*auto pInvestment = // pInvestment is of type

makeInvestment( arguments ); // std::unique\_ptr<Investment>

…

} // destroy \*pInvestment*”*

*Std::unique\_ptr using context. Owvership-migration scenarios – scenarios of program functioning: “*but they could also use it in ownership-migration scenarios, such as when the std::unique\_ptr returned from the factory is moved into a container, the container element is subsequently moved into a data member of an object, and that object is later destroyed. When that happens, the object’s std::unique\_ptr data member would also be destroyed, and its destruction would cause the resource returned from the factory to be destroyed. If the ownership chain got interrupted due to an exception or other atypical control flow (e.g., early function return or break from a loop), the std::unique\_ptr owning the managed resource would eventually have its destructor called,1 and the resource it was managing would thereby be destroyed.*”*

*Std::unique\_ptr structure: custom deleters: “*By default, that destruction would take place via delete, but, during construction, std::unique\_ptr objects can be configured to use custom deleters: arbitrary functions (or function objects, including those arising from lambda expressions) to be invoked when it’s time for their resources to be destroyed.*” Function objects, lambda expression using context. Intbu.*

*Example of it: “*If the object created by makeInvestment shouldn’t be directly deleted, but instead should first have a log entry written, makeInvestment could be implemented as follows. (An explanation follows the code, so don’t worry if you see something whose motivation is less than obvious.)

auto delInvmt = [](Investment\* pInvestment) // custom

{ // deleter

makeLogEntry(pInvestment); // (a lambda

delete pInvestment; // expression)

};

template<typename... Ts> // revised

std::unique\_ptr<Investment, decltype(delInvmt)> // return type

makeInvestment(Ts&&... params)

{

std::unique\_ptr<Investment, decltype(delInvmt)> // ptr to be

pInv(nullptr, delInvmt); // returned

if ( /\* a Stock object should be created \*/ )

{

pInv.reset(new Stock(std::forward<Ts>(params)...));

}

else if ( /\* a Bond object should be created \*/ )

{

pInv.reset(new Bond(std::forward<Ts>(params)...));

}

else if ( /\* a RealEstate object should be created \*/ )

{

pInv.reset(new RealEstate(std::forward<Ts>(params)...));

}

return pInv;

}*” intbu this code.*

*Nuances of std::unique\_ptr using: “*There are a few exceptions to this rule. Most stem from abnormal program termination. If an exception propagates out of a thread’s primary function (e.g., main, for the program’s initial thread) or if a noexcept specification is violate (see Item 14), local objects may not be destroyed, and if std::abort or an exit function (i.e., std::\_Exit, std::exit, or std::quick\_exit) is called, they definitely won’t be*” intbu these cases.*

*p. 139*

*advantage of std::unique\_ptr: “*In a moment, I’ll explain how this works, but first consider how things look if you’re a caller. Assuming you store the result of the makeInvestment call in an auto variable, you frolic in blissful ignorance of the fact that the resource you’re using requires special treatment during deletion. In fact, you veritably bathe in bliss, because the use of std::unique\_ptr means you need not concern yourself with when the resource should be destroyed, much less ensure that the destruction happens exactly once along every path through the program. std::unique\_ptr takes care of all those things automatically. From a client’s perspective, makeInvestment’s interface is sweet.*”*

*Example explanation: “*delInvmt is the custom deleter for the object returned from makeInvestment. All custom deletion functions accept a raw pointer to the object to be destroyed, then do what is necessary to destroy that object. In this case, the action is to call makeLogEntry and then apply delete. Using a lambda expression to create delInvmt is convenient, but, as we’ll see shortly, it’s also more efficient than writing a conventional function.*” lambda expression example and advantage. Intbu.*

*Example expression: “*When a custom deleter is to be used, its type must be specified as the second type argument to std::unique\_ptr. In this case, that’s the type of delInvmt, and that’s why the return type of makeInvestment is std::unique\_ptr<Investment, decltype(delInvmt)>. (For information about decltype, see Item 3.)*” book structure (Item 3). Intbu decltype using here.*

*Example explanation: “*The basic strategy of makeInvestment is to create a null std::unique\_ptr, make it point to an object of the appropriate type, and then return it. To associate the custom deleter delInvmt with pInv, we pass that as its second constructor argument.*” I understood example.*

*p. 140*

*example eplanation and C++ feature: prohibition of direct conversion of raw pointers to smart pointers: “*Attempting to assign a raw pointer (e.g., from new) to a std::unique\_ptr won’t compile, because it would constitute an implicit conversion from a raw to a smart pointer. Such implicit conversions can be problematic, so C++11’s smart pointers prohibit them. That’s why reset is used to have pInv assume ownership of the object created via new.*”*

*Intbu std::forward: “*With each use of new, we use std::forward to perfect-forward the arguments passed to makeInvestment (see Item 25). This makes all the information provided by callers available to the constructors of the objects being created.*” Book structure (Item 25).*

*Custom deleter structure and virtual destructor purpose: “*The custom deleter takes a parameter of type Investment\*. Regardless of the actual type of object created inside makeInvestment (i.e., Stock, Bond, or Real Estate), it will ultimately be deleted inside the lambda expression as an Investment\* object. This means we’ll be deleting a derived class object via a base class pointer. For that to work, the base class—Investment—must have a virtual destructor:

class Investment {

public:

… // essential

virtual ~Investment(); // design

… // component!

};*”*

*Function return type deduction: C++14 feature. Example structure: “*In C++14, the existence of function return type deduction (see Item 3) means that makeInvestment could be implemented in this simpler and more encapsulated fashion:

template<typename... Ts>

auto makeInvestment(Ts&&... params) // C++14

{

auto delInvmt = [](Investment\* pInvestment) // this is now

{ // inside

makeLogEntry(pInvestment); // make

delete pInvestment; // Investment

};

std::unique\_ptr<Investment, decltype(delInvmt)> // as

pInv(nullptr, delInvmt); // before

if ( … ) // as before

{

pInv.reset(new Stock(std::forward<Ts>(params)...));

}

else if ( … ) // as before

{

pInv.reset(new Bond(std::forward<Ts>(params)...));

}

else if ( … ) // as before

{

pInv.reset(new RealEstate(std::forward<Ts>(params)...));

}

return pInv; // as before

}

*” example of factory method.*

*p. 141*

I remarked earlier that…

*26.12.2016*

*Default deleter (delete keyword) is C++ feature.*

*Custom deleters influence on std::unique\_ptr and relation of last with raw pointers: “*I remarked earlier that, when using the default deleter (i.e., delete), you can reasonably assume that std::unique\_ptr objects are the same size as raw pointers. When custom deleters enter the picture, this may no longer be the case. Deleters that are function pointers generally cause the size of a std::unique\_ptr to grow from one word to two. For deleters that are function objects, the change in size depends on how much state is stored in the function object. Stateless function objects (e.g., from lambda expressions with no captures) incur no size penalty, and this means that when a custom deleter can be implemented as either a function or a captureless lambda expression, the lambda is preferable:*” advantage of captureless lambda functions over function objects in custom deleters. Intbu what is stateless lambda? Then there is example of it in book example.*

*Book example: “*auto delInvmt1 = [](Investment\* pInvestment) // custom

{ // deleter

makeLogEntry(pInvestment); // as

delete pInvestment; // stateless

}; // lambda

template<typename... Ts> // return type

std::unique\_ptr<Investment, decltype(delInvmt1)> // has size of

makeInvestment(Ts&&... args); // Investment\*

void delInvmt2(Investment\* pInvestment) // custom

{ // deleter

makeLogEntry(pInvestment); // as function

delete pInvestment;

}

template<typename... Ts> // return type has

std::unique\_ptr<Investment, // size of Investment\*

void (\*)(Investment\*)> // plus at least size

makeInvestment(Ts&&... params); // of function pointer!*” what is function object?*

*Nuance of unique\_ptr and function objects possible not good in some situations characteristic: “*Function object deleters with extensive state can yield std::unique\_ptr objects of std::unique\_ptrs unacceptably large, you probably need to change your design.*”*

*p. 142*

*using context of std::unique\_ptr: “*Factory functions are not the only common use case for std::unique\_ptrs. They’re even more popular as a mechanism for implementing the Pimpl Idiom. The code for that isn’t complicated, but in some cases it’s less than straightforward, so I’ll refer you to Item 22, which is dedicated to the topic.*” To contexts. Book structure (Item 23).*

*Forms of std::unique\_ptr: “*std::unique\_ptr comes in two forms, one for individual objects (std::unique\_ptr<T>) and one for arrays (std::unique\_ptr<T[]>). As a result, there’s never any ambiguity about what kind of entity a std::unique\_ptr points to. The std::unique\_ptr API is designed to match the form you’re using. For example, there’s no indexing operator (operator[]) for the single-object form, while the array form lacks dereferencing operators (operator\* and operator->).*” structure of these forms and std::unique\_ptr advantage.*

*Book recommendation. Using context of STL arrays and their kinds: “*The existence of std::unique\_ptr for arrays should be of only intellectual interest to you, because std::array, std::vector, and std::string are virtually always better data structure choices than raw arrays. About the only situation I can conceive of when a std::unique\_ptr<T[]> would make sense would be when you’re using a C-like API that returns a raw pointer to a heap array that you assume ownership of.*” other type of array are used only for compatibility.*

*Attractive feature of std::unique\_ptr. Convertability to std::shared\_ptr: “*std::unique\_ptr is the C++11 way to express exclusive ownership, but one of its most attractive features is that it easily and efficiently converts to a std::shared\_ptr:

std::shared\_ptr<Investment> sp = // converts std::unique\_ptr

makeInvestment( arguments ); // to std::shared\_ptr*” C++ type conversion: std::unique\_ptr to std::shared\_ptr.*

*Object kinds: with exclusive-ownership semantics and with shared ownership semantic. Factory function structure. Advantage of std::unique\_ptr: “*This is a key part of why std::unique\_ptr is so well suited as a factory function return type. Factory functions can’t know whether callers will want to use exclusiveownership semantics for the object they return or whether shared ownership (i.e., std::shared\_ptr) would be more appropriate. By returning a std::unique\_ptr, factories provide callers with the most efficient smart pointer, but they don’t hinder callers from replacing it with its more flexible sibling. (For information about std::shared\_ptr, proceed to Item 19.)*” Book structure (Item 19). std::shared\_ptr is more flexible kind of smart pointer.*

*Book structure. Resume: “*Things to Remember

• std::unique\_ptr is a small, fast, move-only smart pointer for managing resources with exclusive-ownership semantics.

• By default, resource destruction takes place via delete, but custom deleters can be specified. Stateful deleters and function pointers as deleters increase the size of std::unique\_ptr objects.

• Converting a std::unique\_ptr to a std::shared\_ptr is easy.*”*

*p. 143*

### **Item 19: Use std::shared\_ptr for shared-ownership resource management**

*Book recommendation. Programming idiom “shared-ownership resource management”.*

*Garbage collection designation: “*Programmers using languages with garbage collection point and laugh at what C++ programmers go through to prevent resource leaks. “How primitive!” they jeer. “Didn’t you get the memo from Lisp in the 1960s? Machines should manage resource lifetimes, not humans.” C++ developers roll their eyes. “You mean the memo where the only resource is memory and the timing of resource reclamation is nondeterministic? We prefer the generality and predictability of destructors, thank you.”*” To prevent problems with garbage collection leaks. C++ conception of memory governing.*

*C++ shared pointing idiom: “*But our bravado is part bluster. Garbage collection really is convenient, and manual lifetime management really can seem akin to constructing a mnemonic memory circuit using stone knives and bear skins. Why can’t we have the best of both worlds: a system that works automatically (like garbage collection), yet applies to all resources and has predictable timing (like destructors)?*” garbage collection disadvantage.*

*std::shared\_ptr structure and advantages: “*std::shared\_ptr is the C++11 way of binding these worlds together. An object accessed via std::shared\_ptrs has its lifetime managed by those pointers through shared ownership. No specific std::shared\_ptr owns the object. Instead, all std::shared\_ptrs pointing to it collaborate to ensure its destruction at the point where it’s no longer needed. When the last std::shared\_ptr pointing to an object stops pointing there (e.g., because the std::shared\_ptr is destroyed or made to point to a different object), that std::shared\_ptr destroys the object it points to. As with garbage collection, clients need not concern themselves with managing the lifetime of pointed-to objects, but as with destructors, the timing of the objects’ destruction is deterministic.*” garbage collection advantage – there is no necessity to control lifetime of object. Situations when pointer stop pointing object.*

*std::shared\_ptr structure. Reference count: “*A std::shared\_ptr can tell whether it’s the last one pointing to a resource by consulting the resource’s reference count, a value associated with the resource that keeps track of how many std::shared\_ptrs point to it. std::shared\_ptr constructors increment this count (usually—see below), std::shared\_ptr destructors decrement it, and copy assignment operators do both. (If sp1 and sp2 are std::shared\_ptrs to different objects, the assignment “sp1 = sp2;” modifies sp1 such that it points to the object pointed to by sp2. The net effect of the assignment is that the reference count for the object originally pointed to by sp1 is decremented, while that for the object pointed to by sp2 is incremented.) If a std::shared\_ptr sees a reference count of zero after performing a decrement, no more std::shared\_ptrs point to the resource, so the std::shared\_ptr destroys it.*”*

*Advantage of reference count: “*The existence of the reference count has performance implications:

*p. 144*

*comparison of std::shared\_ptr and raw pointer by size: “*std::shared\_ptrs are twice the size of a raw pointer, because they internally contain a raw pointer to the resource as well as a raw pointer to the resource’s reference count.*” First advantage.*

*“*Memory for the reference count must be dynamically allocated. Conceptually, the reference count is associated with the object being pointed to, but pointed-to objects know nothing about this. They thus have no place to store a reference count. (A pleasant implication is that any object—even those of built-in types— may be managed by std::shared\_ptrs.)… Either way, the reference count is stored as dynamically allocated data.*” the second advantage. Why is this advantage?*

*Purpose of std::make\_shared and its aspects: “*Item 21 explains that the cost of the dynamic allocation is avoided when the std::shared\_ptr is created by std::make\_shared, but there are situations where std::make\_shared can’t be used.*” Book structure (Item 21).*

*Very important third advantage of shared pointers. Concurrentness: “*Increments and decrements of the reference count must be atomic, because there can be simultaneous readers and writers in different threads. For example, a std::shared\_ptr pointing to a resource in one thread could be executing its destructor (hence decrementing the reference count for the resource it points to), while, in a different thread, a std::shared\_ptr to the same object could be copied (and therefore incrementing the same reference count).*”*

*Aspects of atomic opearations rate and std::shared\_ptr rate: “*Atomic operations are typically slower than non-atomic operations, so even though reference counts are usually only a word in size, you should assume that reading and writing them is comparatively costly.*”*

*std::shared\_ptr nuance: “*Did I pique your curiosity when I wrote that std::shared\_ptr constructors only “usually” increment the reference count for the object they point to? Creating a std::shared\_ptr pointing to an object always yields one more std::shared\_ptr pointing to that object, so why mustn’t we always increment the reference count? Move construction, that’s why. Move-constructing a std::shared\_ptr from another std::shared\_ptr sets the source std::shared\_ptr to null, and that means that the old std::shared\_ptr stops pointing to the resource at the moment the new std::shared\_ptr starts. As a result, no reference count manipulation is required.*” move semantics structure.*

*Moving operation speed and comparison with copy operations speed: “*Moving std::shared\_ptrs is therefore faster than copying them: copying requires incrementing the reference count, but moving doesn’t. This is as true for assignment as for construction, so move construction is faster than copy construction, and move assignment is faster than copy assignment.*”*

*std::shared\_ptr structure. Custom deleters. Difference from std::unique\_ptr: “*Like std::unique\_ptr (see Item 18), std::shared\_ptr uses delete as its default resource-destruction mechanism, but it also supports custom deleters. The design of this support differs from that for std::unique\_ptr, however.*” Book structure (Item 18)*

*p. 145*

*std::shared\_ptr structure: “*std::unique\_ptr, the type of the deleter is part of the type of the smart pointer. For std::shared\_ptr, it’s not:

auto loggingDel = [](Widget \*pw) // custom deleter

{ // (as in Item 18)

makeLogEntry(pw);

delete pw;

};

std::unique\_ptr< // deleter type is

Widget, decltype(loggingDel) // part of ptr type

> upw(new Widget, loggingDel);

std::shared\_ptr<Widget> // deleter type is not

spw(new Widget, loggingDel); // part of ptr type*”*

*std::shared\_ptr flexibility and lambda expressions using context: “*The std::shared\_ptr design is more flexible. Consider two std::shared\_ptr <Widget>s, each with a custom deleter of a different type (e.g., because the custom deleters are specified via lambda expressions):

auto customDeleter1 = [](Widget \*pw) { … }; // custom deleters,

auto customDeleter2 = [](Widget \*pw) { … }; // each with a

// different type

std::shared\_ptr<Widget> pw1(new Widget, customDeleter1);

std::shared\_ptr<Widget> pw2(new Widget, customDeleter2);

Because pw1 and pw2 have the same type, they can be placed in a container of objects of that type:

std::vector<std::shared\_ptr<Widget>> vpw{ pw1, pw2 };

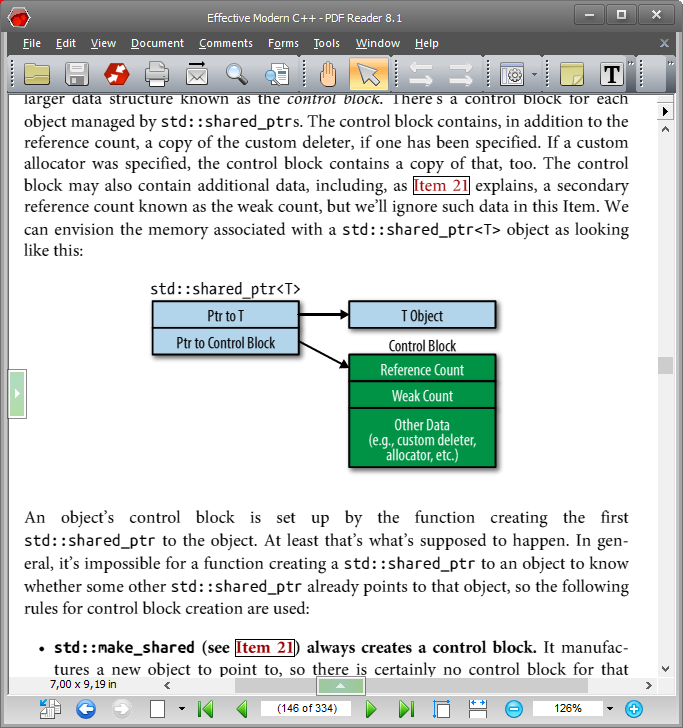
They could also be assigned to one another, and they could each be passed to a function taking a parameter of type std::shared\_ptr<Widget>. None of these things can be done with std::unique\_ptrs that differ in the types of their custom deleters, because the type of the custom deleter would affect the type of the std::unique\_ptr. In another difference from std::unique\_ptr, specifying a custom deleter doesn’t change the size of a std::shared\_ptr object. Regardless of deleter, a std::shared\_ptr object is two pointers in size. That’s great news, but it should make you vaguely uneasy.*” difference from std::unque\_ptr (2).*

*Book question: “*Custom deleters can be function objects, and function objects can contain arbitrary amounts of data. That means they can be arbitrarily large. How can a std::shared\_ptr refer to a deleter of arbitrary size without using any more memory?*”*

*p. 146*

*answer and std::shared\_ptr memory structure: “*It can’t. It may have to use more memory. However, that memory isn’t part of the std::shared\_ptr object. It’s on the heap or, if the creator of the std::shared\_ptr took advantage of std::shared\_ptr support for custom allocators, it’s wherever the memory managed by the allocator is located.*” What is custom allocator?*

*Memory allocation in std::shared\_ptr. Its structure: “*I remarked earlier that a std::shared\_ptr object contains a pointer to the reference count for the object it points to. That’s true, but it’s a bit misleading, because the reference count is part of a larger data structure known as the control block. There’s a control block for each object managed by std::shared\_ptrs. The control block contains, in addition to the reference count, a copy of he custom deleter, if one has been specified. If a custom allocator was specified, the control block contains a copy of that, too. The control block may also contain additional data, including, as Item 21 explains, a secondary reference count known as the weak count, but we’ll ignore such data in this Item. We can envision the memory associated with a std::shared\_ptr<T> object as looking

like this: 

*” book structure (Item 21).*

*std::shared\_ptr structure: “*An object’s control block is set up by the function creating the first std::shared\_ptr to the object. At least that’s what’s supposed to happen.*”*

*std::shared\_ptr structure: rules of control block creation: “*In general, it’s impossible for a function creating a std::shared\_ptr to an object to know whether some other std::shared\_ptr already points to that object, so the following rules for control block creation are used:

• std::make\_shared (see Item 21) always creates a control block. It manufactures a new object to point to, so there is certainly no control block for that object at the time std::make\_shared is called.

• A control block is created when a std::shared\_ptr is constructed from a unique-ownership pointer (i.e., a td::unique\_ptr or std::auto\_ptr).

Unique-ownership pointers don’t use control blocks, so there should be no control block for the pointed-to object. (As part of its construction, the std::shared\_ptr assumes ownership of the pointed-to object, so the uniqueownership pointer is set to null.) When a std::shared\_ptr constructor is called with a raw pointer, it creates a control block. If you wanted to create a std::shared\_ptr from an object that already had a control block, you’d presumably pass a std::shared\_ptr or a std::weak\_ptr (see Item 20) as a constructor argument, not a raw pointer. std::shared\_ptr constructors taking std::shared\_ptrs or std::weak\_ptrs as constructor arguments don’t create new control blocks, because they can rely on the smart pointers passed to them to point to any necessary control blocks.*” book structure (Item 21). Structure of std::make\_shared.*

*p. 147*

*C++ undefined behavior and std::shared\_ptr nuance: “*A consequence of these rules is that constructing more than one std::shared\_ptr from a single raw pointer gives you a complimentary ride on the particle accelerator of undefined behavior, because the pointed-to object will have multiple control blocks. Multiple control blocks means multiple reference counts, and multiple reference counts means the object will be destroyed multiple times (once for each reference count). That means that code like this is bad, bad, bad:

auto pw = new Widget; // pw is raw ptr

…

std::shared\_ptr<Widget> spw1(pw, loggingDel); // create control

// block for \*pw

…

std::shared\_ptr<Widget> spw2(pw, loggingDel); // create 2nd

// control block

// for \*pw!

The creation of the raw pointer pw to a dynamically allocated object is bad, because it runs contrary to the advice behind this entire chapter: to prefer smart pointers to raw pointers. (If you’ve forgotten the motivation for that advice, refresh your memory on page 115.) But set that aside. The line creating pw is a stylistic abomination, but at least it doesn’t cause undefined program behavior. Now, the constructor for spw1 is called with a raw pointer, so it creates a control block (and thereby a reference count) for what’s pointed to. In this case, that’s \*pw (i.e., the object pointed to by pw). In and of itself, that’s okay, but the constructor for spw2 is called with the same raw pointer, so it also creates a control block (hence a reference count) for \*pw. \*pw thus has two reference counts, each of which will eventually become zero, and that will ultimately lead to an attempt to destroy \*pw twice. The second destruction is responsible for the undefined behavior. *” book recommendation and book structure.*

*std::shared\_ptr using technique and std::make\_shared purpose: “*There are at least two lessons regarding std::shared\_ptr use here. First, try to avoid passing raw pointers to a std::shared\_ptr constructor. The usual alternative is to use std::make\_shared (see Item 21), but in the example above, we’re using custom deleters, and that’s not possible with std::make\_shared.*” book structure (Item 21).*

*p. 148*

*book recommendation: “*Second, if you must pass a raw pointer to a std::shared\_ptr constructor, pass the result of new directly instead of going through a raw pointer variable. If the first part of the code above were rewritten like this,

std::shared\_ptr<Widget> spw1(new Widget, // direct use of new

loggingDel);

it’d be a lot less tempting to create a second std::shared\_ptr from the same raw pointer. Instead, the author of the code creating spw2 would naturally use spw1 as an initialization argument (i.e., would call the std::shared\_ptr copy constructor), and that would pose no problem whatsoever:

std::shared\_ptr<Widget> spw2(spw1); // spw2 uses same

// control block as spw1*” Good C++ practice.*

*C++ concept – this pointer. It is dangerous in context of std::shared\_ptr. Example of this: “*An especially surprising way that using raw pointer variables as std::shared\_ptr constructor arguments can lead to multiple control blocks involves the this pointer. Suppose our program uses std::shared\_ptrs to manage Widget objects, and we have a data structure that keeps track of Widgets that have been processed:

std::vector<std::shared\_ptr<Widget>> processedWidgets;

Further suppose that Widget has a member function that does the processing:

class Widget {

public:

…

void process();

…

};

Here’s a reasonable-looking approach for Widget::process:

void Widget::process()

{

… // process the Widget

processedWidgets.emplace\_back(this); // add it to list of

} // processed Widgets;

// this is wrong!

The comment about this being wrong says it all—or at least most of it. (The part that’s wrong is the passing of this, not the use of emplace\_back. If you’re not familiar with emplace\_back, see Item 42.) This code will compile, but it’s passing a raw pointer (this) to a container of std::shared\_ptrs. The std::shared\_ptr thus constructed will create a new control block for the pointed-to Widget (\*this). That doesn’t sound harmful until you realize that if there are std::shared\_ptrs outside the member function that already point to that Widget, it’s game, set, and match for undefined behavior.*” book structure (Item 42). What is emplace\_back?*

*p. 149*

*std::enable\_shared\_from\_this purpose and structure: “*The std::shared\_ptr API includes a facility for just this kind of situation. It has probably the oddest of all names in the Standard C++ Library: std::enable\_shared\_from\_this. That’s a template for a base class you inherit from if you want a class managed by std::shared\_ptrs to be able to safely create a std::shared\_ptr from a this pointer. In our example, Widget would inherit from std::enable\_shared\_from\_this as follows:

class Widget: public std::enable\_shared\_from\_this<Widget> {

public:

…

void process();

…

};

As I said, std::enable\_shared\_from\_this is a base class template. Its type parameter is always the name of the class being derived, so Widget inherits from std::enable\_shared\_from\_this<Widget>. If the idea of a derived class inheriting from a base class templatized on the derived class makes your head hurt, try not to think about it. The code is completely legal, and the design pattern behind it is so well established, it has a standard name, albeit one that’s almost as odd as std::enable\_shared\_from\_this. The name is The Curiously Recurring Template Pattern (CRTP). If you’d like to learn more about it, unleash your search engine, because here we need to get back to std::enable\_shared\_from\_this.*” What is “The Curiously Recurring Template Pattern (CRTP)”? (“* [*https://en.wikipedia.org/wiki/Curiously\_recurring\_template\_pattern*](https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern) *”)*

std::enable\_shared\_from\_this defines a member function

*p. 149*

*27.12.2016*

*Structure and purpose of std::enable\_shared\_from\_this: “*std::enable\_shared\_from\_this defines a member function that creates a std::shared\_ptr to the current object, but it does it without duplicating control blocks. The member function is shared\_from\_this, and you use it in member functions whenever you want a std::shared\_ptr that points to the same object as the this pointer.*”*

*Example of this: “*Here’s a safe implementation of Widget::process:

void Widget::process()

{

// as before, process the Widget

…

// add std::shared\_ptr to current object to processedWidgets

processedWidgets.emplace\_back(shared\_from\_this());

}*”*

*p. 150*

*Structure of std::enable\_shared\_from\_this and C++ undefined behavior: “*Internally, shared\_from\_this looks up the control block for the current object, and it creates a new std::shared\_ptr that refers to that control block. The design relies on the current object having an associated control block. For that to be the case, there must be an existing std::shared\_ptr (e.g., one outside the member function calling shared\_from\_this) that points to the current object. If no such std::shared\_ptr exists (i.e., if the current object has no associated control block), behavior is undefined, although shared\_from\_this typically throws an exception.*” conditions of using this mechanism.*

*Pattern for providing existence of at least one shared pointer to object: “*To prevent clients from calling member functions that invoke shared\_from\_this before a std::shared\_ptr points to the object, classes inheriting from std::enable\_shared\_from\_this often declare their constructors private and have clients create objects by calling factory functions that return std::shared\_ptrs. Widget, for example, could look like this:

class Widget: public std::enable\_shared\_from\_this<Widget> {

public:

// factory function that perfect-forwards args

// to a private ctor

template<typename... Ts>

static std::shared\_ptr<Widget> create(Ts&&... params);

…

void process(); // as before

…

private:

… // ctors

};*” example of it.*

*Book structure. Example of cognitive orientation of reader: “*By now, you may only dimly recall that our discussion of control blocks was motivated by a desire to understand the costs associated with std::shared\_ptrs. Now that we understand how to avoid creating too many control blocks, let’s return to the original topic.*”. прыклад адступлення? Паразважаць.*

*std:shared\_ptr structure: “*A control block is typically only a few words in size, although custom deleters and allocators may make it larger. The usual control block implementation is more sophisticated than you might expect. It makes use of inheritance, and there’s even a virtual function. (It’s used to ensure that the pointed-to object is properly destroyed.) That means that using std::shared\_ptrs also incurs the cost of the machinery for the virtual function used by the control block.*” Their nuances concerning memory and performance issues.*

*std::shared\_ptr structure: “*Having read about dynamically allocated control blocks, arbitrarily large deleters and allocators, virtual function machinery, and atomic reference count manipulations, your enthusiasm for std::shared\_ptrs may have waned somewhat.*”*

*p. 151*

*Using context of* std::shared\_ptrs and its ordinary case: “But for the functionality they provide, std::shared\_ptrs exact a very reasonable cost. Under typical conditions, where the default deleter and default allocator are used and where the std::shared\_ptr is created by std::make\_shared, the control block is only about three words in size, and its allocation is essentially free. (It’s incorporated into the memory allocation for the object being pointed to. For details, see Item 21.) ” book structure (Item 21). std::shared\_ptrs structure and comparison with raw pointers: “Dereferencing a std::shared\_ptr is no more expensive than dereferencing a raw pointer. Performing an operation requiring a reference count manipulation (e.g., copy construction or copy assignment, destruction) entails one or two atomic operations, but these operations typically map to individual machine instructions, so although they may be expensive compared to non-atomic instructions, they’re still just single instructions. The virtual function machinery in the control block is generally used only once per object managed by std::shared\_ptrs: when the object is destroyed.” *what is virtual function machinery?*

*Advantave of std::shared\_ptr: “*In exchange for these rather modest costs, you get automatic lifetime management of dynamically allocated resources. Most of the time, using std::shared\_ptr is vastly preferable to trying to manage the lifetime of an object with shared ownership by hand.*”*

*How to make decision about std::shared\_ptr or std::unique\_ptr use: “*If you find yourself doubting whether you can afford use of std::shared\_ptr, reconsider whether you really need shared ownership. If exclusive ownership will do or even may do, std::unique\_ptr is a better choice. Its performance profile is close to that for raw pointers, and “upgrading” from std::unique\_ptr to std::shared\_ptr is easy, because a std::shared\_ptr can be created from a std::unique\_ptr. The reverse is not true. Once you’ve turned lifetime management of a resource over to a std::shared\_ptr, there’s no changing your mind. Even if the reference count is one, you can’t reclaim ownership of the resource in order to, say, have a std::unique\_ptr manage it. The ownership contract between a resource and the std::shared\_ptrs that point to it is of the ’til-death-do-us-part variety. No divorce, no annulment, no dispensations.*” std::shared\_ptr is not convertible to other pointer. This is its nuance. For explanation of concept contract in programming.*

*Using contact of std::shared\_ptr. Their limitation. Arrays: “*Something else std::shared\_ptrs can’t do is work with arrays. In yet another difference from std::unique\_ptr, std::shared\_ptr has an API that’s designed only for pointers to single objects. There’s no std::shared\_ptr<T[]>. From time to time, “clever” programmers stumble on the idea of using a std::shared\_ptr<T> to point to an array, specifying a custom deleter to perform an array delete (i.e., delete []). This can be made to compile, but it’s a horrible idea. For one thing, std::shared\_ptr offers no operator[], so indexing into the array requires awkward expressions based on pointer arithmetic. For another, std::shared\_ptr supports derived-to-base pointer conversions that make sense for single objects, but that open holes in the type system when applied to arrays. (For this reason, the std::unique\_ptr<T[]> API prohibits such conversions.) Most importantly, given the variety of C++11 alternatives to built-in arrays (e.g., std::array, std::vector, std::string), declaring a smart pointer to a dumb array is almost always a sign of bad design.*” C++ type system features: derived-to-base pointer conversion. it doesn’t work for arrays. How can it work for arrays? I can’t imagine. Intbu. Bad C++ practice. Programming field – pointer arithmetics. C++ bad practice.*

*p. 152*

*book structure. Resume: “*Things to Remember

• std::shared\_ptrs offer convenience approaching that of garbage collection for the shared lifetime management of arbitrary resources.

• Compared to std::unique\_ptr, std::shared\_ptr objects are typically twice as big, incur overhead for control blocks, and require atomic reference count manipulations.

• Default resource destruction is via delete, but custom deleters are supported. The type of the deleter has no effect on the type of the std::shared\_ptr.

• Avoid creating std::shared\_ptrs from variables of raw pointer type.*”*

### **Item 20: Use std::weak\_ptr for std::shared\_ptr-like pointers that can dangle**

*Using context of std::weak\_ptr and book recommendation.*

*Structure and motivation of std::weak\_ptr: “*Paradoxically, it can be convenient to have a smart pointer that acts like a std::shared\_ptr (see Item 19), but that doesn’t participate in the shared ownership of the pointed-to resource. In other words, a pointer like std::shared\_ptr that doesn’t affect an object’s reference count. This kind of smart pointer has to contend with a problem unknown to std::shared\_ptrs: the possibility that what it points to has been destroyed. A truly smart pointer would deal with this problem by tracking when it dangles, i.e., when the object it is supposed to point to no longer exists. That’s precisely the kind of smart pointer std::weak\_ptr is.*” definition of pointer dangling. Book structure (Item 19).*

*Structure of std::weak\_ptr: “*You may be wondering how a std::weak\_ptr could be useful. You’ll probably wonder even more when you examine the std::weak\_ptr API. It looks anything but smart. std::weak\_ptrs can’t be dereferenced, nor can they be tested for nullness. That’s because std::weak\_ptr isn’t a standalone smart pointer. It’s an augmentation of std::shared\_ptr.*” Relation with std::shared\_ptr.*

*Structure and using technique of std::weak\_ptr: “*The relationship begins at birth. std::weak\_ptrs are typically created from std::shared\_ptrs. They point to the same place as the std::shared\_ptrs initializing them, but they don’t affect the reference count of the object they point to:

auto spw = // after spw is constructed,

std::make\_shared<Widget>(); // the pointed-to Widget's

// ref count (RC) is 1. (See

// Item 21 for info on

// std::make\_shared.)

…

std::weak\_ptr<Widget> wpw(spw); // wpw points to same Widget

// as spw. RC remains 1

…

spw = nullptr; // RC goes to 0, and the

// Widget is destroyed.

// wpw now dangles*”*

*p. 153*

*std:weak\_ptr structure: testing if dangles: “*std::weak\_ptrs that dangle are said to have expired. You can test for this directly,

if (wpw.expired()) … // if wpw doesn't point

// to an object…*”.*

*Using context of std::weak\_ptr: “*but often what you desire is a check to see if a std::weak\_ptr has expired and, if it hasn’t (i.e., if it’s not dangling), to access the object it points to.*”*

*Nuances of std::weak\_ptr: “*This is easier desired than done. Because std::weak\_ptrs lack dereferencing operations, there’s no way to write the code.*” What are dereferencing operations? Intbu.*

*C++ undefined behavior example: “*Even if there were, separating the check and the dereference would introduce a race condition: between the call to expired and the dereferencing action, another thread might reassign or destroy the last std::shared\_ptr pointing to the object, thus causing that object to be destroyed. In that case, your dereference would yield undefined behavior.*” nuance of std::weak\_ptr.*

*C++ type conversion system. 2 mechanisms of conversion of weak\_ptr to shared\_ptr: “*What you need is an atomic operation that checks to see if the std::weak\_ptr has expired and, if not, gives you access to the object it points to. This is done by creating a std::shared\_ptr from the std::weak\_ptr. The operation comes in two forms, depending on what you’d like to have happen if the std::weak\_ptr has expired when you try to create a std::shared\_ptr from it. One form is std::weak\_ptr::lock, which returns a std::shared\_ptr. The std::shared\_ptr is null if the std::weak\_ptr has expired:

std::shared\_ptr<Widget> spw1 = wpw.lock(); // if wpw's expired,

// spw1 is null

auto spw2 = wpw.lock(); // same as above,

// but uses auto

The other form is the std::shared\_ptr constructor taking a std::weak\_ptr as an argument. In this case, if the std::weak\_ptr has expired, an exception is thrown:

std::shared\_ptr<Widget> spw3(wpw); // if wpw's expired,

// throw std::bad\_weak\_ptr*”*

*p. 154*

*Author level. This question is true fact: ”* But you’re probably still wondering about how std::weak\_ptrs can be useful.*”*

*Using context with example: “*Consider a factory function that produces smart pointers to read-only objects based on a unique ID. In accord with Item 18’s advice regarding factory function return types, it returns a std::unique\_ptr:

std::unique\_ptr<const Widget> loadWidget(WidgetID id);

If loadWidget is an expensive call (e.g., because it performs file or database I/O) and it’s common for IDs to be used repeatedly, a reasonable optimization would be to write a function that does what loadWidget does, but also caches its results. Clogging the cache with every Widget that has ever been requested can lead to performance problems of its own, however, so another reasonable optimization would be to destroy cached Widgets when they’re no longer in use. For this caching factory function, a std::unique\_ptr return type is not a good fit. Callers should certainly receive smart pointers to cached objects, and callers should certainly determine the lifetime of those objects, but the cache needs a pointer to the objects, too. The cache’s pointers need to be able to detect when they dangle, because when factory clients are finished using an object returned by the factory, that object will be destroyed, and the corresponding cache entry will dangle. The cached pointers should therefore be std::weak\_ptrs—pointers that can detect when they dangle. That means that the factory’s return type should be a std::shared\_ptr, because std::weak\_ptrs can detect when they dangle only when an object’s lifetime is managed by std::shared\_ptrs.*” intbu. It’s not so obvious for me. They are necessary for caching?*

*Example of it: “*Here’s a quick-and-dirty implementation of a caching version of loadWidget:

std::shared\_ptr<const Widget> fastLoadWidget(WidgetID id)

{

static std::unordered\_map<WidgetID,

std::weak\_ptr<const Widget>> cache;

auto objPtr = cache[id].lock(); // objPtr is std::shared\_ptr

// to cached object (or null

// if object's not in cache)

if (!objPtr) { // if not in cache,

objPtr = loadWidget(id); // load it

cache[id] = objPtr; // cache it

}

return objPtr;

}*” I understood this example. Why can’t we test std::unique\_ptr or shared\_ptr on null directly? Why is std::weak\_ptr better? Intbu.*

*p. 155*

*C++ hash table container: std::unordered\_map.*

*Intbu the following example description: “*This implementation employs one of C++11’s hash table containers (std::unordered\_map), though it doesn’t show the WidgetID hashing and equality-comparison functions that would also have to be present.*”*

*Example improvement: “*The implementation of fastLoadWidget ignores the fact that the cache may accumulate expired std::weak\_ptrs corresponding to Widgets that are no longer in use (and have therefore been destroyed). The implementation can be refined, but rather than spend time on an issue that lends no additional insight into std::weak\_ptrs, let’s consider a second use case: the Observer design pattern.*”*

*Using context of std::weak\_ptr. Pattern Observer: “*let’s consider a second use case: the Observer design pattern. The primary components of this pattern are subjects (objects whose state may change) and observers (objects to be notified when state changes occur). In most implementations, each subject contains a data member holding pointers to its observers. That makes it easy for subjects to issue state change notifications. Subjects have no interest in controlling the lifetime of their observers (i.e., when they’re destroyed), but they have a great interest in making sure that if an observer gets destroyed, subjects don’t try to subsequently access it. A reasonable design is for each subject to hold a container of std::weak\_ptrs to its observers, thus making it possible for the subject to determine whether a pointer dangles before using it.*” I understood this usecase. Observer pattern implementation.*

*Using context of std::weak\_ptr and std::shared\_ptr too: “*As a final example of std::weak\_ptr’s utility, consider a data structure with objects A, B, and C in it, where A and C share ownership of B and therefore hold std::shared\_ptrs to it:

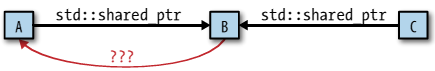
**

Suppose it’d be useful to also have a pointer from

B

back to A. What kind of pointer

should this be?

**

There are three choices:

• A raw pointer. With this approach, if A is destroyed, but C continues to point to B, B will contain a pointer to A that will dangle. B won’t be able to detect that, so B may inadvertently dereference the dangling pointer. That would yield undefined behavior.

• A std::shared\_ptr. In this design, A and B contain std::shared\_ptrs to each other. The resulting std::shared\_ptr cycle (A points to B and B points to A) will prevent both A and B from being destroyed. Even if A and B are unreachable from other program data structures (e.g., because C no longer points to B), each will have a reference count of one. If that happens, A and B will have been leaked, for all practical purposes: it will be impossible for the program to access them, yet their resources will never be reclaimed. A std::weak\_ptr. This avoids both problems above. If A is destroyed, B’s pointer back to it will dangle, but B will be able to detect that. Furthermore, though A and B will point to one another, B’s pointer won’t affect A’s reference count, hence can’t keep A from being destroyed when std::shared\_ptrs no longer point to it.*” second case is nuance of std::shared\_ptr. It can be destroyed only if reference count will be zero.*

*p. 156*

*frequency of the last example in practice: “*Using std::weak\_ptr is clearly the best of these choices. However, it’s worth noting that the need to employ std::weak\_ptrs to break prospective cycles of std::shared\_ptrs is not terribly common.*”*

*using context of std::unique\_ptr and raw pointers. Pointers in hierarchical data structures: ”* In strictly hierarchal data structures such as trees, child nodes are typically owned only by their parents. When a parent node is destroyed, its child nodes should be destroyed, too. Links from parents to children are thus generally best represented by std::unique\_ptrs. Back-links from children to parents can be safely implemented as raw pointers, because a child node should never have a lifetime longer than its parent. There’s thus no risk of a child node dereferencing a dangling parent pointer.*”*

*std::weak\_ptr using context: “*Of course, not all pointer-based data structures are strictly hierarchical, and when that’s the case, as well as in situations such as caching and the implementation of lists of observers, it’s nice to know thatstd::weak\_ptr stands at the ready.*”*

*Structure of std::weak\_ptr. Reference count realization: “*From an efficiency perspective, the std::weak\_ptr story is essentially the same as that for std::shared\_ptr. std::weak\_ptr objects are the same size as std::shared\_ptr objects, they make use of the same control blocks as std::shared\_ptrs (see Item 19), and operations such as construction, destruction, and assignment involve atomic reference count manipulations. That probably surprises you, because I wrote at the beginning of this Item that std::weak\_ptrs don’t participate in reference counting. Except that’s not quite what I wrote. What I wrote was that std::weak\_ptrs don’t participate in the shared ownership of objects and hence don’t affect the pointed-to object’s reference count. There’s actually a second reference count in the control block, and it’s this second reference count that std::weak\_ptrs manipulate. For details, continue on to Item 21.*” book structure (Item 21, Item 19). Performance and memory characteristics of std::weak\_ptr. Control block strucuture.*

*p. 157*

*Book structure. Resume: “*Things to Remember

• Use std::weak\_ptr for std::shared\_ptr-like pointers that can dangle.

• Potential use cases for std::weak\_ptr include caching, observer lists, and the prevention of std::shared\_ptr cycles.*” use cases of std::weak\_ptr.*

### **Item 21: Prefer std::make\_unique and std::make\_shared to direct use of new**

*C++ concepts and book recommendation. Their using context.*

*С++ history and std::make\_unique structure and emulation: “*Let’s begin by leveling the playing field for std::make\_unique and std::make\_shared. std::make\_shared is part of C++11, but, sadly, std::make\_unique isn’t. It joined the Standard Library as of C++14. If you’re using C++11, never fear, because a basic version of std::make\_unique is easy to write yourself. Here, look:

template<typename T, typename... Ts>

std::unique\_ptr<T> make\_unique(Ts&&... params)

{

return std::unique\_ptr<T>(new T(std::forward<Ts>(params)...));

}

As you can see, make\_unique just perfect-forwards its parameters to the constructor of the object being created, constructs a std::unique\_ptr from the raw pointer new produces, and returns the std::unique\_ptr so created. This form of the function doesn’t support arrays or custom deleters (see Item 18), but it demonstrates that with only a little effort, you can create make\_unique if you need to.3 Just remember not to put your version in namespace std, because you won’t want it to clash with a vendorprovided version when you upgrade to a C++14 Standard Library implementation.*” reference to more functional realization is given on this page.*

*C++ structure and make function idiom definition: “*std::make\_unique and std::make\_shared are two of the three make functions: functions that take an arbitrary set of arguments, perfect-forward them to the constructor for a dynamically allocated object, and return a smart pointer to that object. The third make function is std::allocate\_shared. It acts just like std::make\_shared, except its first argument is an allocator object to be used for the dynamic memory allocation.*” std::allocate\_shared intbu. What is allocator object? Intbu.*

*p. 158*

*advantages of make functions: “*Even the most trivial comparison of smart pointer creation using and not using a make function reveals the first reason why using such functions is preferable. Consider:

auto upw1(std::make\_unique<Widget>()); // with make func

std::unique\_ptr<Widget> upw2(new Widget); // without make func

auto spw1(std::make\_shared<Widget>()); // with make func

std::shared\_ptr<Widget> spw2(new Widget); // without make func

I’ve highlighted the essential difference: the versions using new repeat the type being created, but the make functions don’t. Repeating types runs afoul of a key tenet of software engineering: code duplication should be avoided. Duplication in source code increases compilation times, can lead to bloated object code, and generally renders a code base more difficult to work with. It often evolves into inconsistent code, and inconsistency in a code base often leads to bugs. Besides, typing something twice takes more effort than typing it once, and who’s not a fan of reducing their typing burden?*” and the common software design rule. Intbu articulated above problems concerning inconsistency.*

*Advantage of make functions – exception safety: “*The second reason to prefer make functions has to do with exception safety. Suppose we have a function to process a Widget in accord with some priority:

void processWidget(std::shared\_ptr<Widget> spw, int priority);

Passing the std::shared\_ptr by value may look suspicious, but Item 41 explains that if processWidget always makes a copy of the std::shared\_ptr (e.g., by storing it in a data structure tracking Widgets that have been processed), this can be a reasonable design choice. Now suppose we have a function to compute the relevant priority,

int computePriority();

and we use that in a call to processWidget that uses new instead of std::make\_shared:

processWidget(std::shared\_ptr<Widget>(new Widget), // potential

computePriority()); // resource

// leak!

As the comment indicates, this code could leak the Widget conjured up by new. But how? Both the calling code and the called function are using std::shared\_ptrs, and std::shared\_ptrs are designed to prevent resource leaks. They automatically destroy what they point to when the last std::shared\_ptr pointing there goes away. If everybody is using std::shared\_ptrs everywhere, how can this code leak?*” the first example concerns Item 41. Intbu it.*

*p. 159*

*example continue and nuance of std::shared\_ptr and compiler implementation. Source code, object code. What are they? C++ realization of function in runtime: “*The answer has to do with compilers’ translation of source code into object code. At runtime, the arguments for a function must be evaluated before the function can be invoked, so in the call to processWidget, the following things must occur before processWidget can begin execution:

• The expression “new Widget” must be evaluated, i.e., a Widget must be created on the heap.

• The constructor for the std::shared\_ptr<Widget> responsible for managing the pointer produced by new must be executed.

• computePriority must run.

Compilers are not required to generate code that executes them in this order. “new Widget” must be executed before the std::shared\_ptr constructor may be called, because the result of that new is used as an argument to that constructor, but compute Priority may be executed before those calls, after them, or, crucially, between them. That is, compilers may emit code to execute the operations in this order:

1. Perform “new Widget”.

2. Execute computePriority.

3. Run std::shared\_ptr constructor.

If such code is generated and, at runtime, computePriority produces an exception, the dynamically allocated Widget from Step 1 will be leaked, because it will never be stored in the std::shared\_ptr that’s supposed to start managing it in Step 3.*” compiler nuances. Code in function arguments list is of undefined performing order? Function implementation nuances. Very-very thin nuance…*

*Advantage of make functions: “*At runtime, either std::make\_shared or computePriority will be called first. If it’s std::make\_shared, the raw pointer to the dynamically allocated Widget is safely stored in the returned std::shared\_ptr before computePriority is called. If computePriority then yields an exception, the std::shared\_ptr destructor will see to it that the Widget it owns is destroyed. And if computePriority is called first and yields an exception, std::make\_shared will not be invoked, and there will hence be no dynamically allocated Widget to worry about.*”*

If we replace…

*p. 160*

*28.12.2016*

*Purpose of std::unique\_ptr. What is exception-safe code?: “*If we replace std::shared\_ptr and std::make\_shared with std::unique\_ptr and std::make\_unique, exactly the same reasoning applies. Using std::make\_unique instead of new is thus just as important in writing exception-safe code as using std::make\_shared.*”*

*Advantage of std::make\_shared: “*A special feature of std::make\_shared (compared to direct use of new) is improved efficiency. Using std::make\_shared allows compilers to generate smaller, faster code that employs leaner data structures. Consider the following direct use of new:

std::shared\_ptr<Widget> spw(new Widget);

It’s obvious that this code entails a memory allocation, but it actually performs two. Item 19 explains that every std::shared\_ptr points to a control block containing, among other things, the reference count for the pointed-to object. Memory for this control block is allocated in the std::shared\_ptr constructor. Direct use of new, then, requires one memory allocation for the Widget and a second allocation for the control block. If std::make\_shared is used instead,

auto spw = std::make\_shared<Widget>();

one allocation suffices. That’s because std::make\_shared allocates a single chunk of memory to hold both the Widget object and the control block.*” Realization of std::make\_shared.*

*What is static size of program? Intbu: “*This optimization reduces the static size of the program, because the code contains only one memory allocation call, and it increases the speed of the executable code, because memory is allocated only once. Furthermore, using std::make\_shared obviates the need for some of the bookkeeping information in the control block, potentially reducing the total memory footprint for the program.*”*

*Performance advantages of std::allocate\_shared: “*The efficiency analysis for std::make\_shared is equally applicable to std::allocate\_shared, so the performance advantages of std::make\_shared extend to that function, as well.*”. Intbu this C++ mechanism.*

*Three advantages of make functions. What is exception safety and how does it work here?: “*The arguments for preferring make functions over direct use of new are strong ones. Despite their software engineering, exception safety, and efficiency advantages,*”*

*Using context of make functions: “*however, this Item’s guidance is to prefer the make functions, not to rely on them exclusively. That’s because there are circumstances where they can’t or shouldn’t be used.*”*

*First limitation of make function using: custom deleter “*For example, none of the make functions permit the specification of custom deleters (see Items 18 and 19), but both std::unique\_ptr and std::shared\_ptr have constructors that do. Given a custom deleter for a Widget,

auto widgetDeleter = [](Widget\* pw) { … };

creating a smart pointer using it is straightforward using new:

std::unique\_ptr<Widget, decltype(widgetDeleter)>

upw(new Widget, widgetDeleter);

std::shared\_ptr<Widget> spw(new Widget, widgetDeleter);

There’s no way to do the same thing with a make function.*”book structure (Item 18, Item 19)*

*p. 161*

*second limitation of make function using: braced initializer: “*A second limitation of make functions stems from a syntactic detail of their implementations. Item 7 explains that when creating an object whose type overloads constructors both with and without std::initializer\_list parameters, creating the object using braces prefers the std::initializer\_list constructor, while creating the object using parentheses calls the non-std::initializer\_list constructor. The make functions perfect-forward their parameters to an object’s constructor, but do they do so using parentheses or using braces? For some types, the answer to this question makes a big difference. For example, in these calls,

auto upv = std::make\_unique<std::vector<int>>(10, 20);

auto spv = std::make\_shared<std::vector<int>>(10, 20);

do the resulting smart pointers point to std::vectors with 10 elements, each of value 20, or to std::vectors with two elements, one with value 10 and the other with value 20? Or is the result indeterminate? The good news is that it’s not indeterminate: both calls create std::vectors of size 10 with all values set to 20. That means that within the make functions, the perfect forwarding code uses parentheses, not braces. The bad news is that if you want to construct your pointed-to object using a braced initializer, you must use new directly.*” structure of make functions. Book structure (Item 7).*

*Braced-initializer structure: “*Using a make function would require the ability to perfect-forward a braced initializer, but, as Item 30 explains, braced initializers can’t be perfect-forwarded. However, Item 30 also describes a workaround: use auto type deduction to create a std::initializer\_list object from a braced initializer (see Item 2), then pass the autocreated object through the make function:

// create std::initializer\_list

auto initList = { 10, 20 };

// create std::vector using std::initializer\_list ctor

auto spv = std::make\_shared<std::vector<int>>(initList);*” book structure (Item 30). What is perfect-forwarding?*

*Way to solve limitation with braced initializer: “*However, Item 30 also describes a workaround: use auto type deduction to create a std::initializer\_list object from a braced initializer (see Item 2), then pass the autocreated object through the make function:

// create std::initializer\_list

auto initList = { 10, 20 };

// create std::vector using std::initializer\_list ctor

auto spv = std::make\_shared<std::vector<int>>(initList);*” C++ hack.*

*std::unique\_ptr limitations: “*For std::unique\_ptr, these two scenarios (custom deleters and braced initializers) are the only ones where its make functions are problematic.*”*

*std::shared\_ptr limitations: “*For std::shared\_ptr and its make functions, there are two more. Both are edge cases, but some developers live on the edge, and you may be one of them.*”*

*p. 162*

*limitation of make functions: create objects of types with class-specific versions of operator new and operator delete: “*Some classes define their own versions of operator new and operator delete. The presence of these functions implies that the global memory allocation and deallocation routines for objects of these types are inappropriate. Often, class-specific routines are designed only to allocate and deallocate chunks of memory of precisely the size of objects of the class, e.g., operator new and operator delete for class Widget are often designed only to handle allocation and deallocation of chunks of memory of exactly size sizeof(Widget). Such routines are a poor fit for std::shared\_ptr’s support for custom allocation (via std::allocate\_shared) and deallocation (via custom deleters), because the amount of memory that std::allocate\_shared requests isn’t the size of the dynamically allocated object, it’s the size of that object plus the size of a control block. Consequently, using make functions to create objects of types with class-specific versions of operator new and operator delete is typically a poor idea.*” C++ feature: custom allocation and deallocation. Intbu this and using of std::allocate\_shared.*

*Structure of std::make\_shared: “*The size and speed advantages of std::make\_shared vis-à-vis direct use of new stem from std::shared\_ptr’s control block being placed in the same chunk of memory as the managed object. When that object’s reference count goes to zero, the object is destroyed (i.e., its destructor is called). However, the memory it occupies can’t be released until the control block has also been destroyed, because the same chunk of dynamically allocated memory contains both.*”*

*Structure of control block: weak count: “*As I noted, the control block contains bookkeeping information beyond just the reference count itself. The reference count tracks how many std::shared\_ptrs refer to the control block, but the control block contains a second reference count, one that tallies how many std::weak\_ptrs refer to the control block. This second reference count is known as the weak count.4When a std::weak\_ptr checks to see if it has expired (see Item 19), it does so by examining the reference count (not the weak count) in the control block that it refers to. If the reference count is zero (i.e., if the pointed-to object has no std::shared\_ptrs referring to it and has thus been destroyed), the std::weak\_ptr has expired. Otherwise, it hasn’t.*” structure of std::weak\_ptr.*

*Intbu this control block nuance: “*In practice, the value of the weak count isn’t always equal to the number of std::weak\_ptrs referring to the control block, because library implementers have found ways to slip additional information into the weak count that facilitate better code generation. For purposes of this Item, we’ll ignore this and assume that the weak count’s value is the number of std::weak\_ptrs referring to the control block.*”*

*Nuance of std:: shared\_ptr. Is it potential memory leakage?: “*As long as std::weak\_ptrs refer to a control block (i.e., the weak count is greater than zero), that control block must continue to exist. And as long as a control block exists, the memory containing it must remain allocated. The memory allocated by a std::shared\_ptr make function, then, can’t be deallocated until the last std::shared\_ptr and the last std::weak\_ptr referring to it have been destroyed.*”*

*p. 163*

*std::shared\_ptr nuance. Example of this potential memory leakage: “*If the object type is quite large and the time between destruction of the last std::shared\_ptr and the last std::weak\_ptr is significant, a lag can occur between when an object is destroyed and when the memory it occupied is freed:

class ReallyBigType { … };

auto pBigObj = // create very large

std::make\_shared<ReallyBigType>(); // object via

// std::make\_shared

… // create std::shared\_ptrs and std::weak\_ptrs to

// large object, use them to work with it

… // final std::shared\_ptr to object destroyed here,

// but std::weak\_ptrs to it remain

… // during this period, memory formerly occupied

// by large object remains allocated

… // final std::weak\_ptr to object destroyed here;

// memory for control block and object is released

With a direct use of new, the memory for the ReallyBigType object can be released as soon as the last std::shared\_ptr to it is destroyed:

class ReallyBigType { … }; // as before

std::shared\_ptr<ReallyBigType> pBigObj(new ReallyBigType);

// create very large

// object via new

… // as before, create std::shared\_ptrs and

// std::weak\_ptrs to object, use them with it

… // final std::shared\_ptr to object destroyed here,

// but std::weak\_ptrs to it remain;

// memory for object is deallocated

… // during this period, only memory for the

// control block remains allocated

… // final std::weak\_ptr to object destroyed here;

// memory for control block is released*” andvantage of use new over using std::make\_shared.*

*p. 164*

*Using technique of new with smart pointers: “*Should you find yourself in a situation where use of std::make\_shared is impossible or inappropriate, you’ll want to guard yourself against the kind of exception-safety problems we saw earlier. The best way to do that is to make sure that when you use new directly, you immediately pass the result to a smart pointer constructor in a statement that does nothing else. This prevents compilers from generating code that could emit an exception between the use of new and invocation of the constructor for the smart pointer that will manage the newed object.*” about exceptional safety.*

*Remind of example with memory leakage: “*As an example, consider a minor revision to the exception-unsafe call to the process Widget function we examined earlier. This time, we’ll specify a custom deleter:

void processWidget(std::shared\_ptr<Widget> spw, // as before

int priority);

void cusDel(Widget \*ptr); // custom

// deleter

Here’s the exception-unsafe call:

processWidget( // as before,

std::shared\_ptr<Widget>(new Widget, cusDel), // potential

computePriority() // resource

); // leak!

Recall: if computePriority is called after “new Widget” but before the std::shared\_ptr constructor, and if computePriority yields an exception, the dynamically allocated Widget will be leaked.*”*

*C++ good programming practice in this situation: “*Here the use of a custom deleter precludes use of std::make\_shared, so the way to avoid the problem is to put the allocation of the Widget and the construction of the std::shared\_ptr into their own statement, then call processWidget with the resulting std::shared\_ptr. Here’s the essence of the technique, though, as we’ll see in a moment, we can tweak it to improve its performance:

std::shared\_ptr<Widget> spw(new Widget, cusDel);

processWidget(spw, computePriority()); // correct, but not

// optimal; see below*”*

*std::shared\_ptr structure. Intbu what will be in other cases and what other cases are?: “*This works, because a std::shared\_ptr assumes ownership of the raw pointer passed to its constructor, even if that constructor yields an exception. In this example, if spw’s constructor throws an exception (e.g., due to an inability to dynamically allocate memory for a control block), it’s still guaranteed that cusDel will be invoked on the pointer resulting from “new Widget”.*”*

*p. 165*

*Intbu this example: “*The minor performance hitch is that in the exception-unsafe call, we’re passing an rvalue to processWidget,

processWidget(

std::shared\_ptr<Widget>(new Widget, cusDel), // arg is rvalue

computePriority()

);

but in the exception-safe call, we’re passing an lvalue:

processWidget(spw, computePriority()); // arg is lvalue*” example of difference between rvalue and lvalue.*

*Std::move purpose. It make rvalue from lvalue: “*Because processWidget’s std::shared\_ptr parameter is passed by value, construction from an rvalue entails only a move, while construction from an lvalue requires a copy. For std::shared\_ptr, the difference can be significant, because copying a std::shared\_ptr requires an atomic increment of its reference count, while moving a std::shared\_ptr requires no reference count manipulation at all. For the exception-safe code to achieve the level of performance of the exception-unsafe code, we need to apply std::move to spw

to turn it into an rvalue (see Item 23):

processWidget(std::move(spw), // both efficient and

computePriority()); // exception safe*”*

*Structure of make functions using context: “*That’s interesting and worth knowing, but it’s also typically irrelevant, because you’ll rarely have a reason not to use a make function. And unless you have a compelling reason for doing otherwise, using a make function is what you should do.*” book recommendation.*

*Book structure. Resume: “*Things to Remember

• Compared to direct use of new, make functions eliminate source code duplication, improve exception safety, and, for std::make\_shared and std::allocate\_shared, generate code that’s smaller and faster.

• Situations where use of make functions is inappropriate include the need to specify custom deleters and a desire to pass braced initializers.

• For std::shared\_ptrs, additional situations where make functions may be ill-advised include (1) classes with custom memory management and (2) systems with memory concerns, very large objects, and std::weak\_ptrs that outlive the corresponding std::shared\_ptrs.*” using context of make functions.*

### **Item 22: When using the Pimpl Idiom, define special member functions in the implementation file.**

*Intbu this recommendation concerning Pimpl Idiom.*

*Purpose and structure of Pimpl: “*If you’ve ever had to combat excessive build times, you’re familiar with the Pimpl (“pointer to implementation”) Idiom. That’s the technique whereby you replace the data members of a class with a pointer to an implementation class (or struct), put the data members that used to be in the primary class into the implementation class, and access those data members indirectly through the pointer.*”*

*Example of Pimpl: “*For example, suppose Widget looks like this:

class Widget { // in header "widget.h"

public:

Widget();

…

private:

std::string name;

std::vector<double> data;

Gadget g1, g2, g3; // Gadget is some user};

// defined type

Because Widget’s data members are of types std::string, std::vector, and Gadget, headers for those types must be present for Widget to compile, and that means that Widget clients must #include <string>, <vector>, and gadget.h. Those headers increase the compilation time for Widget clients, plus they make those clients dependent on the contents of the headers. If a header’s content changes, Widget clients must recompile. The standard headers <string> and <vector> don’t change very often, but it could be that gadget.h is subject to frequent revision.*” C++ program compilation and using by client code. 2 problems to be solved by Pimpl: decrease dependencies and recompilations. Frequency of revision is characteristic of classes in C++. Intbu possible values of its measure.*

*C++ history and variant of Pimpl realization: “*Applying the Pimpl Idiom in C++98 could have Widget replace its data members with a raw pointer to a struct that has been declared, but not defined:

class Widget { // still in header "widget.h"

public:

Widget();

~Widget(); // dtor is needed—see below

…

private:

struct Impl; // declare implementation struct

Impl \*pImpl; // and pointer to it

};

Because Widget no longer mentions the types std::string, std::vector, and Gadget, Widget clients no longer need to #include the headers for these types. That speeds compilation, and it also means that if something in these headers changes, Widget clients are unaffected.*” advantages of Pimpl.*

*Incomplete type definition: “*A type that has been declared, but not defined, is known as an incomplete type. Widget::Impl is such a type. There are very few things you can do with an incomplete type, but declaring a pointer to it is one of them. The Pimpl Idiom takes advantage of that.*” C++ structure: possibilities of incomplete types.*

*p. 167*

*Pimpl structure: “*Part 1 of the Pimpl Idiom is the declaration of a data member that’s a pointer to an incomplete type. Part 2 is the dynamic allocation and deallocation of the object that holds the data members that used to be in the original class. The allocation and deallocation code goes in the implementation file, e.g., for Widget, in widget.cpp:

#include "widget.h" // in impl. file "widget.cpp"

#include "gadget.h"

#include <string>

#include <vector>

struct Widget::Impl { // definition of Widget::Impl

std::string name; // with data members formerly

std::vector<double> data; // in Widget

Gadget g1, g2, g3;

};

Widget::Widget() // allocate data members for

: pImpl(new Impl) // this Widget object

{}

Widget::~Widget() // destroy data members for

{ delete pImpl; } // this object

Here I’m showing #include directives to make clear that the overall dependencies on the headers for std::string, std::vector, and Gadget continue to exist. However, these dependencies have been moved from widget.h (which is visible to and used by Widget clients) to widget.cpp (which is visible to and used only by the Widget implementer). I’ve also highlighted the code that dynamically allocates and deallocates the Impl object. The need to deallocate this object when a Widget is destroyed is what necessitates the Widget destructor.*” C++ structure. Purpose of header files: show only interface. Example of memory governing in C++: manual delete of internally created objects.*

*Usefulness of std::unique\_ptr for Pimpl. Its using context: “*But I’ve shown you C++98 code, and that reeks of a bygone millennium. It uses raw pointers and raw new and raw delete and it’s all just so…raw. This chapter is built on the idea that smart pointers are preferable to raw pointers, and if what we want is to dynamically allocate a Widget::Impl object inside the Widget constructor and have it destroyed at the same time the Widget is, std::unique\_ptr (see Item 18) is precisely the tool we need.*”*

*p. 168*

*Example of this: “*Replacing the raw pImpl pointer with a std::unique\_ptr yields this code for the header file, class Widget { // in "widget.h"

public:

Widget();

…

private:

struct Impl;

std::unique\_ptr<Impl> pImpl; // use smart pointer

}; // instead of raw pointer

and this for the implementation file:

#include "widget.h" // in "widget.cpp"

#include "gadget.h"

#include <string>

#include <vector>

struct Widget::Impl { // as before

std::string name;

std::vector<double> data;

Gadget g1, g2, g3;

};

Widget::Widget() // per Item 21, create

: pImpl(std::make\_unique<Impl>()) // std::unique\_ptr

{} // via std::make\_unique

You’ll note that the Widget destructor is no longer present. That’s because we have no code to put into it. std::unique\_ptr automatically deletes what it points to when it (the std::unique\_ptr) is destroyed, so we need not delete anything ourselves. That’s one of the attractions of smart pointers: they eliminate the need for us to sully our hands with manual resource release.*” advantage of smart pointers: eliminate the need to sully our hands with manual resourse release.*

*Example continue and potential problem: “*This code compiles, but, alas, the most trivial client use doesn’t:

#include "widget.h"

Widget w; // error!

The error message you receive depends on the compiler you’re using, but the text generally mentions something about applying sizeof or delete to an incomplete type. Those operations aren’t among the things you can do with such types.*” incomplete type possibilities in C++. Compiler differences.*

*Problem description: “*This apparent failure of the Pimpl Idiom using std::unique\_ptrs is alarming, because (1) std::unique\_ptr is advertised as supporting incomplete types, and (2) the Pimpl Idiom is one of std::unique\_ptrs most common use cases. Fortunately, getting the code to work is easy. All that’s required is a basic understanding of the cause of the problem. The issue arises due to the code that’s generated when w is destroyed (e.g., goes out of scope). At that point, its destructor is called. In the class definition using std::unique\_ptr, we didn’t declare a destructor, because we didn’t have any code to put into it. In accord with the usual rules for compiler-generated special member*”*

*p. 169*

*description continue: “*functions (see Item 17), the compiler generates a destructor for us. Within that destructor, the compiler inserts code to call the destructor for Widget’s data member pImpl. pImpl is a std::unique\_ptr<Widget::Impl>, i.e., a std::unique\_ptr using the default deleter. The default deleter is a function that uses delete on the raw pointer inside the std::unique\_ptr. Prior to using delete, however, implementations typically have the default deleter employ C++11’s static\_assert to ensure that the raw pointer doesn’t point to an incomplete type. When the compiler generates code for the destruction of the Widget w, then, it generally encounters a static\_assert that fails, and that’s usually what leads to the error message. This message is associated with the point where w is destroyed, because Widget’s destructor, like all compiler-generated special member functions, is implicitly inline. The message itself often refers to the line where w is created, because it’s the source code explicitly creating the object that leads to its later implicit destruction.*” example of subtle problem. C++ structure.*

*Fix of the problem: “*To fix the problem, you just need to make sure that at the point where the code to destroy the std::unique\_ptr<Widget::Impl> is generated, Widget::Impl is a complete type. The type becomes complete when its definition has been seen, and Widget::Impl is defined inside widget.cpp. The key to successful compilation, then, is to have the compiler see the body of Widget’s destructor (i.e., the place where the compiler will generate code to destroy the std::unique\_ptr data member) only inside widget.cpp after Widget::Impl has been defined. Arranging for that is simple. Declare Widget’s destructor in widget.h, but don’t define it there:

class Widget { // as before, in "widget.h"

public:

Widget();

~Widget(); // declaration only

…

private: // as before

struct Impl;

std::unique\_ptr<Impl> pImpl;

};

Define it in widget.cpp after Widget::Impl has been defined:

#include "widget.h" // as before, in "widget.cpp"

#include "gadget.h"

#include <string>

#include <vector>

struct Widget::Impl { // as before, definition of

std::string name; // Widget::Impl

std::vector<double> data;

Gadget g1, g2, g3;

};

Widget::Widget() // as before

: pImpl(std::make\_unique<Impl>())

{}

Widget::~Widget() // ~Widget definition

{}*” intbu why destructor without body works well.*

*p. 170*

*default using context and fix improvement: “*This works well, and it requires the least typing, but if you want to emphasize that the compiler-generated destructor would do the right thing—that the only reason you declared it was to cause its definition to be generated in Widget’s implementation file, you can define the destructor body with “= default”:

Widget::~Widget() = default; // same effect as above*”*

*Pimple structure: move support: “*Classes using the Pimpl Idiom are natural candidates for move support, because compiler-generated move operations do exactly what’s desired: perform a move on the underlying std::unique\_ptr. As Item 17 explains, the declaration of a destructor in Widget prevents compilers from generating the move operations, so if you want move support, you must declare the functions yourself.*” Pimple using technique in modern C++.*

*Example of it: “*Given that the compiler-generated versions would behave correctly, you’re likely to be tempted to implement them as follows:

class Widget { // still in

public: // "widget.h"

Widget();

~Widget();

Widget(Widget&& rhs) = default; // right idea,

Widget& operator=(Widget&& rhs) = default; // wrong code!

…

private: // as before

struct Impl;

std::unique\_ptr<Impl> pImpl;

};

This approach leads to the same kind of problem as declaring the class without a destructor, and for the same fundamental reason. The compiler-generated move assignment operator needs to destroy the object pointed to by pImpl before reassigning it, but in the Widget header file, pImpl points to an incomplete type. The situation is different for the move constructor. The problem there is that compilers typically generate code to destroy pImpl in the event that an exception arises inside the move constructor, and destroying pImpl requires that Impl be complete.*” another problem. With assignment operator. I don’t see its origin. Compiler structure. C++ structure.*

*p. 171*

*fix of this problem: “*Because the problem is the same as before, so is the fix—move the definition of the move operations into the implementation file:

class Widget { // still in "widget.h"

public:

Widget();

~Widget();

Widget(Widget&& rhs); // declarations

Widget& operator=(Widget&& rhs); // only

…

private: // as before

struct Impl;

std::unique\_ptr<Impl> pImpl;

};

#include <string> // as before,

… // in "widget.cpp"

struct Widget::Impl { … }; // as before

Widget::Widget() // as before

: pImpl(std::make\_unique<Impl>())

{}

Widget::~Widget() = default; // as before

Widget::Widget(Widget&& rhs) = default; // defini-

Widget& Widget::operator=(Widget&& rhs) = default; // tions*” this is example of the necessity to know type realization and C++ compiler structure for better understanding of problems.*

*Pimpl structure, purpose and conceptions: “*The Pimpl Idiom is a way to reduce compilation dependencies between a class’s implementation and the class’s clients, but, conceptually, use of the idiom doesn’t change what the class represents. The original Widget class contained std::string, std::vector, and Gadget data members, and, assuming that Gadgets, like std::strings and std::vectors, can be copied, it would make sense for Widget to support the copy operations.*”*

*Copying of types with Pimpl: “*We have to write these functions ourselves, because (1) compilers won’t generate copy operations for classes with move-only types like std::unique\_ptr and (2) even if they did, the generated functions would copy only the std::unique\_ptr (i.e., perform a shallow copy), and we want to copy what the pointer points to (i.e., perform a deep copy).*” kinds of copying: shallow and deep.*

*p. 172*

*fix of the problem: “*In a ritual that is by now familiar, we declare the functions in the header file and implement them in the implementation file:

class Widget { // still in "widget.h"

public:

… // other funcs, as before

Widget(const Widget& rhs); // declarations

Widget& operator=(const Widget& rhs); // only

private: // as before

struct Impl;

std::unique\_ptr<Impl> pImpl;

};

#include "widget.h" // as before,

… // in "widget.cpp"

struct Widget::Impl { … }; // as before

Widget::~Widget() = default; // other funcs, as before

Widget::Widget(const Widget& rhs) // copy ctor

: pImpl(std::make\_unique<Impl>(\*rhs.pImpl))

{}

Widget& Widget::operator=(const Widget& rhs) // copy operator=

{

\*pImpl = \*rhs.pImpl;

return \*this;

}

Both function implementations are conventional. In each case, we simply copy the fields of the Impl struct from the source object (rhs) to the destination object (\*this). Rather than copy the fields one by one, we take advantage of the fact that compilers will create the copy operations for Impl, and these operations will copy each field automatically. We thus implement Widget’s copy operations by calling Widget::Impl’s compiler-generated copy operations. In the copy constructor, note that we still follow the advice of Item 21 to prefer use of std::make\_unique over direct use of new.*” C++ good practice. Way to copy object: value to value.*

*p. 173*

*std::unique\_ptr using context. Pimpl realization: “*For purposes of implementing the Pimpl Idiom, std::unique\_ptr is the smart pointer to use, because the pImpl pointer inside an object (e.g., inside a Widget) has exclusive ownership of the corresponding implementation object (e.g., the Widget::Impl object).*”*

*std::shared\_ptr using in Pimpl: “*Still, it’s interesting to note that if we were to use std::shared\_ptr instead of std::unique\_ptr for pImpl, we’d find that the advice of this Item no longer applied. There’d be no need to declare a destructor in Widget, and without a user-declared destructor, compilers would happily generate the move operations, which would do exactly what we’d want them to. That is, given this code in widget.h,

class Widget { // in "widget.h"

public:

Widget();

… // no declarations for dtor

// or move operations

private:

struct Impl;

std::shared\_ptr<Impl> pImpl; // std::shared\_ptr

}; // instead of std::unique\_ptr

and this client code that #includes widget.h,

Widget w1;

auto w2(std::move(w1)); // move-construct w2

w1 = std::move(w2); // move-assign w1

everything would compile and run as we’d hope: w1 would be default constructed, its value would be moved into w2, that value would be moved back into w1, and then both w1 and w2 would be destroyed (thus causing the pointed-to Widget::Impl object to be destroyed).*”*

*Comparison of both realizations of Pimpl: with std::unique\_ptr and std::shared\_ptr: “*The difference in behavior between std::unique\_ptr and std::shared\_ptr for pImpl pointers stems from the differing ways these smart pointers support custom deleters. For std::unique\_ptr, the type of the deleter is part of the type of the smart pointer, and this makes it possible for compilers to generate smaller runtime data structures and faster runtime code. A consequence of this greater efficiency is that pointed-to types must be complete when compiler-generated special functions (e.g., destructors or move operations) are used. For std::shared\_ptr, the type of the deleter is not part of the type of the smart pointer. This necessitates larger runtime data structures and somewhat slower code, but pointed-to types need not be complete when compiler-generated special functions are employed.*”*

*p. 174*

*Book recommendation about choose of Pimpl realization: “*For the Pimpl Idiom, there’s not really a trade-off between the characteristics of std::unique\_ptr and std::shared\_ptr, because the relationship between classes like Widget and classes like Widget::Impl is exclusive ownership, and that makes std::unique\_ptr the proper tool for the job. Nevertheless, it’s worth knowing that in other situations—situations where shared ownership exists (and std::shared\_ptr is hence a fitting design choice), there’s no need to jump through the function-definition hoops that use of std::unique\_ptr entails.*” It is not evident for me why std::unique\_ptr is more good. Я адчуваю, што разуменне Майерса ў дадзеным пытанні можна апісаць словамі ”здаецца, што так лепей”.*

*Book structure. Resume: “*Things to Remember

• The Pimpl Idiom decreases build times by reducing compilation dependencies between class clients and class implementations.

• For std::unique\_ptr pImpl pointers, declare special member functions in the class header, but implement them in the implementation file. Do this even if the default function implementations are acceptable.

• The above advice applies to std::unique\_ptr, but not to std::shared\_ptr.*”*

P. 175

## CHAPTER 5 Rvalue References, Move Semantics, and Perfect Forwarding

When you first…

*29.12.2016*

*Move semantics C++ idiom: “*Move semantics makes it possible for compilers to replace expensive copying operations with less expensive moves. In the same way that copy constructors and copy assignment operators give you control over what it means to copy objects, move constructors and move assignment operators offer control over the semantics of moving. Move semantics also enables the creation of move-only types, such as std::unique\_ptr, std::future, and std::thread.*” examples of move-only types.*

*Perfect forwarding C++ idiom definition: “*Perfect forwarding makes it possible to write function templates that take arbitrary arguments and forward them to other functions such that the target functions receive exactly the same arguments as were passed to the forwarding functions.*” intbu this concept.*

*Rvalue reference structure: “*Rvalue references are the glue that ties these two rather disparate features together. They’re the underlying language mechanism that makes both move semantics and perfect forwarding possible.*” fundamental relation of C++.*

*Nuances of these three concepts: “*The more experience you have with these features, the more you realize that your initial impression was based on only the metaphorical tip of the proverbial iceberg. The world of move semantics, perfect forwarding, and rvalue references is more nuanced than it appears.*”*

*Std::move structure: “*std::move doesn’t move anything, for example, and perfect forwarding is imperfect.*”*

*Move opeations nuances: “*Move operations aren’t always cheaper than copying; when they are, they’re not always as cheap as you’d expect; and they’re not always called in a context where moving is valid.*”*

*Rvalue reference nuances: “*The construct “type&&” doesn’t always represent an rvalue reference.*” C++ construct. “construct” is fundamental feature of programming languages.*

*p. 176*

*book structure and these conceptions description: “*No matter how far you dig into these features, it can seem that there’s always more to uncover. Fortunately, there is a limit to their depths. This chapter will take you to the bedrock. Once you arrive, this part of C++11 will make a lot more sense. You’ll know the usage conventions for std::move and std::forward, for example. You’ll be comfortable with the ambiguous nature of “type&&”. You’ll understand the reasons for the surprisingly varied behavioral profiles of move operations. All those pieces will fall into place. At that point, you’ll be back where you started, because move semantics, perfect forwarding, and rvalue references will once again seem pretty straightforward. But this time, they’ll stay that way.*”*

*Lvalue structure. Function parameter is always lvalue: “*In the Items in this chapter, it’s especially important to bear in mind that a parameter is always an lvalue, even if its type is an rvalue reference. That is, given

void f(Widget&& w);

the parameter w is an lvalue, even though its type is rvalue-reference-to-Widget. (If this surprises you, please review the overview of lvalues and rvalues that begins on page 2.)*”*

## **Item 23: Understand std::move and std::forward**

*Kind of definition: “*It’s useful to approach std::move and std::forward in terms of what they don’t do.*” and example of it.*

*Std::move structure: “*std::move doesn’t move anything.*”*

*Std::forward structure: “*std::forward doesn’t forward anything.*”*

*They common property: “*At runtime, neither does anything at all. They generate no executable code. Not a single byte.*”*

*Std::move and std::forward structure: “*std::move and std::forward are merely functions (actually function templates) that perform casts. std::move unconditionally casts its argument to an rvalue, while std::forward performs this cast only if a particular condition is fulfilled. That’s it. The explanation leads to a new set of questions, but, fundamentally, that’s the complete story.*” аспекты дын.разумення: адказ на адны пытанні выклікае другія. Так што зноў узбагачаецца 3у адначасова з павелічэннем ст на 4у і выш.*

*Example purpose: “*To make the story more concrete, here’s a sample implementation of std::move in C++11.*” Aspect of representation theory.*

*Book example: “*It’s not fully conforming to the details of the Standard, but it’s very close.

template<typename T> // in namespace std

typename remove\_reference<T>::type&&

move(T&& param)

{

using ReturnType = // alias declaration;

typename remove\_reference<T>::type&&; // see Item 9

return static\_cast<ReturnType>(param);

}

158 | Item 22

www.it-ebooks.info*” intbu this code. I don’t understand typename. Book structure (Item 9).*

*p. 177*

*example continue: “*I’ve highlighted two parts of the code for you. One is the name of the function, because the return type specification is rather noisy, and I don’t want you to lose your bearings in the din. The other is the cast that comprises the essence of the function. As you can see, std::move takes a reference to an object (a universal reference, to be precise—see Item 24) and it returns a reference to the same object.*” book structure (Item 24)*

*Example description: “*The “&&” part of the function’s return type implies that std::move returns an rvalue reference, but, as Item 28 explains, if the type T happens to be an lvalue reference, T&& would become an lvalue reference*” book structure (Item 28)*

*example description: “*To prevent this from happening, the type trait (see Item 9) std::remove\_reference is applied to T, thus ensuring that “&&” is applied to a type that isn’t a reference. That guarantees that std::move truly returns an rvalue reference, and that’s important, because rvalue references returned from functions are rvalues. Thus, std::move casts its argument to an rvalue, and that’s all it does.*” Book structure (Item 9). Std::move structure.*

*Alias declarations and function return type deduction advantages demonstration: “*As an aside, std::move can be implemented with less fuss in C++14. Thanks to function return type deduction (see Item 3) and to the Standard Library’s alias template std::remove\_reference\_t (see Item 9), std::move can be written this way:

template<typename T> // C++14; still in

decltype(auto) move(T&& param) // namespace std

{

using ReturnType = remove\_reference\_t<T>&&;

return static\_cast<ReturnType>(param);

}

Easier on the eyes, no?*” the same example, but in C++14. Book structure (Item 3 and Item 9).*

*Inaccuracy of std::move name: “*Because std::move does nothing but cast its argument to an rvalue, there have been suggestions that a better name for it might have been something like rvalue\_cast. Be that as it may, the name we have is std::move, so it’s important to remember what std::move does and doesn’t do. It does cast. It doesn’t move.*”*

*Using context of std::move: “*Of course, rvalues are candidates for moving, so applying std::move to an object tells the compiler that the object is eligible to be moved from. That’s why std::move has the name it does: to make it easy to designate objects that may be moved from.*”*

*Rvalue using contexts: “*In truth, rvalues are only usually candidates for moving. Suppose you’re writing a class representing annotations. The class’s constructor takes a std::string parameter comprising the annotation, and it copies the parameter to a data member. Flush with the information in Item 41, you declare a by-value parameter:

class Annotation {

public:

explicit Annotation(std::string text); // param to be copied,

// so per Item 41,

}; // pass by value*” book structure (Item 41)*

*p. 176*

*example continue: “*But Annotation’s constructor needs only to read text’s value. It doesn’t need to modify it. In accord with the time-honored tradition of using const whenever possible, you revise your declaration such that text is const:

class Annotation {

public:

explicit Annotation(const std::string text)

…

};

To avoid paying for a copy operation when copying text into a data member, you remain true to the advice of Item 41 and apply std::move to text, thus producing an rvalue:

class Annotation {

public:

explicit Annotation(const std::string text)

: value(std::move(text)) // "move" text into value; this code

{ … } // doesn't do what it seems to!

…

private:

std::string value;

};

This code compiles. This code links. This code runs. This code sets the data member value to the content of text. The only thing separating this code from a perfect realization of your vision is that text is not moved into value, it’s copied. Sure, text is cast to an rvalue by std::move, but text is declared to be a const std::string, so before the cast, text is an lvalue const std::string, and the result of the cast is an rvalue const std::string, but throughout it all, the constness remains. Consider the effect that has when compilers have to determine which std::string constructor to call. There are two possibilities:

class string { // std::string is actually a

public: // typedef for std::basic\_string<char>

…

string(const string& rhs); // copy ctor

string(string&& rhs); // move ctor

…

};*”*

*p. 179*

*example description: “*In the Annotation constructor’s member initialization list, the result of std::move(text) is an rvalue of type const std::string. That rvalue can’t be passed to std::string’s move constructor, because the move constructor takes an rvalue reference to a non-const std::string. The rvalue can, however, be passed to the copy constructor, because an lvalue-reference-to-const is permitted to bind to a const rvalue. The member initialization therefore invokes the copy constructor in std::string, even though text has been cast to an rvalue! Such behavior is essential to maintaining const-correctness. Moving a value out of an object generally modifies the object, so the language should not permit const objects to be passed to functions (such as move constructors) that could modify them.*” C++ feature “maintaining const-correctness”.*

*Nuances of move semantics concenrning const qualifier. Move semantics using technique. Book recommendations: “*There are two lessons to be drawn from this example. First, don’t declare objects const if you want to be able to move from them. Move requests on const objects are silently transformed into copy operations. Second, std::move not only doesn’t actually move anything, it doesn’t even guarantee that the object it’s casting will be eligible to be moved. The only thing you know for sure about the result of applying std::move to an object is that it’s an rvalue.*” difference between rvalue and movable type.*

*Std::forward structure: “*The story for std::forward is similar to that for std::move, but whereas std::move unconditionally casts its argument to an rvalue, std::forward does it only under certain conditions. std::forward is a conditional cast.*”*

*Example of typical using of std::forward: “*To understand when it casts and when it doesn’t, recall how std::forward is typically used. The most common scenario is a function template taking a universal reference parameter that is to be passed to another function:

void process(const Widget& lvalArg); // process lvalues

void process(Widget&& rvalArg); // process rvalues

template<typename T> // template that passes

void logAndProcess(T&& param) // param to process

{

auto now = // get current time

std::chrono::system\_clock::now();

makeLogEntry("Calling 'process'", now);

process(std::forward<T>(param));

}

Consider two calls to logAndProcess, one with an lvalue, the other with an rvalue:

Widget w;

logAndProcess(w); // call with lvalue

logAndProcess(std::move(w)); // call with rvalue*”*

*p. 180*

*example continue: “*Inside logAndProcess, the parameter param is passed to the function process. Process is overloaded for lvalues and rvalues. When we call logAndProcess with an lvalue, we naturally expect that lvalue to be forwarded to process as an lvalue, and when we call logAndProcess with an rvalue, we expect the rvalue overload of process to be invoked. But param, like all function parameters, is an lvalue. Every call to process inside logAndProcess will thus want to invoke the lvalue overload for process. To prevent this, we need a mechanism for param to be cast to an rvalue if and only if the argument with which param was initialized—the argument passed to logAndProcess— was an rvalue. This is precisely what std::forward does. That’s why std::forward is a conditional cast: it casts to an rvalue only if its argument was initialized with an rvalue.*” std::forward definition. It’s conditional cast.*

*Std::forward realization: “*You may wonder how std::forward can know whether its argument was initialized with an rvalue. In the code above, for example, how can std::forward tell whether param was initialized with an lvalue or an rvalue? The brief answer is that that information is encoded in logAndProcess’s template parameter T. That parameter is passed to std::forward, which recovers the encoded information. For details on exactly how that works, consult Item 28.*” book structure (Item 28). There is more detailed information.*

*Comparison of std::move and std::forward: “*Given that both std::move and std::forward boil down to casts, the only differ- ence being that std::move always casts, while std::forward only sometimes does, you might ask whether we can dispense with std::move and just use std::forward everywhere. From a purely technical perspective, the answer is yes: std::forward can do it all. std::move isn’t necessary. Of course, neither function is really necessary, because we could write casts everywhere, but I hope we agree that that would be, well, yucky.*”*

*Std::move attractions: “*std::move’s attractions are convenience, reduced likelihood of error, and greater clarity.*”*

*Example of it: “*Consider a class where we want to track how many times the move constructor is called. A static counter that’s incremented during move construction is all we need. Assuming the only non-static data in the class is a std::string, here’s the conventional way (i.e., using std::move) to implement the move constructor:

class Widget {

public:

Widget(Widget&& rhs)

: s(std::move(rhs.s))

{ ++moveCtorCalls; }

…

private:

static std::size\_t moveCtorCalls;

std::string s;

};

To implement the same behavior with

std::forward, the code would look like this:

class Widget {

public:

Widget(Widget&& rhs) // unconventional,

: s(std::forward<std::string>(rhs.s)) // undesirable

{ ++moveCtorCalls; } // implementation

…

};*”*

*p. 181*

*std::move advantages over using of std::forward: “*Note first that std::move requires only a function argument (rhs.s), while std::forward requires both a function argument (rhs.s) and a template type argument (std::string). Then note that the type we pass to std::forward should be a non-reference, because that’s the convention for encoding that the argument being passed is an rvalue (see Item 28). Together, this means that std::move requires less typing than std::forward, and it spares us the trouble of passing a type argument that encodes that the argument we’re passing is an rvalue. It also eliminates the possibility of our passing an incorrect type (e.g., std::string&, which would result in the data member s being copy constructed instead of move constructed).*” book structure (Item 28).*

*Difference between them and their structure: “*More importantly, the use of std::move conveys an unconditional cast to an rvalue, while the use of std::forward indicates a cast to an rvalue only for references to which rvalues have been bound. Those are two very different actions. The first one typically sets up a move, while the second one just passes—forwards—an object to another function in a way that retains its original lvalueness or rvalueness. Because these actions are so different, it’s good that we have two different functions (and function names) to distinguish them.*”*

*Book structure. Resume: “*Things to Remember

• std::move performs an unconditional cast to an rvalue. In and of itself, it doesn’t move anything.

• std::forward casts its argument to an rvalue only if that argument is bound to an rvalue.

• Neither std::move nor std::forward do anything at runtime.*”*

## **Item 24: Distinguish universal references from rvalue references**

*To conceptrs of C++. What difference is between them?*

*Intbu translation: “*It’s been said that the truth shall set you free, but under the right circumstances, a well-chosen lie can be equally liberating. This Item is such a lie. Because we’re dealing with software, however, let’s eschew the word “lie” and instead say that this Item comprises an “abstraction.”*”*

*Rvalue reference syntax and situation where construct && is not rvalue. These situations are connected with type parameters? : “*To declare an rvalue reference to some type T, you write T&&. It thus seems reasonable to assume that if you see “T&&” in source code, you’re looking at an rvalue reference. Alas, it’s not quite that simple:

void f(Widget&& param); // rvalue reference

Widget&& var1 = Widget(); // rvalue reference

auto&& var2 = var1; // not rvalue reference

template<typename T>

void f(std::vector<T>&& param); // rvalue reference

template<typename T>

void f(T&& param); // not rvalue reference*”*

*Meaning of construct T&&: “*In fact, “T&&” has two different meanings. One is rvalue reference, of course. Such references behave exactly the way you expect: they bind only to rvalues, and their primary raison d’être is to identify objects that may be moved from.*” rvalue purpose.*

*Other meaning of construct T&&: “*The other meaning for “T&&” is either rvalue reference or lvalue reference. Such references look like rvalue references in the source code (i.e., “T&&”), but they can behave as if they were lvalue references (i.e., “T&”). Their dual nature permits them to bind to rvalues (like rvalue references) as well as lvalues (like lvalue references). Furthermore, they can bind to const or non-const objects, to volatile or non-volatile objects, even to objects that are both const and volatile. They can bind to virtually anything. Such unprecedentedly flexible references deserve a name of their own. I call them universal references.1*” concept “universal reference” definition.*

*Book recommendation nad using technique of universal references: “*Item 25 explains that universal references should almost always have std::forward applied to them, and as this book goes to press, some members of the C++ community have started referring to universal references as forwarding references.*” book structure (Item 25).*

*Arising contexts of universal references: “*Universal references arise in two contexts. The most common is function template parameters, such as this example from the sample code above: *”*

*p. 183*

*Arising contexts of universal reference continue: “*template<typename T>

void f(T&& param); // param is a universal reference

The second context is auto declarations, including this one from the sample code above:

auto&& var2 = var1; // var2 is a universal reference

What these contexts have in common is the presence of type deduction. In the template f, the type of param is being deduced, and in the declaration for var2, var2’s type is being deduced. Compare that with the following examples (also from the sample code above), where type deduction is missing. If you see “T&&” without type deduction, you’re looking at an rvalue reference:

void f(Widget&& param); // no type deduction;

// param is an rvalue reference

Widget&& var1 = Widget(); // no type deduction;

// var1 is an rvalue reference*” connection between auto and universal references. Type deduction is situation of universal references use. Arising context of rvalue reference.*

*Rvalue reference structure and reference structure: “*Because universal references are references, they must be initialized. The initializer for a universal reference determines whether it represents an rvalue reference or an lvalue reference. If the initializer is an rvalue, the universal reference corresponds to an rvalue reference. If the initializer is an lvalue, the universal reference corresponds to an lvalue reference.*” Universal reference structure.*

*Example of it: “*For universal references that are function parameters, the initializer is provided at the call site:

template<typename T>

void f(T&& param); // param is a universal reference

Widget w;

f(w); // lvalue passed to f; param's type is

// Widget& (i.e., an lvalue reference)

f(std::move(w)); // rvalue passed to f; param's type is

// Widget&& (i.e., an rvalue reference)*” universal reference initialization as function parameter.*

*Arising context of rvalue and universal reference: “*For a reference to be universal, type deduction is necessary, but it’s not sufficient. The form of the reference declaration must also be correct, and that form is quite constrained. It must be precisely “T&&”. Look again at this example from the sample code we saw earlier:

template<typename T>

void f(std::vector<T>&& param); // param is an rvalue reference

When f is invoked, the type T will be deduced (unless the caller explicitly specifies it, an edge case we’ll not concern ourselves with). But the form of param’s type declaration isn’t “T&&”, it’s “std::vector<T>&&”. That rules out the possibility that param is a universal reference. Param is therefore an rvalue reference, something that your compilers will be happy to confirm for you if you try to pass an lvalue to f:

std::vector<int> v;

f(v); // error! can't bind lvalue to

// rvalue reference*”*

*p. 184*

*if there is no type deduction we have rvalue reference: “*Even the simple presence of a const qualifier is enough to disqualify a reference from being universal:

template<typename T>

void f(const T&& param); // param is an rvalue reference

If you’re in a template and you see a function parameter of type “T&&”, you might think you can assume that it’s a universal reference. You can’t. That’s because being in a template doesn’t guarantee the presence of type deduction. Consider this push\_back member function in std::vector:

template<class T, class Allocator = allocator<T>> // from C++

class vector { // Standards

public:

void push\_back(T&& x);

…

};

push\_back’s parameter certainly has the right form for a universal reference, but there’s no type deduction in this case. That’s because push\_back can’t exist without a particular vector instantiation for it to be part of, and the type of that instantiation fully determines the declaration for push\_back. That is, saying

std::vector<Widget> v;

causes the std::vector template to be instantiated as follows:

class vector<Widget, allocator<Widget>> {

public:

void push\_back(Widget&& x); // rvalue reference

…

};

Now you can see clearly that push\_back employs no type deduction. This push\_back for vector<T> (there are two—the function is overloaded) always declares a parameter of type rvalue-reference-to-T. *” examples of it. Intbu the last.*

*p. 185*

*example of type deduction presence: “*In contrast, the conceptually similar emplace\_back member function in std::vector does employ type deduction:

template<class T, class Allocator = allocator<T>> // still from

class vector { // C++

public: // Standards

template <class... Args>

void emplace\_back(Args&&... args);

…

};

Here, the type parameter Args is independent of vector’s type parameter T, so Args must be deduced each time emplace\_back is called. (Okay, Args is really a parameter pack, not a type parameter, but for purposes of this discussion, we can treat it as if it were a type parameter.)*” what is parameter pack?*

*Type deduction is absent where caller explicitly specifies type. Type deduction arising context: “*The fact that emplace\_back’s type parameter is named Args, yet it’s still a universal reference, reinforces my earlier comment that it’s the form of a universal reference that must be “T&&”. There’s no requirement that you use the name T. For example, the following template takes a universal reference, because the form (“type&&”) is right, and param’s type will be deduced (again, excluding the corner case where the caller explicitly specifies the type):

template<typename MyTemplateType> // param is a

void someFunc(MyTemplateType&& param); // universal reference*”*

*C++ history and auto universal referneces arising context: “*I remarked earlier that auto variables can also be universal references. To be more precise, variables declared with the type auto&& are universal references, because type deduction takes place and they have the correct form (“T&&”). Auto universal references are not as common as universal references used for function template parameters, but they do crop up from time to time in C++11. They crop up a lot more in C++14, because C++14 lambda expressions may declare auto&& parameters.*” connection with lambda expressions. Aspect of lambda expressions.*

*Example of it: “*For example, if you wanted to write a C++14 lambda to record the time taken in an arbitrary function invocation, you could do this:

auto timeFuncInvocation =

[](auto&& func, auto&&... params) // C++14

{

start timer;

std::forward<decltype(func)>(func)( // invoke func

std::forward<decltype(params)>(params)... // on params

);

stop timer and record elapsed time;

};

If your reaction to the “std::forward<decltype(blah blah blah)>” code inside the lambda is, “What the…?!”, that probably just means you haven’t yet read Item 33. Don’t worry about it. The important thing in this Item is the auto&& parameters that the lambda declares. Func is a universal reference that can be bound to any callable object, lvalue or rvalue. Args is zero or more universal references (i.e., a universal reference parameter pack) that can be bound to any number of objects of arbitrary types. The result, thanks to auto universal references, is that timeFuncInvocation can time pretty much any function execution. (For information on the difference between “any” and “pretty much any,” turn to Item 30.)*” Is this example of meta-programming? Intbu. Book structure (Item 30, Item 33). Advantage of lambda expressions.*

*p. 186*

*book structure and C++ problem: “*Bear in mind that this entire Item—the foundation of universal references—is a lie… er, an “abstraction.” The underlying truth is known as reference collapsing, a topic to which Item 28 is dedicated.*” Book structure (Item 28). Цікавы прыклад рэпрэзентацыі. Калі выказваецца паверхневы бок сутнасці і адзначаецца, што гэта часта няпраўда. Гэта робіцца праз увядзенне па ходу зместа такіх сказікаў і ўставачак. Гэта працуе, толькі калі большая частка параграфа моцна канцэтруецца на сутнасці бачымасці і толькі зрэдку нагадвае пра гэту бачымасць. Я так адчуваю. Паразважаць больш глыбока.*

*Importance to distringuish rvalues from universal references: “*But the truth doesn’t make the abstraction any less useful. Distinguishing between rvalue references and universal references will help you read source code more accurately (“Does that T&& I’m looking at bind to rvalues only or to everything?”), and it will avoid ambiguities when you communicate with your colleagues (“I’m using a universal reference here, not an rvalue reference…”). It will also allow you to make sense of Items 25 and 26, which rely on the distinction. So embrace the abstraction. Revel in it. Just as Newton’s laws of motion (which are technically incorrect) are typically just as useful as and easier to apply than Einstein’s theory of general relativity (“the truth”), so is the notion of universal references normally preferable to working through the details of reference collapsing. *” book structure (Item 25, Item 26). Good analogy with physics.*

*Book structure. Resume: “*Things to Remember

• If a function template parameter has type T&& for a deduced type T, or if an object is declared using auto&&, the parameter or object is a universal reference.

• If the form of the type declaration isn’t precisely type&&, or if type deduction does not occur, type&& denotes an rvalue reference.

• Universal references correspond to rvalue references if they’re initialized with rvalues. They correspond to lvalue references if they’re initialized with lvalues.*”*

## **Item 25: Use std::move on rvalue references, std::forward on universal references.**

*Using context of std::move and std::forward*

*p. 186*

*03.01.2016*

*Rvalue reference structure: “*Rvalue references bind only to objects that are candidates for moving. If you have an rvalue reference parameter, you know that the object it’s bound to may be moved:*”*

*p. 187*

*example of it: “*class Widget {

Widget(Widget&& rhs); // rhs definitely refers to an

… // object eligible for moving

};*”*

*Purpose of std::move: “*That being the case, you’ll want to pass such objects to other functions in a way that permits those functions to take advantage of the object’s rvalueness. The way to do that is to cast parameters bound to such objects to rvalues. As Item 23 explains, that’s not only what std::move does, it’s what it was created for:

class Widget {

public:

Widget(Widget&& rhs) // rhs is rvalue reference

: name(std::move(rhs.name)),

p(std::move(rhs.p))

{ … }

…

private:

std::string name;

std::shared\_ptr<SomeDataStructure> p;

};*” book structure (Item 23)*

*Std::forward purpose: “*A universal reference, on the other hand (see Item 24), might be bound to an object that’s eligible for moving. Universal references should be cast to rvalues only if they were initialized with rvalues. Item 23 explains that this is precisely what std::forward does:

class Widget {

public:

template<typename T>

void setName(T&& newName) // newName is

{ name = std::forward<T>(newName); } // universal reference

…

};*” book structure (Item 23, Item 24)*

*Rvalues ans universal references structure: “*In short, rvalue references should be unconditionally cast to rvalues (via std::move) when forwarding them to other functions, because they’re always bound to rvalues, and universal references should be conditionally cast to rvalues (via std::forward) when forwarding them, because they’re only sometimes bound to rvalues.”

“” Book structure (Item 23)

p. 188

using context of std::move and std::forward: “Item 23 explains that using std::forward on rvalue references can be made to exhibit the proper behavior, but the source code is wordy, error-prone, and unidiomatic, so you should avoid using std::forward with rvalue references. Even worse is the idea of using std::move with universal references, because that can have the effect of unexpectedly modifying lvalues (e.g., local variables):” *What does unidiomatic mean? Book structure (Item 23)*

*p. 188*

*example of big mistake connected with moving: “*class Widget {

public:

template<typename T>

void setName(T&& newName) // universal reference

{ name = std::move(newName); } // compiles, but is

… // bad, bad, bad!

private:

std::string name;

std::shared\_ptr<SomeDataStructure> p;

};

std::string getWidgetName(); // factory function

Widget w;

auto n = getWidgetName(); // n is local variable

w.setName(n); // moves n into w!

… // n's value now unknown

Here, the local variable n is passed to w.setName, which the caller can be forgiven for assuming is a read-only operation on n. But because setName internally uses std::move to unconditionally cast its reference parameter to an rvalue, n’s value will be moved into w.name, and n will come back from the call to setName with an unspecified value. That’s the kind of behavior that can drive callers to despair—possibly to violence.*” kind of undefined behavior.*

*Example continue: “*You might argue that setName shouldn’t have declared its parameter to be a universal reference. Such references can’t be const (see Item 24), yet setName surely shouldn’t modify its parameter. You might point out that if setName had simply been overloaded for const lvalues and for rvalues, the whole problem could have been avoided. Like this:

class Widget {

public:

void setName(const std::string& newName) // set from

{ name = newName; } // const lvalue

void setName(std::string&& newName) // set from

{ name = std::move(newName); } // rvalue

…

};*” way to avoid problem of std::move using with universal references.*

*p. 189*

*example continue: “*That would certainly work in this case, but there are drawbacks. First, it’s more source code to write and maintain (two functions instead of a single template). Second, it can be less efficient. For example, consider this use of setName:

w.setName("Adela Novak");

With the version of setName taking a universal reference, the string literal "Adela Novak" would be passed to setName, where it would be conveyed to the assignment operator for the std::string inside w. w’s name data member would thus be assigned directly from the string literal; no temporary std::string objects would arise. With the overloaded versions of setName, however, a temporary std::string object would be created for setName’s parameter to bind to, and this temporary std::string would then be moved into w’s data member. A call to setName would thus entail execution of one std::string constructor (to create the temporary), one std::string move assignment operator (to move newName into w.name), and one std::string destructor (to destroy the temporary). That’s almost certainly a more expensive execution sequence than invoking only the std::string assignment operator taking a const char\* pointer. The additional cost is likely to vary from implementation to implementation, and whether that cost is worth worrying about will vary from application to application and library to library, but the fact is that replacing a template taking a universal reference with a pair of functions overloaded on lvalue references and rvalue references is likely to incur a runtime cost in some cases. If we generalize the example such that Widget’s data member may be of an arbitrary type (rather than knowing that it’s std::string), the performance gap can widen considerably, because not all types are as cheap to move as std::string (see Item 29).*” C++ realization (temporary object). Advantages of template with universal reference: performance. Book structure (Item 29). Disadvantage of overloading of rvalues and lvalues.*

*Intbu these source code characteristics: volume and idiomaticity: “*The most serious problem with overloading on lvalues and rvalues, however, isn’t the volume or idiomaticity of the source code, nor is it the code’s runtime performance. It’s the poor scalability of the design. Widget::setName takes only one parameter, so only two overloads are necessary, but for functions taking more parameters, each of which could be an lvalue or an rvalue, the number of overloads grows geometrically: n parameters necessitates 2n overloads. And that’s not the worst of it. Some functions —function templates, actually—take an unlimited number of parameters, each of which could be an lvalue or rvalue. The poster children for such functions are std::make\_shared, and, as of C++14, std::make\_unique (see Item 21).*” programming code common disadvantage “poor scalability of the design”. Example of scalability. Book structure (Item 21). Structure of std::make\_share and std::make\_unique.*

*p. 190*

*description of this structure: “*Check out the declarations of their most commonly used overloads:

template<class T, class... Args> // from C++11

shared\_ptr<T> make\_shared(Args&&... args); // Standard

template<class T, class... Args> // from C++14

unique\_ptr<T> make\_unique(Args&&... args); // Standard

For functions like these, overloading on lvalues and rvalues is not an option: universal references are the only way to go. And inside such functions, I assure you, std::forward is applied to the universal reference parameters when they’re passed to other functions. Which is exactly what you should do.*” using context of std::forward.*

Well, usually. Eventually. But…

p. 190

*04.01.2016*

*Using context of std::move and std::forward: “*Well, usually. Eventually. But not necessarily initially. In some cases, you’ll want to use the object bound to an rvalue reference or a universal reference more than once in a single function, and you’ll want to make sure that it’s not moved from until you’re otherwise done with it. In that case, you’ll want to apply std::move (for rvalue references) or std::forward (for universal references) to only the final use of the reference. For example:

template<typename T> // text is

void setSignText(T&& text) // univ. reference

{

sign.setText(text); // use text, but

// don't modify it

auto now = // get current time

std::chrono::system\_clock::now();

signHistory.add(now,

std::forward<T>(text)); // conditionally cast

} // text to rvalue

Here, we want to make sure that text’s value doesn’t get changed by sign.setText, because we want to use that value when we call signHistory.add. Ergo the use of std::forward on only the final use of the universal reference.*” when object moving is necessary only in final situation. Revalue or universal reference says that object can be moved.*

*Std::move\_if\_noexcept using context. Book structure (Item 14): “*For std::move, the same thinking applies (i.e., apply std::move to an rvalue reference the last time it’s used), but it’s important to note that in rare cases, you’ll want to call std::move\_if\_noexcept instead of std::move. To learn when and why, consult Item 14.*”*

*Using context of std::move – reuse of function parameter memory and use moving while returning value from function*

*Example of it: “*If you’re in a function that returns by value, and you’re returning an object bound to an rvalue reference or a universal reference, you’ll want to apply std::move or std::forward when you return the reference. To see why, consider an operator+ function to add two matrices together, where the left-hand matrix is known to be an rvalue (and can hence have its storage reused to hold the sum of the matrices):

Matrix // by-value return

operator+(Matrix&& lhs, const Matrix& rhs)

{

lhs += rhs;

return std::move(lhs); // move lhs into

} // return value

By casting lhs to an rvalue in the return statement (via std::move), lhs will be moved into the function’s return value location. If the call to std::move were omitted,

Matrix // as above

operator+(Matrix&& lhs, const Matrix& rhs)

{

lhs += rhs;

return lhs; // copy lhs into

} // return value*” advantage of moving for program memory size.*

*p. 191*

*advantage of move semantics: more efficient code: “*the fact that lhs is an lvalue would force compilers to instead copy it into the return value location. Assuming that the Matrix type supports move construction, which is more efficient than copy construction, using std::move in the return statement yields more efficient code.*”*

*Moving semantics behavior: “*If Matrix does not support moving, casting it to an rvalue won’t hurt, because the rvalue will simply be copied by Matrix’s copy constructor (see Item 23). If Matrix is later revised to support moving, operator+ will automatically benefit the next time it is compiled. That being the case, there’s nothing to be lost (and possibly much to be gained) by applying std::move to rvalue references being returned from functions that return by value.*”*

*Advantage of std::forward and moving semantics: “*The situation is similar for universal references and std::forward. Consider a function template reduceAndCopy that takes a possibly unreduced Fraction object, reduces it, and then returns a copy of the reduced value. If the original object is an rvalue, its value should be moved into the return value (thus avoiding the expense of making a copy), but if the original is an lvalue, an actual copy must be created. Hence:

template<typename T>

Fraction // by-value return

reduceAndCopy(T&& frac) // universal reference param

{

frac.reduce();

return std::forward<T>(frac); // move rvalue into return

} // value, copy lvalue

If the call to std::forward were omitted, frac would be unconditionally copied into reduceAndCopy’s return value.*”*

*p. 192*

*разнавіднасць памылкі разумення: “*Some programmers take the information above and try to extend it to situations where it doesn’t apply.*”*

*Frequent error of using moving semantics: “*“If using std::move on an rvalue reference parameter being copied into a return value turns a copy construction into a move construction,” they reason, “I can perform the same optimization on local variables that I’m returning.” In other words, they figure that given a function returning a local variable by value, such as this,

Widget makeWidget() // "Copying" version of makeWidget

{

Widget w; // local variable

… // configure w

return w; // "copy" w into return value

}

they can “optimize” it by turning the “copy” into a move:

Widget makeWidget() // Moving version of makeWidget

{

Widget w;

…

return std::move(w); // move w into return value

} // (don't do this!)

My liberal use of quotation marks should tip you off that this line of reasoning is flawed. But why is it flawed? It’s flawed, because the Standardization Committee is way ahead of such programmers when it comes to this kind of optimization. It was recognized long ago that the “copying” version of makeWidget can avoid the need to copy the local variable w by constructing it in the memory alloted for the function’s return value. This is known as the return value optimization (RVO), and it’s been expressly blessed by the C++ Standard for as long as there’s been one.*” С++ feature: RVO – return value optimization.*

*RVO structure: “*Wording such a blessing is finicky business, because you want to permit such copy elision only in places where it won’t affect the observable behavior of the software. Paraphrasing the legalistic (arguably toxic) prose of the Standard, this particular blessing says that compilers may elide the copying (or moving) of a local object2 in a function that returns by value if (1) the type of the local object is the same as that returned by the function and (2) the local object is what’s being returned. With that in mind, look again at the “copying” version of makeWidget:

Widget makeWidget() // "Copying" version of makeWidget

{

Widget w;

…

return w; // "copy" w into return value

}

Both conditions are fulfilled here, and you can trust me when I tell you that for this code, every decent C++ compiler will employ the RVO to avoid copying w. That means that the “copying” version of makeWidget doesn’t, in fact, copy anything.*”*

*C++ structure: eligible local objects: “*Eligible local objects include most local variables (such as w inside makeWidget) as well as temporary objects created as part of a return statement. Function parameters don’t qualify. Some people draw a distinction between application of the RVO to named and unnamed (i.e., temporary) local objects, limiting the term RVO to unnamed objects and calling its application to named objects the named return value optimization (NRVO).*” what are named and unnamed local objects?*

*p. 193*

*situation where moving is less effective: “*The moving version of makeWidget does just what its name says it does (assuming Widget offers a move constructor): it moves the contents of w into makeWidget’s return value location. But why don’t compilers use the RVO to eliminate the move, again constructing w in the memory alloted for the function’s return value? The answer is simple: they can’t. Condition (2) stipulates that the RVO may be performed only if what’s being returned is a local object, but that’s not what the moving version of makeWidget is doing. Look again at its return statement:

return std::move(w);*” nuance of std::move using.*

*Another limitation of using std::move: “*But the RVO is an optimization. Compilers aren’t required to elide copy and move operations, even when they’re permitted to. Maybe you’re paranoid, and you worry that your compilers will punish you with copy operations, just because they can. Or perhaps you’re insightful enough to recognize that there are cases where the RVO is difficult for compilers to implement, e.g., when different control paths in a function return different local variables. (Compilers would have to generate code to construct the appropriate local variable in the memory allotted for the function’s return value, but how could compilers determine which local variable would be appropriate?) If so, you might be willing to pay the price of a move as insurance against the cost of a copy. That is, you might still think it’s reasonable to apply std::move to a local object you’re returning, simply because you’d rest easy knowing you’d never pay for a copy. In that case, applying std::move to a local object would still be a bad idea. The part of the Standard blessing the RVO goes on to say that if the conditions for the RVO are met, but compilers choose not to perform copy elision, the object being returned must be treated as an rvalue. In effect, the Standard requires that when the RVO is permitted, either copy elision takes place or std::move is implicitly applied to local objects being returned.*” C++ structure: compiler treatment of RVO.*

*Example of this: “*So in the “copying” version of makeWidget,

Widget makeWidget() // as before

{

Widget w;

…

return w;

}

compilers must either elide the copying of w or they must treat the function as if it were written like this:

Widget makeWidget()

{

Widget w;

…

return std::move(w); // treat w as rvalue, because

} // no copy elision was performed*”*

*C++ structure: default move semantics using: “*The situation is similar for by-value function parameters. They’re not eligible for copy elision with respect to their function’s return value, but compilers must treat them as rvalues if they’re returned. As a result, if your source code looks like this,

Widget makeWidget(Widget w) // by-value parameter of same

{ // type as function's return

…

return w;

}

compilers must treat it as if it had been written this way:

Widget makeWidget(Widget w)

{

…

return std::move(w); // treat w as rvalue

}*”*

*Nuance of moving semantics using context: “*This means that if you use std::move on a local object being returned from a function that’s returning by value, you can’t help your compilers (they have to treat the local object as an rvalue if they don’t perform copy elision), but you can certainly hinder them (by precluding the RVO).*”*

*Using context of moving semantics: “*There are situations where applying std::move to a local variable can be a reasonable thing to do (i.e., when you’re passing it to a function and you know you won’t be using the variable any longer), but as part of a return statement that would otherwise qualify for the RVO or that returns a byvalue parameter isn’t among them.*” intbu examples of it.*

*p. 195*

*book structure. Resume. Std::move and std::forward using context: “*Things to Remember

• Apply std::move to rvalue references and std::forward to universal references the last time each is used.

• Do the same thing for rvalue references and universal references being returned from functions that return by value.

• Never apply std::move or std::forward to local objects if they would otherwise be eligible for the return value optimization.*”*

### **Item 26: Avoid overloading on universal references.**

*Example of introduction example: trepr structure: “*Suppose you need to write a function that takes a name as a parameter, logs the current date and time, then adds the name to a global data structure. You might come up with a function that looks something like this:

std::multiset<std::string> names; // global data structure

void logAndAdd(const std::string& name)

{

auto now = // get current time

std::chrono::system\_clock::now();

log(now, "logAndAdd"); // make log entry

names.emplace(name); // add name to global data

} // structure; see Item 42

// for info on emplace

This isn’t unreasonable code, but it’s not as efficient as it could be. Consider three potential calls:

std::string petName("Darla");

logAndAdd(petName); // pass lvalue std::string

logAndAdd(std::string("Persephone")); // pass rvalue std::string

logAndAdd("Patty Dog"); // pass string literal

In the first call, logAndAdd’s parameter name is bound to the variable petName. Within logAndAdd, name is ultimately passed to names.emplace. Because name is an lvalue, it is copied into names. There’s no way to avoid that copy, because an lvalue (petName) was passed into logAndAdd.*” lvalue and rvalue examples. String literal is lvalue or rvalue?*

*p. 196*

*example continue: “*In the second call, the parameter name is bound to an rvalue (the temporary std::string explicitly created from "Persephone"). Name itself is an lvalue, so it’s. *” this is actual example of move.*

*Example continue: “*In the third call, the parameter name is again bound to an rvalue, but this time it’s to a temporary std::string that’s implicitly created from "Patty Dog". In the third call, the parameter name is again bound to an rvalue, but this time it’s to a temporary std::string that’s implicitly created from "Patty Dog". As in the second call, name is copied into names, but in this case, the argument originally passed to logAndAdd was a string literal. Had that string literal been passed directly to emplace, there would have been no need to create a temporary std::string at all. Instead, emplace would have used the string literal to create the std::string object directly inside the std::multiset. In this third call, then, we’re paying to copy a std::string, yet there’s really no reason to pay even for a move, much less a copy.*” String literal is rvalue.*

*Std::forward and universal reference using structure. Example continue: “*We can eliminate the inefficiencies in the second and third calls by rewriting logAndAdd to take a universal reference (see Item 24) and, in accord with Item 25, std::forwarding this reference to emplace. The results speak for themselves:

template<typename T>

void logAndAdd(T&& name)

{

auto now = std::chrono::system\_clock::now();

log(now, "logAndAdd");

names.emplace(std::forward<T>(name));

}

std::string petName("Darla"); // as before

logAndAdd(petName); // as before, copy

// lvalue into multiset

logAndAdd(std::string("Persephone")); // move rvalue instead

// of copying it

logAndAdd("Patty Dog"); // create std::string

// in multiset instead

// of copying a temporary

// std::string

Hurray, optimal efficiency!*”*

*p. 197*

*problem formulation: “*Were this the end of the story, we could stop here and proudly retire, but I haven’t told you that clients don’t always have direct access to the names that logAndAdd requires. Some clients have only an index that logAndAdd uses to look up the corresponding name in a table. To support such clients, logAndAdd is overloaded:

std::string nameFromIdx(int idx); // return name

// corresponding to idx

void logAndAdd(int idx) // new overload

{

auto now = std::chrono::system\_clock::now();

log(now, "logAndAdd");

names.emplace(nameFromIdx(idx));

}

Resolution of calls to the two overloads works as expected:

std::string petName("Darla"); // as before

logAndAdd(petName); // as before, these

logAndAdd(std::string("Persephone")); // calls all invoke

logAndAdd("Patty Dog"); // the T&& overload

logAndAdd(22); // calls int overload

Actually, resolution works as expected only if you don’t expect too much. Suppose a client has a short holding an index and passes that to logAndAdd:

short nameIdx;

… // give nameIdx a value

logAndAdd(nameIdx); // error!

The comment on the last line isn’t terribly illuminating, so let me explain what happens here. There are two logAndAdd overloads. The one taking a universal reference can deduce T to be short, thus yielding an exact match. The overload with an int parameter can match the short argument only with a promotion. Per the normal overload resolution rules, an exact match beats a match with a promotion, so the universal reference overload is invoked. Within that overload, the parameter name is bound to the short that’s passed in. Name is then std::forwarded to the emplace member function on names (a std::multiset<std::string>), which, in turn, dutifully forwards it to the std::string constructor. There is no constructor for std::string that takes a short, so the std::string constructor call inside the call to multiset::emplace inside the call to logAndAdd fails. All because the universal reference overload was a better match for a short argument than an int.*” C++ type deducing structure.*

*p. 198*

*function type and universal reference characteristic: “*Functions taking universal references are the greediest functions in C++. They instantiate to create exact matches for almost any type of argument. (The few kinds of arguments where this isn’t the case are described in Item 30.) This is why combining overloading and universal references is almost always a bad idea: the universal reference overload vacuums up far more argument types than the developer doing the overloading generally expects.*” book structure (Item 30). Using context of universal references – to use with overloading is bad idea.*

*Another example demonstating problem: “*An easy way to topple into this pit is to write a perfect forwarding constructor. A small modification to the logAndAdd example demonstrates the problem. Instead of writing a free function that can take either a std::string or an index that can be used to look up a std::string, imagine a class Person with constructors that do the same thing:

class Person {

public:

template<typename T>

explicit Person(T&& n) // perfect forwarding ctor;

: name(std::forward<T>(n)) {} // initializes data member

explicit Person(int idx) // int ctor

: name(nameFromIdx(idx)) {}

…

private:

std::string name;

};

As was the case with logAndAdd, passing an integral type other than int (e.g., std::size\_t, short, long, etc.) will call the universal reference constructor overload instead of the int overload, and that will lead to compilation failures. The problem here is much worse, however, because there’s more overloading present in Person than meets the eye. Item 17 explains that under the appropriate conditions, C++ will generate both copy and move constructors, and this is true even if the class contains a templatized constructor that could be instantiated to produce the signature of the copy or move constructor. If the copy and move constructors for Person are thus generated, Person will effectively look like this:

class Person {

public:

template<typename T> // perfect forwarding ctor

explicit Person(T&& n)

: name(std::forward<T>(n)) {}*” constructor kind – perfect forwarding constructor. Problem connects with default class members (Item 17).*

*p. 199*

*example continue: “*explicit Person(int idx); // int ctor

Person(const Person& rhs); // copy ctor

// (compiler-generated)

Person(Person&& rhs); // move ctor

… // (compiler-generated)

};

This leads to behavior that’s intuitive only if you’ve spent so much time around compilers and compiler-writers, you’ve forgotten what it’s like to be human:

Person p("Nancy");

auto cloneOfP(p); // create new Person from p;

// this won't compile!

Here we’re trying to create a Person from another Person, which seems like about as obvious a case for copy construction as one can get. (p’s an lvalue, so we can banish any thoughts we might have about the “copying” being accomplished through a move operation.) But this code won’t call the copy constructor. It will call the perfectforwarding constructor. That function will then try to initialize Person’s std::string data member with a Person object (p). std::string having no constructor taking a Person, your compilers will throw up their hands in exasperation, possibly punishing you with long and incomprehensible error messages as an expression of their displeasure. “Why,” you might wonder, “does the perfect-forwarding constructor get called instead of the copy constructor? We’re initializing a Person with another Person!” Indeed we are, but compilers are sworn to uphold the rules of C++, and the rules of relevance here are the ones governing the resolution of calls to overloaded functions. Compilers reason as follows. cloneOfP is being initialized with a non-const lvalue (p), and that means that the templatized constructor can be instantiated to take a non-const lvalue of type Person. After such instantiation, the Person class looks like this:

class Person {

public:

explicit Person(Person& n) // instantiated from

: name(std::forward<Person&>(n)) {} // perfect-forwarding

// template

explicit Person(int idx); // as before*”*

*p. 200*

*example continue: “*Person(const Person& rhs); // copy ctor

… // (compiler-generated)

};

In the statement,

auto cloneOfP(p);

p could be passed to either the copy constructor or the instantiated template. Calling the copy constructor would require adding const to p to match the copy constructor’s parameter’s type, but calling the instantiated template requires no such addition. The overload generated from the template is thus a better match, so compilers do what they’re designed to do: generate a call to the better-matching function. “Copying” non-const lvalues of type Person is thus handled by the perfect-forwarding constructor, not the copy constructor. *” compiler structure. And explanation of subtle error.*

*Example continue. another client using example: “*If we change the example slightly so that the object to be copied is const, we hear an entirely different tune:

const Person cp("Nancy"); // object is now const

auto cloneOfP(cp); // calls copy constructor!

Because the object to be copied is now const, it’s an exact match for the parameter taken by the copy constructor. The templatized constructor can be instantiated to have the same signature,

class Person {

public:

explicit Person(const Person& n); // instantiated from

// template

Person(const Person& rhs); // copy ctor

// (compiler-generated)

…

};

because one of the overload-resolution rules in C++ is that in situations where a template instantiation and a non-template function (i.e., a “normal” function) are equally good matches for a function call, the normal function is preferred. The copy constructor (a normal function) thereby trumps an instantiated template with the same signature.*” C++ overload-resolution rules. There were such rules upper.*

*p. 201*

*book structure (Item 17): “*(If you’re wondering why compilers generate a copy constructor when they could instantiate a templatized constructor to get the signature that the copy constructor would have, review Item 17.) *” remind and intbu.*

The interaction among…

p. 201

*06.01.2016*

*C++ subtle interaction – between compiler generated constructors and forwarding constructors.*

*This interaction leads to subtle problems in the case of inheritance: “*The interaction among perfect-forwarding constructors and compiler-generated copy and move operations develops even more wrinkles when inheritance enters the picture. In particular, the conventional implementations of derived class copy and move operations behave quite surprisingly. Here, take a look:

class SpecialPerson: public Person {

public:

SpecialPerson(const SpecialPerson& rhs) // copy ctor; calls

: Person(rhs) // base class

{ … } // forwarding ctor!

SpecialPerson(SpecialPerson&& rhs) // move ctor; calls

: Person(std::move(rhs)) // base class

{ … } // forwarding ctor!

};

As the comments indicate, the derived class copy and move constructors don’t call their base class’s copy and move constructors, they call the base class’s perfectforwarding constructor! To understand why, note that the derived class functions are using arguments of type SpecialPerson to pass to their base class, then work through the template instantiation and overload-resolution consequences for the constructors in class Person. Ultimately, the code won’t compile, because there’s no std::string constructor taking a SpecialPerson.*”*

*Book recommendation: “*I hope that by now I’ve convinced you that overloading on universal reference parameters is something you should avoid if at all possible.*”*

*Enough general programming task: “*But if overloading on universal references is a bad idea, what do you do if you need a function that forwards most argument types, yet needs to treat some argument types in a special fashion?*”*

*Book structure: “*That egg can be unscrambled in a number of ways. So many, in fact, that I’ve devoted an entire Item to them. It’s Item 27. The next Item. Keep reading, you’ll bump right into it.*” (Item 27).*

*Book structure. Resume: “*Things to Remember

• Overloading on universal references almost always leads to the universal reference overload being called more frequently than expected.

• Perfect-forwarding constructors are especially problematic, because they’re typically better matches than copy constructors for non-const lvalues, and they can hijack derived class calls to base class copy and move constructors.*”*

*p. 202*

## Item 27: Familiarize yourself with alternatives to overloading on universal references.

*Book structure and solution for C++ cituation described above: “*Item 26 explains that overloading on universal references can lead to a variety of problems, both for freestanding and for member functions (especially constructors). Yet it also gives examples where such overloading could be useful. If only it would behave the way we’d like! This Item explores ways to achieve the desired behavior, either through designs that avoid overloading on universal references or by employing them in ways that constrain the types of arguments they can match.*” book structure (Item 26).*

*Example of reader orientation in a way of material understanding: “*The discussion that follows builds on the examples introduced in Item 26. If you haven’t read that Item recently, you’ll want to review it before continuing.*”*

### *Abandon overloading*

*Advantages and drawbacks of this method: “*The first example in Item 26, logAndAdd, is representative of the many functions that can avoid the drawbacks of overloading on universal references by simply using different names for the would-be overloads. The two logAndAdd overloads, for example, could be broken into logAndAddName and logAndAddNameIdx. Alas, this approach won’t work for the second example we considered, the Person constructor, because constructor names are fixed by the language. Besides, who wants to give up overloading?*”*

### *Pass by const T&*

*Advantages and drawbacks: “*An alternative is to revert to C++98 and replace pass-by-universal-reference with pass-by-lvalue-reference-to-const. In fact, that’s the first approach Item 26 considers (shown on page 175). The drawback is that the design isn’t as efficient as we’d prefer. Knowing what we now know about the interaction of universal references and overloading, giving up some efficiency to keep things simple might be a more attractive trade-off than it initially appeared.*” less performance but more simplicity and robustness.*

### *Pass by value*

*Book structure (Item 41). Using context of passing object by value (when you know that you’ll copy it): “*An approach that often allows you to dial up performance without any increase in complexity is to replace pass-by-reference parameters with, counterintuitively, pass by value. The design adheres to the advice in Item 41 to consider passing objects by value when you know you’ll copy them, so I’ll defer to that Item for a detailed discussion of how things work and how efficient they are.*”*

*Example of technique: “*Here, I’ll just show how the technique could be used in the Person example:

class Person {

public:

explicit Person(std::string n) // replaces T&& ctor; see

: name(std::move(n)) {} // Item 41 for use of std::move

explicit Person(int idx) // as before

: name(nameFromIdx(idx)) {}

…

private:

std::string name;

};*”*

*p. 203*

*example continue: “*Because there’s no std::string constructor taking only an integer, all int and intlike arguments to a Person constructor (e.g., std::size\_t, short, long) get funneled to the int overload. Similarly, all arguments of type std::string (and things from which std::strings can be created, e.g., literals such as "Ruth") get passed to the constructor taking a std::string. There are thus no surprises for callers. You could argue, I suppose, that some people might be surprised that using 0 or NULL to indicate a null pointer would invoke the int overload, but such people should be referred to Item 8 and required to read it repeatedly until the thought of using 0 or NULL as a null pointer makes them recoil.*” example description. Book structure (Item 8). Nullptr advantage.*

### *Use Tag dispatch*

*Description of this alternative ot overloading C++ technique: “*Neither pass by lvalue-reference-to-const nor pass by value offers support for perfect forwarding. If the motivation for the use of a universal reference is perfect forwarding, we have to use a universal reference; there’s no other choice. Yet we don’t want to abandon overloading. So if we don’t give up overloading and we don’t give up universal references, how can we avoid overloading on universal references? It’s actually not that hard. Calls to overloaded functions are resolved by looking at all the parameters of all the overloads as well as all the arguments at the call site, then choosing the function with the best overall match—taking into account all parameter/argument combinations. A universal reference parameter generally provides an exact match for whatever’s passed in, but if the universal reference is part of a parameter list containing other parameters that are not universal references, sufficiently poor matches on the non-universal reference parameters can knock an overload with a universal reference out of the running. That’s the basis behind the tag dispatch approach, and an example will make the foregoing description easier to understand.*”*

*Good repeating practice of representation: “*We’ll apply tag dispatch to the logAndAdd example on page 177. Here’s the code for that example, lest you get sidetracked looking it up:

std::multiset<std::string> names; // global data structure

template<typename T> // make log entry and add

void logAndAdd(T&& name) // name to data structure

{

auto now = std::chrono::system\_clock::now();

log(now, "logAndAdd");

names.emplace(std::forward<T>(name));

}*” remind this example.*

*p. 204*

*example continue: “*By itself, this function works fine, but were we to introduce the overload taking an int that’s used to look up objects by index, we’d be back in the troubled land of Item 26. The goal of this Item is to avoid that. Rather than adding the overload, we’ll reimplement logAndAdd to delegate to two other functions, one for integral values and one for everything else. logAndAdd itself will accept all argument types, both integral and non-integral.*” tag dispatch technique description.*

*Tag dispatch technique description: “*The two functions doing the real work will be named logAndAddImpl, i.e., we’ll use overloading. One of the functions will take a universal reference. So we’ll have both overloading and universal references. But each function will also take a second parameter, one that indicates whether the argument being passed is integral. This second parameter is what will prevent us from tumbling into the morass described in Item 26, because we’ll arrange it so that the second parameter will be the factor that determines which overload is selected.*” Book structure (Item 26).*

*Example of it: “*Yes, I know, “Blah, blah, blah. Stop talking and show me the code!” No problem. Here’s an almost-correct version of the updated logAndAdd:

template<typename T>

void logAndAdd(T&& name)

{

logAndAddImpl(std::forward<T>(name),

std::is\_integral<T>()); // not quite correct

}

This function forwards its parameter to logAndAddImpl, but it also passes an argument indicating whether that parameter’s type (T) is integral. At least, that’s what it’s supposed to do. For integral arguments that are rvalues, it’s also what it does. But, as Item 28 explains, if an lvalue argument is passed to the universal reference name, the type deduced for T will be an lvalue reference. So if an lvalue of type int is passed to logAndAdd, T will be deduced to be int&. That’s not an integral type, because references aren’t integral types. That means that std::is\_integral<T> will be false for any lvalue argument, even if the argument really does represent an integral value. Recognizing the problem is tantamount to solving it, because the ever-handy Standard C++ Library has a type trait (see Item 9), std::remove\_reference, that does both what its name suggests and what we need: remove any reference qualifiers from a type. The proper way to write logAndAdd is therefore:*” book structure (Item 9). Type trait using context.*

*p. 205*

*example of solution: “*template<typename T>

void logAndAdd(T&& name)

{

logAndAddImpl(

std::forward<T>(name),

std::is\_integral<typename std::remove\_reference<T>::type>()

);

}

This does the trick. (In C++14, you can save a few keystrokes by using std::remove\_reference\_t<T> in place of the highlighted text. For details, see Item 9.) *” book structure (Item 9).*

*Example continue: “*With that taken care of, we can shift our attention to the function being called, logAndAddImpl. There are two overloads, and the first is applicable only to nonintegral types (i.e., to types where std::is\_integral<typename std::remove\_reference<T>::type> is false):

template<typename T> // non-integral

void logAndAddImpl(T&& name, std::false\_type) // argument:

{ // add it to

auto now = std::chrono::system\_clock::now(); // global data

log(now, "logAndAdd"); // structure

names.emplace(std::forward<T>(name));

}*” std::is\_integral<T>, std::false\_type – С++ тыпы. Ведаць іх. Яны дазваляюць цікавым чынам перагружаць функцыі.*

*Std::false\_type and std::true\_type purpose: “*This is straightforward code, once you understand the mechanics behind the highlighted parameter. Conceptually, logAndAdd passes a boolean to logAndAddImpl indicating whether an integral type was passed to logAndAdd, but true and false are runtime values, and we need to use overload resolution—a compile-time phenomenon—to choose the correct logAndAddImpl overload. That means we need a type that corresponds to true and a different type that corresponds to false. This need is common enough that the Standard Library provides what is required under the names std::true\_type and std::false\_type. The argument passed to logAndAd dImpl by logAndAdd is an object of a type that inherits from std::true\_type if T is integral and from std::false\_type if T is not integral. The net result is that this logAndAddImpl overload is a viable candidate for the call in logAndAdd only if T is not an integral type. *”*

*Example continue: “*The second overload covers the opposite case: when T is an integral type. In that event, logAndAddImpl simply finds the name corresponding to the passed-in index and passes that name back to logAndAdd:

std::string nameFromIdx(int idx); // as in Item 26

void logAndAddImpl(int idx, std::true\_type) // integral

{ // argument: look

logAndAdd(nameFromIdx(idx)); // up name and

} // call logAndAdd

// with it

By having logAndAddImpl for an index look up the corresponding name and pass it to logAndAdd (from where it will be std::forwarded to the other logAndAddImpl overload), we avoid the need to put the logging code in both logAndAddImpl overloads.*”*

*p. 206*

*example of pattern “tag dispatch”: “*In this design, the types std::true\_type and std::false\_type are “tags” whose only purpose is to force overload resolution to go the way we want. Notice that we don’t even name those parameters. They serve no purpose at runtime, and in fact we hope that compilers will recognize that the tag parameters are unused and will optimize them out of the program’s execution image. (Some compilers do, at least some of the time.) The call to the overloaded implementation functions inside logAndAdd “dispatches” the work to the correct overload by causing the proper tag object to be created. Hence the name for this design: tag dispatch. It’s a standard building block of template metaprogramming, and the more you look at code inside contemporary C++ libraries, the more often you’ll encounter it.*” technique of template metaprogramming. Example of program entity which doesn’t serve purpose at runtime.*

*Tag dispatch description: “*For our purposes, what’s important about tag dispatch is less how it works and more how it permits us to combine universal references and overloading without the problems described in Item 26. The dispatching function—logAndAdd—takes an unconstrained universal reference parameter, but this function is not overloaded. The implementation functions—logAndAddImpl—are overloaded, and one takes a universal reference parameter, but resolution of calls to these functions depends not just on the universal reference parameter, but also on the tag parameter, and the tag values are designed so that no more than one overload will be a viable match. As a result, it’s the tag that determines which overload gets called. The fact that the universal reference parameter will always generate an exact match for its argument is immaterial.*”*

### *Constraining templates that take universal references*

*Tag disparch using limitation: “*A keystone of tag dispatch is the existence of a single (unoverloaded) function as the client API. This single function dispatches the work to be done to the implementation functions. Creating an unoverloaded dispatch function is usually easy, but the second problem case Item 26 considers, that of a perfect-forwarding constructor for the Person class (shown on page 178), is an exception. Compilers may generate copy and move constructors themselves, so even if you write only one constructor and use tag dispatch within it, some constructor calls may be handled by compiler-generated functions that bypass the tag dispatch system.*”*

*p. 207*

*description of problem with interaction of forwarding constructor and compiler generator special class members: “*In truth, the real problem is not that the compiler-generated functions sometimes bypass the tag dispatch design, it’s that they don’t always pass it by. You virtually always want the copy constructor for a class to handle requests to copy lvalues of that type, but, as Item 26 demonstrates, providing a constructor taking a universal reference causes the universal reference constructor (rather than the copy constructor) to be called when copying non-const lvalues. That Item also explains that when a base class declares a perfect-forwarding constructor, that constructor will typically be called when derived classes implement their copy and move constructors in the conventional fashion, even though the correct behavior is for the base class’s copy and move constructors to be invoked. For situations like these, where an overloaded function taking a universal reference is greedier than you want, yet not greedy enough to act as a single dispatch function, tag dispatch is not the droid you’re looking for. You need a different technique, one that lets you rachet down the conditions under which the function template that the universal reference is part of is permitted to be employed. What you need, my friend, is std::enable\_if.*” using context of std::enable\_if.*

*Description of std::enable\_if: “*std::enable\_if gives you a way to force compilers to behave as if a particular template didn’t exist. Such templates are said to be disabled. By default, all templates are enabled, but a template using std::enable\_if is enabled only if the condition specified by std::enable\_if is satisfied.*”*

*Example of it: “*In our case, we’d like to enable the Person perfect-forwarding constructor only if the type being passed isn’t Person. If the type being passed is Person, we want to disable the perfect-forwarding constructor (i.e., cause compilers to ignore it), because that will cause the class’s copy or move constructor to handle the call, which is what we want when a Person object is initialized with another Person.*”*

*Demonstration example of std::if\_enabled using: “*The way to express that idea isn’t particularly difficult, but the syntax is off-putting, especially if you’ve never seen it before, so I’ll ease you into it. There’s some boilerplate that goes around the condition part of std::enable\_if, so we’ll start with that. Here’s the declaration for the perfect-forwarding constructor in Person, showing only as much of the std::enable\_if as is required simply to use it. I’m showing only the declaration for this constructor, because the use of std::enable\_if has no effect on the function’s implementation. The implementation remains the same as in Item 26.

class Person {

public:

template<typename T,

typename = typename std::enable\_if<condition>::type>

explicit Person(T&& n);

…

};*”*

*p. 208*

*example explanation: “*To understand exactly what’s going on in the highlighted text, I must regretfully suggest that you consult other sources, because the details take a while to explain, and there’s just not enough space for it in this book. (During your research, look into “SFINAE” as well as std::enable\_if, because SFINAE is the technology that makes std::enable\_if work.) Here, I want to focus on expression of the condition that will control whether this constructor is enabled.*”*

### *Невялікае адступленне па SFINAE (substitution failure is not an error):* [*https://habrahabr.ru/post/205772/*](https://habrahabr.ru/post/205772/)

*Сутнасць метапраграмавання: “*Сначала, на всякий случай, очень коротко скажу о метапрограммировании в C++. Метапрограммирование — это операции, производимые во время компиляции программы. Подобно тому, как обычные функции позволяют получать значения, при помощи метафункций получают типы и константы времени компиляции. Одно из наиболее популярных применений метапрограммирования, хотя далеко не единственное — узнавать свойства типов. Всё это, разумеется, применяется при разработке шаблонов: где-то бывает полезно знать, имеем мы дело со сложным пользовательским классом с нетривиальным конструктором или с обычным **int**, где-то необходимо установить, унаследован ли один тип от другого, или можно ли преобразовать один тип в другой. Мы рассмотрим механизм применения SFINAE на классическом примере: проверке того, существует ли в классе функция-член с заданными типами аргументов и возвращаемого значения. Я постараюсь подробно и детально пройти по всем этапам создания проверочной метафункции и проследить откуда что берётся.*”*

*Метафункцыі і метапраграмаванне ў STL: “*Попробуем же придумать, как сделать метафункцию, которая говорит нам, есть ли в каком-то типе метод void foo(int). Заботливая STL, особенно начиная с версии C++11, уже определила для нас много полезных метафункций, размещённых в основном в заголовках [type\_traits](http://en.cppreference.com/w/cpp/types) и [limits](http://en.cppreference.com/w/cpp/header/limits), однако именно такой, какую мы дерзновенно замыслили сделать, там почему-то нет. Метафункция обычно выглядит как шаблонная структура без данных, внутри которой определён результат операции: заданный через **typedef** тип с именем type или статическая константа value. Такого соглашения придерживается STL, и причин оригинальничать у нас нет, поэтому будем придерживаться установленного образца.*”*

*Далей тэкст быў складаны.*

### [*https://en.wikipedia.org/wiki/Substitution\_failure\_is\_not\_an\_error*](https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error)

*“*Although SFINAE was introduced to avoid creating ill-formed programs when unrelated template declarations were visible (e.g., through the inclusion of a header file), many developers found the behavior useful for compile-time introspection. Specifically, it allows a template to determine certain properties of its template arguments at instantiation time.

*”*

*Прыклады кода там зразумеў слаба.*

### [*http://ru.cppreference.com/w/cpp/types/enable\_if*](http://ru.cppreference.com/w/cpp/types/enable_if)

// Включение перегруженных вариантов foo1 при помощи возвращаемого типа

template<class T>

typename std::enable\_if<[std::is\_floating\_point](http://ru.cppreference.com/w/cpp/types/is_floating_point)<T>::value, T>::type

foo1(T t)

{

[std::cout](http://ru.cppreference.com/w/cpp/io/basic_ostream) << "foo1: float**\n**";

return t;

}

*У дадзеным выпадку enable\_if паказвае, што калі Т – мае рэчыўны тып, то гэта функцыя выконваецца.*

// Включение перегруженных вариантов foo2 при помощи дополнительного неиспользуемого параметра

template<class T>

T foo2(T t, typename std::enable\_if<[std::is\_integral](http://ru.cppreference.com/w/cpp/types/is_integral)<T>::value >::type\* = 0)

{

return t;

}

*Гэта я не зразумеў.*

// Включение перегруженных вариантов foo3 при помощи дополнительного параметра шаблона

template<class T ,

class = typename std::enable\_if<[std::is\_integral](http://ru.cppreference.com/w/cpp/types/is_integral)<T>::value>::type >

T foo3(T t) // обратите внимание, сигнатура функции не меняется

{

return t;

}

*І гэта не зразумеў.*

// Включение объявления класса A при помощи дополнительного шаблонного параметра

template<class T, class Enable = void>

class A; // еще неопределенный (undefined) класс

template<class T>

class A<T, typename std::enable\_if<[std::is\_floating\_point](http://ru.cppreference.com/w/cpp/types/is_floating_point)<T>::value >::type> {

};

int main()

{

foo1(1.2); // OK, будет вызвана первая версия foo1()

foo1(10); // OK, будет вызвана вторая версия foo1()

// foo2(0.1); // ошибка времени компиляции

foo2(7); // OK

// A<int> a1; // ошибка времени компиляции

A<double> a1; // OK

}

*І гэта я яшчэ не вельмі добра разумею, нажаль…*

### *Вяртаемся да параграфа Маерса*

*Std::is\_same purpose: “*The condition we want to specify is that T isn’t Person, i.e., that the templatized constructor should be enabled only if T is a type other than Person. Thanks to a type trait that determines whether two types are the same (std::is\_same), it would seem that the condition we want is !std::is\_same<Person, T>::value. (Notice the “!” at the beginning of the expression. We want for Person and T to not be the same.)*”*

*Example description: “*This is close to what we need, but it’s not quite correct, because, as Item 28 explains, the type deduced for a universal reference initialized with an lvalue is always an lvalue reference. That means that for code like this,

Person p("Nancy");

auto cloneOfP(p); // initialize from lvalue

the type T in the universal constructor will be deduced to be Person&. The types Person and Person& are not the same, and the result of std::is\_same will reflect that: std::is\_same<Person, Person&>::value is false.*”*

*Example description: “*If we think more precisely about what we mean when we say that the templatized constructor in Person should be enabled only if T isn’t Person, we’ll realize that when we’re looking at T, we want to ignore

• Whether it’s a reference. For the purpose of determining whether the universal reference constructor should be enabled, the types Person, Person&, and Person&& are all the same as Person.

• Whether it’s const or volatile. As far as we’re concerned, a const Person and a volatile Person and a const volatile Person are all the same as a Person.*”*

*Another type trait and its purposes and structure: std::decay: “*This means we need a way to strip any references, consts, and volatiles from T before checking to see if that type is the same as Person. Once again, the Standard Library gives us what we need in the form of a type trait. That trait is std::decay. std::decay<T>::type is the same as T, except that references and cv-qualifiers (i.e., const or volatile qualifiers) are removed. (I’m fudging the truth here, because std::decay, as its name suggests, also turns array and function types into pointers (see Item 1), but for purposes of this discussion, std::decay behaves as I’ve described.)*” book structure (Item 1)”*

*p. 209*

*example continue: “*The condition we want to control whether our constructor is enabled, then, is

!std::is\_same<Person, typename std::decay<T>::type>::value

i.e., Person is not the same type as T, ignoring any references or cv-qualifiers. (As Item 9 explains, the “typename” in front of std::decay is required, because the type std::decay<T>::type depends on the template parameter T.)*” using context of typename keyword.*

*Example continue: “*Inserting this condition into the std::enable\_if boilerplate above, plus formatting the result to make it easier to see how the pieces fit together, yields this declaration for Person’s perfect-forwarding constructor:

class Person {

public:

template<

typename T,

typename = typename std::enable\_if<

!std::is\_same<Person,

typename std::decay<T>::type

>::value

>::type

>

explicit Person(T&& n);

…

};

If you’ve never seen anything like this before, count your blessings. There’s a reason I saved this design for last.*”*

*Choose between different mechanisms of universal reference parameter functions overloading: “*When you can use one of the other mechanisms to avoid mixing universal references and overloading (and you almost always can), you should.*”*

*Description of enable\_if mechanism: “*Still, once you get used to the functional syntax and the proliferation of angle brackets, it’s not that bad. Furthermore, this gives you the behavior you’ve been striving for. Given the declaration above, constructing a Person from another Person— lvalue or rvalue, const or non-const, volatile or non-volatile—will never invoke the constructor taking a universal reference.*”*

*Another case of such situation which is connected with inheritance: “*Um, no. Belay that celebration. There’s still one loose end from Item 26 that continues to flap about. We need to tie it down. Suppose a class derived from Person implements the copy and move operations in the conventional manner: *” book structure (Item 26).*

*p. 210*

*problem description: “*class SpecialPerson: public Person {

public:

SpecialPerson(const SpecialPerson& rhs) // copy ctor; calls

: Person(rhs) // base class

{ … } // forwarding ctor!

SpecialPerson(SpecialPerson&& rhs) // move ctor; calls

: Person(std::move(rhs)) // base class

{ … } // forwarding ctor!

…

};

This is the same code I showed in Item 26 (on page 206), including the comments, which, alas, remain accurate. When we copy or move a SpecialPerson object, we expect to copy or move its base class parts using the base class’s copy and move constructors, but in these functions, we’re passing SpecialPerson objects to the base class’s constructors, and because SpecialPerson isn’t the same as Person (not even after application of std::decay), the universal reference constructor in the base class is enabled, and it happily instantiates to perform an exact match for a SpecialPerson argument. This exact match is better than the derived-to-base conversions that would be necessary to bind the SpecialPerson objects to the Person parameters in Person’s copy and move constructors, so with the code we have now, copying and moving SpecialPerson objects would use the Person perfect-forwarding constructor to copy or move their base class parts! It’s déjà Item 26 all over again.*”*

The derived class is just…

p. 210

*task description: “*The derived class is just following the normal rules for implementing derived class copy and move constructors, so the fix for this problem is in the base class and, in particular, in the condition that controls whether Person’s universal reference constructor is enabled. We now realize that we don’t want to enable the templatized constructor for any argument type other than Person, we want to enable it for any argument type other than Person or a type derived from Person. Pesky inheritance!*”*

*Type trait std::is\_base\_of: “*You should not be surprised to hear that among the standard type traits is one that determines whether one type is derived from another. It’s called std::is\_base\_of. std::is\_base\_of<T1, 2>::value is true if T2 is derived from T1. Types are considered to be derived from themselves, so std::is\_base\_of<T, T>::value is true.*”*

*Example description: “*This is handy, because we want to revise our condition controlling Person’s perfectforwarding constructor such that the constructor is enabled only if the type T, after stripping it of references and cv-qualifiers, is neither Person nor a class derived from Person. Using std::is\_base\_of instead of std::is\_same gives us what we need:*”*

*p.211*

*example description: “*class Person {

public:

template<

typename T,

typename = typename std::enable\_if<

!std::is\_base\_of<Person,

typename std::decay<T>::type

>::value

>::type

>

explicit Person(T&& n);

…

};*”*

*Example with using alias template declaration: “*Now we’re finally done. Provided we’re writing the code in C++11, that is. If we’re using C++14, this code will still work, but we can employ alias templates for std::enable\_if and std::decay to get rid of the “typename” and “::type” cruft, thus yielding this somewhat more palatable code:

class Person { // C++14

public:

template<

typename T,

typename = std::enable\_if\_t< // less code here

!std::is\_base\_of<Person,

std::decay\_t<T> // and here

>::value

> // and here

>

explicit Person(T&& n);

…

};*”*

*Аспекты трэпр. Гэта цікава чытаць. Гэта дразніць: “*Okay, I admit it: I lied. We’re still not done. But we’re close. Tantalizingly close. Honest.*”*

*Task formulation: “*We’ve seen how to use std::enable\_if to selectively disable Person’s universal reference constructor for argument types we want to have handled by the class’s copy and move constructors, but we haven’t yet seen how to apply it to distinguish integral and non-integral arguments. That was, after all, our original goal; the constructor ambiguity problem was just something we got dragged into along the way.*”*

*p. 212*

*solution of this example: “*All we need to do—and I really do mean that this is everything—is (1) add a Person constructor overload to handle integral arguments and (2) further constrain the templatized constructor so that it’s disabled for such arguments. Pour these ingredients into the pot with everything else we’ve discussed, simmer over a low flame, and savor the aroma of success:

class Person {

public:

template<

typename T,

typename = std::enable\_if\_t<

!std::is\_base\_of<Person, std::decay\_t<T>>::value

&&

!std::is\_integral<std::remove\_reference\_t<T>>::value

>

>

explicit Person(T&& n) // ctor for std::strings and

: name(std::forward<T>(n)) // args convertible to

{ … } // std::strings

explicit Person(int idx) // ctor for integral args

: name(nameFromIdx(idx))

{ … }

… // copy and move ctors, etc.

private:

std::string name;

};*”*

*Using context of this technique: “*Voilà! A thing of beauty! Well, okay, the beauty is perhaps most pronounced for those with something of a template metaprogramming fetish, but the fact remains that this approach not only gets the job done, it does it with unique aplomb. Because it uses perfect forwarding, it offers maximal efficiency, and because it controls the combination of universal references and overloading rather than forbidding it, this technique can be applied in circumstances (such as constructors) where overloading is unavoidable.*”*

### Trade-offs

*Analysis of five previous problem solutions and their classification by criteria whether to specify type or not: “*The first three techniques considered in this Item—abandoning overloading, passing by const T&, and passing by value—specify a type for each parameter in the function(s) to be called. The last two techniques—tag dispatch and constraining template eligibility—use perfect forwarding, hence don’t specify types for the parameters. This fundamental decision—to specify a type or not—has consequences.*”*

*p. 213*

*advantage of perfect forwarding: “*As a rule, perfect forwarding is more efficient, because it avoids the creation of temporary objects solely for the purpose of conforming to the type of a parameter declaration. In the case of the Person constructor, perfect forwarding permits a string literal such as "Nancy" to be forwarded to the constructor for the std::string inside Person, whereas techniques not using perfect forwarding must create a temporary std::string object from the string literal to satisfy the parameter specification for the Person constructor.*” intbu this concept.*

*Drawbacks of perfect forwarding: “*But perfect forwarding has drawbacks. One is that some kinds of arguments can’t be perfect-forwarded, even though they can be passed to functions taking specific types. Item 30 explores these perfect forwarding failure cases. *” book structure (Item 30)*

*Intbu this problem: “*A second issue is the comprehensibility of error messages when clients pass invalid arguments. Suppose, for example, a client creating a Person object passes a string literal made up of char16\_ts (a type introduced in C++11 to represent 16-bit characters) instead of chars (which is what a std::string consists of):

Person p(u"Konrad Zuse"); // "Konrad Zuse" consists of

// characters of type const char16\_t

With the first three approaches examined in this Item, compilers will see that the available constructors take either int or std::string, and they’ll produce a more or less straightforward error message explaining that there’s no conversion from const char16\_t[12] to int or std::string. With an approach based on perfect forwarding, however, the array of const char16\_ts gets bound to the constructor’s parameter without complaint. From there it’s forwarded to the constructor of Person’s std::string data member, and it’s only at that point that the mismatch between what the caller passed in (a const char16\_t array) and what’s required (any type acceptable to the std::string constructor) is discovered. The resulting error message is likely to be, er, impressive. With one of the compilers I use, it’s more than 160 lines long.*”*

*Explanation of problem: “*In this example, the universal reference is forwarded only once (from the Person constructor to the std::string constructor), but the more complex the system, the more likely that a universal reference is forwarded through several layers of function calls before finally arriving at a site that determines whether the argument type(s) are acceptable. The more times the universal reference is forwarded, the more baffling the error message may be when something goes wrong. Many developers find that this issue alone is grounds to reserve universal reference parameters for interfaces where performance is a foremost concern.*”*

*p.214*

*possible solution: “*In the case of Person, we know that the forwarding function’s universal reference parameter is supposed to be an initializer for a std::string, so we can use a*”*

*Type trait std::is\_constructible: “*The std::is\_constructible type trait performs a compile-time test to determine whether an object of one type can be constructed from an object (or set of objects) of a different type (or set of types),*”*

*intbu static\_assert: “*so the assertion is easy to write:

class Person {

public:

template< // as before

typename T,

typename = std::enable\_if\_t<

!std::is\_base\_of<Person, std::decay\_t<T>>::value

&&

!std::is\_integral<std::remove\_reference\_t<T>>::value

>

>

explicit Person(T&& n)

: name(std::forward<T>(n))

{

// assert that a std::string can be created from a T object

static\_assert(

std::is\_constructible<std::string, T>::value,

"Parameter n can't be used to construct a std::string"

);

… // the usual ctor work goes here

}

… // remainder of Person class (as before)

};

This causes the specified error message to be produced if client code tries to create a Person from a type that can’t be used to construct a std::string. Unfortunately, in this example the static\_assert is in the body of the constructor, but the forwarding code, being part of the member initialization list, precedes it. With the compilers I use, the result is that the nice, readable message arising from the static\_assert appears only after the usual error messages (up to 160-plus lines of them) have been emitted.*” static\_assert structure. There are error messages.*

*p. 215*

*Book structure. Resume: “*Things to Remember

• Alternatives to the combination of universal references and overloading include the use of distinct function names, passing parameters by lvaluereferenceto-const, passing parameters by value, and using tag dispatch.

• Constraining templates via std::enable\_if permits the use of universal references and overloading together, but it controls the conditions under which compilers may use the universal reference overloads.

• Universal reference parameters often have efficiency advantages, but they typically have usability disadvantages.*” advantages and drawbacks of universal references.*

### Item 28: Understand reference collapsing.

p. 215

*09/01/2017*

*Encoding of argument type in template function when it is a universal reference: “*Item 23 remarks that when an argument is passed to a template function, the type deduced for the template parameter encodes whether the argument is an lvalue or an rvalue. The Item fails to mention that this happens only when the argument is used to initialize a parameter that’s a universal reference, but there’s a good reason for the omission: universal references aren’t introduced until Item 24. Together, these observations about universal references and lvalue/rvalue encoding mean that for this template,

template<typename T>

void func(T&& param);

the deduced template parameter T will encode whether the argument passed to param was an lvalue or an rvalue.*”*

*Encoding mechanism:”* The encoding mechanism is simple. When an lvalue is passed as an argument, T is deduced to be an lvalue reference. When an rvalue is passed, T is deduced to be a non-reference. (Note the asymmetry: lvalues are encoded as lvalue references, but rvalues are encoded as non-references.)*” lvalue and rvalue aspects.*

*Lvalue and rvalue examples: “*Hence:

Widget widgetFactory(); // function returning rvalue

Widget w; // a variable (an lvalue)

func(w); // call func with lvalue; T deduced

// to be Widget&

func(widgetFactory()); // call func with rvalue; T deduced

// to be Widget*” why function returning type is rvalue?*

*p. 216*

*std::forward and universal reference structure: “*In both calls to func, a Widget is passed, yet because one Widget is an lvalue and one is an rvalue, different types are deduced for the template parameter T. This, as we shall soon see, is what determines whether universal references become rvalue references or lvalue references, and it’s also the underlying mechanism through which std::forward does its work.*”*

*C++ structure. Reference to reference is forbidden: “*Before we can look more closely at std::forward and universal references, we must note that references to references are illegal in C++. Should you try to declare one, your compilers will reprimand you:

int x;

…

auto& & rx = x; // error! can't declare reference to reference*”*

*when rvalue is passed as parameter to function template: “*But consider what happens when an lvalue is passed to a function template taking a universal reference:

template<typename T>

void func(T&& param); // as before

func(w); // invoke func with lvalue;

// T deduced as Widget&

If we take the type deduced for T (i.e., Widget&) and use it to instantiate the template, we get this:

void func(Widget& && param);

A reference to a reference! And yet compilers issue no protest. We know from Item 24 that because the universal reference param is being initialized with an lvalue, param’s type is supposed to be an lvalue reference,*” intbu. Is this error or no?*

*C++ structure. Reference collapsing phenomenon. Intbu what is it? : “*but how does the compiler get from the result of taking the deduced type for T and substituting it into the template to the following, which is the ultimate function signature?

void func(Widget& param);

The answer is reference collapsing. Yes, you are forbidden from declaring references to references, but compilers may produce them in particular contexts, template instantiation being among them. When compilers generate references to references, reference collapsing dictates what happens next.*”*

*There are four reference to reference combinations in C++: “*There are two kinds of references (lvalue and rvalue), so there are four possible reference-reference combinations (lvalue to lvalue, lvalue to rvalue, rvalue to lvalue, and rvalue to rvalue). If a reference to a reference arises in a context where this is permitted (e.g., during template instantiation), the references collapse to a single reference according to this rule:*” template instantiation is context where reference collapsing is possible.*

*p. 217*

*rules of reference collapsing treatment: “*If either reference is an lvalue reference, the result is an lvalue reference. Otherwise (i.e., if both are rvalue references) the result is an rvalue reference.*”*

Intbu this example description: “In our example above, substitution of the deduced type Widget& into the template func yields an rvalue reference to an lvalue reference, and the reference-collapsing rule tells us that the result is an lvalue reference.”

Std::forward work and reference collapsing: “Reference collapsing is a key part of what makes std::forward work. As explained in Item 25, std::forward is applied to universal reference parameters, so a common use case looks like this:

template<typename T>

void f(T&& fParam)

{

… // do some work

someFunc(std::forward<T>(fParam)); // forward fParam to

} // someFunc

Because fParam is a universal reference, we know that the type parameter T will encode whether the argument passed to f (i.e., the expression used to initialize fParam) was an lvalue or an rvalue. std::forward’s job is to cast fParam (an lvalue) to an rvalue if and only if T encodes that the argument passed to f was an rvalue, i.e., if T is a non-reference type. Here’s how std::forward can be implemented to do that:

template<typename T> // in

T&& forward(typename // namespace

remove\_reference<T>::type& param) // std

{

return static\_cast<T&&>(param);

}

This isn’t quite Standards-conformant (I’ve omitted a few interface details), but the differences are irrelevant for the purpose of understanding how std::forward behaves. Suppose that the argument passed to f is an lvalue of type Widget. T will be deduced as Widget&, and the call to std::forward will instantiate as std::forward <Widget&>. Plugging Widget& into the std::forward implementation yields this:”

p. 218

*std::forward work: “*Widget& && forward(typename

remove\_reference<Widget&>::type& param)

{ return static\_cast<Widget& &&>(param); }

The type trait std::remove\_reference<Widget&>::type yields Widget (see Item 9), so std::forward becomes:

Widget& && forward(Widget& param)

{ return static\_cast<Widget& &&>(param); }

Reference collapsing is also applied to the return type and the cast, and the result is the final version of std::forward for the call:

Widget& forward(Widget& param) // still in

{ return static\_cast<Widget&>(param); } // namespace std

As you can see, when an lvalue argument is passed to the function template f, std::forward is instantiated to take and return an lvalue reference. The cast inside std::forward does nothing, because param’s type is already Widget&, so casting it to Widget& has no effect. An lvalue argument passed to std::forward will thus return an lvalue reference. By definition, lvalue references are lvalues, so passing an lvalue to std::forward causes an lvalue to be returned, just like it’s supposed to.*”*

*Std::forward work: “*Now suppose that the argument passed to f is an rvalue of type Widget. In this case, the deduced type for f’s type parameter T will simply be Widget. The call inside f to std::forward will thus be to std::forward<Widget>. Substituting Widget for T in the std::forward implementation gives this:

Widget&& forward(typename

remove\_reference<Widget>::type& param)

{ return static\_cast<Widget&&>(param); }

Applying std::remove\_reference to the non-reference type Widget yields the same type it started with (Widget), so std::forward becomes this:

Widget&& forward(Widget& param)

{ return static\_cast<Widget&&>(param); }

There are no references to references here, so there’s no reference collapsing, and this is the final instantiated version of std::forward for the call.*”*

*Std::forward work and rvalue reference structure: “*Rvalue references returned from functions are defined to be rvalues, so in this case, std::forward will turn f’s parameter fParam (an lvalue) into an rvalue. The end result is that an rvalue argument passed to f will be forwarded to someFunc as an rvalue, which is precisely what is supposed to happen. *” I think function return rvalue because of RVO. Think about.*

*p. 219*

*alias template using example: “*In C++14, the existence of std::remove\_reference\_t makes it possible to implement std::forward a bit more concisely:

template<typename T> // C++14; still in

T&& forward(remove\_reference\_t<T>& param) // namespace std

{

return static\_cast<T&&>(param);

}*”*

*Arising contexts of reference collapsing: “*Reference collapsing occurs in four contexts. The first and most common is template instantiation. The second is type generation for auto variables. The details are essentially the same as for templates, because type deduction for auto variables is essentially the same as type deduction for templates (see Item 2).*” Type deduction in auto.*

*Example of reference collapsing in auto: “*Consider again this example from earlier in the Item:

template<typename T>

void func(T&& param);

Widget widgetFactory(); // function returning rvalue

Widget w; // a variable (an lvalue)

func(w); // call func with lvalue; T deduced

// to be Widget&

func(widgetFactory()); // call func with rvalue; T deduced

// to be Widget

This can be mimicked in auto form. The declaration

auto&& w1 = w;

initializes w1 with an lvalue, thus deducing the type Widget& for auto. Plugging Widget& in for auto in the declaration for w1 yields this reference-to-reference code,

Widget& && w1 = w;

which, after reference collapsing, becomes

Widget& w1 = w;

As a result, w1 is an lvalue reference. On the other hand, this declaration,

auto&& w2 = widgetFactory();

initializes w2 with an rvalue, causing the non-reference type Widget to be deduced for auto. Substituting Widget for auto gives us this:

Widget&& w2 = widgetFactory();

There are no references to references here, so we’re done; w2 is an rvalue reference. *” lvalue and rvalue generation by means of auto.*

*p. 220*

*universal reference definition: “*We’re now in a position to truly understand the universal references introduced in Item 24. A universal reference isn’t a new kind of reference, it’s actually an rvalue reference in a context where two conditions are satisfied: • Type deduction distinguishes lvalues from rvalues. Lvalues of type T are deduced to have type T&, while rvalues of type T yield T as their deduced type.

• Reference collapsing occurs.*”*

*Universal reference advantage: more effective mentality: “*The concept of universal references is useful, because it frees you from having to recognize the existence of reference collapsing contexts, to mentally deduce different types for lvalues and rvalues, and to apply the reference collapsing rule after mentally substituting the deduced types into the contexts in which they occur.*” Mayers is a very big specialist.*

*Contexts of reference collapsing arising: “*I said there were four such contexts, but we’ve discussed only two: template instantiation and auto type generation. The third is the generation and use of typedefs and alias declarations (see Item 9).*”*

*Example of typedef: “*If, during creation or evaluation of a typedef, references to references arise, reference collapsing intervenes to eliminate them. For example, suppose we have a Widget class template with an embedded typedef for an rvalue reference type,

template<typename T>

class Widget {

public:

typedef T&& RvalueRefToT;

…

};

and suppose we instantiate Widget with an lvalue reference type:

Widget<int&> w;

Substituting int& for T in the Widget template gives us the following typedef:

typedef int& && RvalueRefToT;

Reference collapsing reduces it to this,

typedef int& RvalueRefToT;

which makes clear that the name we chose for the typedef is perhaps not as descriptive as we’d hoped: RvalueRefToT is a typedef for an lvalue reference when Widget is instantiated with an lvalue reference type.*”*

*p. 221*

*Arising context of reference collapsing: “*The final context in which reference collapsing takes place is uses of decltype. If, during analysis of a type involving decltype, a reference to a reference arises, reference collapsing will kick in to eliminate it. (For information about decltype, see Item 3.)*”*

*Book structure. Resume: “*Things to Remember

• Reference collapsing occurs in four contexts: template instantiation, auto type generation, creation and use of typedefs and alias declarations, and decltype.

• When compilers generate a reference to a reference in a reference collapsing context, the result becomes a single reference. If either of the original references is an lvalue reference, the result is an lvalue reference. Otherwise it’s an rvalue reference.

• Universal references are rvalue references in contexts where type deduction distinguishes lvalues from rvalues and where reference collapsing occurs.*”*

## Item 29: Assume that move operations are not present, not cheap, and not used

*Interesting Mayers recommendation.*

*C++ history: “*Move semantics is arguably the premier feature of C++11.*”*

*Moving semantics advantages: “*“Moving containers is now as cheap as copying pointers!” you’re likely to hear, and “Copying temporary objects is now so efficient, coding to avoid it is tantamount to premature optimization!” Such sentiments are easy to understand. Move semantics is truly an important feature. It doesn’t just allow compilers to replace expensive copy operations with comparatively cheap moves, it actually requires that they do so (when the proper conditions are fulfilled). Take your C++98 code base, recompile with a C++11-conformant compiler and Standard Library, and—shazam!—your software runs faster.*” There is no copies of temporary objects. Copying of containers is so cheap as copying of pointers.*

*Book structure and some thinking effect which is used my marketers and scammers: “*Move semantics can really pull that off, and that grants the feature an aura worthy of legend. Legends, however, are generally the result of exaggeration. The purpose of this Item is to keep your expectations grounded.*”*

*Move semantics limitations. C++ history: “*Let’s begin with the observation that many types fail to support move semantics. The entire C++98 Standard Library was overhauled for C++11 to add move operations for types where moving could be implemented faster than copying, and the implementation of the library components was revised to take advantage of these operations, but chances are that you’re working with a code base that has not been completely revised to take advantage of C++11. For types in your applications (or in the libraries you use) where no modifications for C++11 have been made, the existence of move support in your compilers is likely to do you little good. True, C++11 is willing to generate move operations for classes that lack them, but that happens only for classes declaring no copy operations, move operations, or destructors (see Item 17). Data members or base classes of types that have disabled moving (e.g., by deleting the move operations—see Item 11) will also suppress compiler-generated move operations. For types without explicit support for moving and that don’t qualify for compiler-generated move operations, there is no reason to expect C++11 to deliver any kind of performance improvement over C++98.*”*

*p. 222*

*STL containers property and move semantics nuances: “*Even types with explicit move support may not benefit as much as you’d hope. All containers in the standard C++11 library support moving, for example, but it would be a mistake to assume that moving all containers is cheap. For some containers, this is because there’s no truly cheap way to move their contents. For others, it’s because the truly cheap move operations the containers offer come with caveats the container elements can’t satisfy.*”*

*Std::array – new container in C++11. Example of this nuance: “*Consider std::array, a new container in C++11. std::array is essentially a builtin array with an STL interface. This is fundamentally different from the other standard containers, each of which stores its contents on the heap. Objects of such container types hold (as data members), conceptually, only a pointer to the heap memory storing the contents of the container. (The reality is more complex, but for purposes of this analysis, the differences are not important.) The existence of this pointer makes it possible to move the contents of an entire container in constant time: just copy the pointer to the container’s contents from the source container to the target, and set the source’s pointer to null:

std::vector<Widget> vw1;

// put data into vw1

…

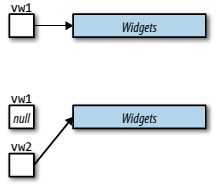
// move vw1 into vw2. Runs in

// constant time. Only ptrs

// in vw1 and vw2 are modified

auto vw2 = std::move(vw1);

*” there is image additionally.*

*moving of a vector.*

*p. 223*

*continue of std::array description and problems of moving it: “*std::array objects lack such a pointer, because the data for a std::array’s contents are stored directly in the std::array object:

std::array<Widget, 10000> aw1;

// put data into aw1

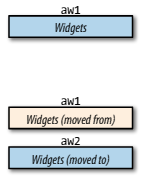
…

// move aw1 into aw2. Runs in

// linear time. All elements in

// aw1 are moved into aw2

auto aw2 = std::move(aw1);

**

Note that the elements in aw1 are moved into aw2. Assuming that Widget is a type where moving is faster than copying, moving a std::array of Widget will be faster than copying the same std::array. So std::array certainly offers move support. Yet both moving and copying a std::array have linear-time computational complexity, because each element in the container must be copied or moved. This is far from the “moving a container is now as cheap as assigning a couple of pointers” claim that one sometimes hears.*” there is no moving advantage in performance. Only in memory. Example where type of container elements defines if moving is cheaper that copying.*

*Move semantics realization and std::string structure (SSO – small string optimization, and moving complexity): “*On the other hand, std::string offers constant-time moves and linear-time copies. That makes it sound like moving is faster than copying, but that may not be the case. Many string implementations employ the small string optimization (SSO). With the SSO, “small” strings (e.g., those with a capacity of no more than 15 characters) are stored in a buffer within the std::string object; no heap-allocated storage is used. Moving small strings using an SSO-based implementation is no faster than copying them, because the copy-only-a-pointer trick that generally underlies the performance advantage of moves over copies isn’t applicable. The motivation for the SSO is extensive evidence that short strings are the norm for many applications. Using an internal buffer to store the contents of such strings eliminates the need to dynamically allocate memory for them, and that’s typically an efficiency win. An implication of the win, however, is that moves are no faster than copies, though one could just as well take a glass-half-full approach and say that for such strings, copying is no slower than moving.*” SSO is some static buffer with commonly used strings?*

*Moving semantics nuances. Book structure (Item 14): “*Even for types supporting speedy move operations, some seemingly sure-fire move situations can end up making copies. Item 14 explains that some container operations in the Standard Library offer the strong exception safety guarantee and that to ensure that legacy C++98 code dependent on that guarantee isn’t broken when upgrading to C++11, the underlying copy operations may be replaced with move operations only if the move operations are known to not throw. A consequence is that even if a type offers move operations that are more efficient than the corresponding copy operations, and even if, at a particular point in the code, a move operation would generally be appropriate (e.g., if the source object is an rvalue), compilers might still be forced to invoke a copy operation because the corresponding move operation isn’t declared noexcept.*”*

*p. 224*

*moving semantics limitations: “*There are thus several scenarios in which C++11’s move semantics do you no good:

• No move operations: The object to be moved from fails to offer move operations. The move request therefore becomes a copy request.

• Move not faster: The object to be moved from has move operations that are no faster than its copy operations.

• Move not usable: The context in which the moving would take place requires a move operation that emits no exceptions, but that operation isn’t declared noexcept. It’s worth mentioning, too, another scenario where move semantics offers no efficiency gain:

• Source object is lvalue: With very few exceptions (see e.g., Item 25) only rvalues may be used as the source of a move operation.*” book structure (Item 25). What exceptions are referring to?*

*Book recommendation concerning writing templates and unstable code. Kind of code instability – frequent modification of types: “*But the title of this Item is to assume that move operations are not present, not cheap, and not used. This is typically the case in generic code, e.g., when writing templates, because you don’t know all the types you’re working with. In such circumstances, you must be as conservative about copying objects as you were in C++98—before move semantics existed. This is also the case for “unstable” code, i.e., code where the characteristics of the types being used are subject to relatively frequent modification.*”*

*move semantics using context and book recommendation: “*Often, however, you know the types your code uses, and you can rely on their characteristics not changing (e.g., whether they support inexpensive move operations). When that’s the case, you don’t need to make assumptions. You can simply look up the move support details for the types you’re using. If those types offer cheap move operations, and if you’re using objects in contexts where those move operations will be invoked, you can safely rely on move semantics to replace copy operations with their less expensive move counterparts.*”*

*Book structure. Resume: “*Things to Remember

• Assume that move operations are not present, not cheap, and not used.

• In code with known types or support for move semantics, there is no need for assumptions.*”*

*p. 225*

## Item 30: Familiarize yourself with perfect forwarding failure cases.

*Is this example of studying by good samples like Brooks proposal?*

*C++ history and structure: perfect forwarding: “*One of the features most prominently emblazoned on the C++11 box is perfect forwarding. Perfect forwarding. It’s perfect! Alas, tear the box open, and you’ll find that there’s “perfect” (the ideal), and then there’s “perfect” (the reality). C++11’s perfect forwarding is very good, but it achieves true perfection only if you’re willing to overlook an epsilon or two. This Item is devoted to familiarizing you with the epsilons.*” intbu this translation. What is epcilon?*

*Definition of forwarding. It is connected with function: “*Before embarking on our epsilon exploration, it’s worthwhile to review what’s meant by “perfect forwarding.” “Forwarding” just means that one function passes—forwards —its parameters to another function. The goal is for the second function (the one

being forwarded to) to receive the same objects that the first function (the one doing the forwarding) received. That rules out by-value parameters, because they’re copies of what the original caller passed in. We want the forwarded-to function to be able to work with the originally-passed-in objects. Pointer parameters are also ruled out, because we don’t want to force callers to pass pointers. When it comes to generalpurpose forwarding, we’ll be dealing with parameters that are references.*” Forwarding usually concerns function parameters which are references.*

*Perfect forwarding definition and universal reference purpose and feature: “*Perfect forwarding means we don’t just forward objects, we also forward their salient characteristics: their types, whether they’re lvalues or rvalues, and whether they’re const or volatile. In conjunction with the observation that we’ll be dealing with reference parameters, this implies that we’ll be using universal references (see Item 24), because only universal reference parameters encode information about the lvalueness and rvalueness of the arguments that are passed to them.*”*

*Example of perfect forwarding: “*Let’s assume we have some function f, and we’d like to write a function (in truth, a function template) that forwards to it. The core of what we need looks like this:

template<typename T>

void fwd(T&& param) // accept any argument

{

f(std::forward<T>(param)); // forward it to f

}

Forwarding functions are, by their nature, generic. The fwd template, for example, accepts any type of argument, and it forwards whatever it gets. A logical extension of this genericity is for forwarding functions to be not just templates, but variadic templates, thus accepting any number of arguments. The variadic form for fwd looks like this:

template<typename... Ts>

void fwd(Ts&&... params) // accept any arguments

{

f(std::forward<Ts>(params)...); // forward them to f

}*” Type of functions – forwarding functions. They are usually generic. Variadic templates using context.*

*p. 226*

*examples of forwarding functions: “*This is the form you’ll see in, among other places, the standard containers’ emplacement functions (see Item 42) and the smart pointer factory functions, std::make\_shared and std::make\_unique (see Item 21).*”*

*Perfect forwarding failure example: “*Given our target function f and our forwarding function fwd, perfect forwarding fails if calling f with a particular argument does one thing, but calling fwd with the same argument does something different:

f( expression ); // if this does one thing,

fwd( expression ); // but this does something else, fwd fails

// to perfectly forward expression to f*”*

*Situations of this kinf of failure will be discussed: “*Several kinds of arguments lead to this kind of failure. Knowing what they are and how to work around them is important, so let’s tour the kinds of arguments that can’t be perfect-forwarded. *” C++ structure and perfect forwarding limitation.*

### Braced initializers

*There is braced initializers structure.*

*p. 226*

*10.01.2017*

*Problem demonstration with example: “*Suppose f is declared like this:

void f(const std::vector<int>& v);

In that case, calling f with a braced initializer compiles,

f({ 1, 2, 3 }); // fine, "{1, 2, 3}" implicitly

// converted to std::vector<int>

but passing the same braced initializer to fwd doesn’t compile:

fwd({ 1, 2, 3 }); // error! doesn't compile

That’s because the use of a braced initializer is a perfect forwarding failure case. *”*

*Example explanation and compiler work with function and function template calling: “*All such failure cases have the same cause. In a direct call to f (such as f({ 1, 2, 3 })), compilers see the arguments passed at the call site, and they see the types of the parameters declared by f. They compare the arguments at the call site to the parameter declarations to see if they’re compatible, and, if necessary, they perform implicit conversions to make the call succeed. In the example above, they generate a temporary std::vector<int> object from { 1, 2, 3 } so that f’s parameter v has a std::vector<int> object to bind to. When calling f indirectly through the forwarding function template fwd, compilers no longer compare the arguments passed at fwd’s call site to the parameter declara- tions in f. Instead, they deduce the types of the arguments being passed to fwd, and they compare the deduced types to f’s parameter declarations.*”*

*p. 227*

*Reasons of perfect forwarding failure: “*Perfect forwarding fails when either of the following occurs: • Compilers are unable to deduce a type for one or more of fwd’s parameters. In this case, the code fails to compile.

• Compilers deduce the “wrong” type for one or more of fwd’s parameters. Here, “wrong” could mean that fwd’s instantiation won’t compile with the types that were deduced, but it could also mean that the call to f using fwd’s deduced types behaves differently from a direct call to f with the arguments that were passed to fwd. One source of such divergent behavior would be if f were an overloaded function name, and, due to “incorrect” type deduction, the overload of f called inside fwd were different from the overload that would be invoked if f were called directly. *” the reasons are type deduction and functions overloading.*

*Example explanation and C++ structure. Usgin context of std::initializer\_list: “*In the “fwd({ 1, 2, 3 })” call above, the problem is that passing a braced initializer to a function template parameter that’s not declared to be a std::initializer\_list is decreed to be, as the Standard puts it, a “non-deduced context.” In plain English, that means that compilers are forbidden from deducing a type for the expression { 1, 2, 3 } in the call to fwd, because fwd’s parameter isn’t declared to be a std::initializer\_list. Being prevented from deducing a type for fwd’s parame- ter, compilers must understandably reject the call.*”*

*Intbu this solution. Book structure (Item 2): “*Interestingly, Item 2 explains that type deduction succeeds for auto variables initialized with a braced initializer. Such variables are deemed to be std::initializer\_list objects, and this affords a simple workaround for cases where the type the forwarding function should deduce is a std::initializer\_list—declare a local variable using auto, then pass the local variable to the forwarding function:

auto il = { 1, 2, 3 }; // il's type deduced to be

// std::initializer\_list<int>

fwd(il); // fine, perfect-forwards il to f*”*

### 0 or NULL as null pointers

*Problem formulation. Nullptr using context. C++ structure. Type deducing: “*Item 8 explains that when you try to pass 0 or NULL as a null pointer to a template, type deduction goes awry, deducing an integral type (typically int) instead of a pointer type for the argument you pass. The result is that neither 0 nor NULL can be perfect-forwarded as a null pointer. The fix is easy, however: pass nullptr instead of 0 or NULL. For details, consult Item 8. *”*

*p. 228*

### Declaration-only integral static const data members

*C++ feature “const propagation”. C++ practice rule: “*As a general rule, there’s no need to define integral static const data members in classes; declarations alone suffice. That’s because compilers perform const propagation on such members’ values, thus eliminating the need to set aside memory for them. For example, consider this code:

class Widget {

public:

static const std::size\_t MinVals = 28; // MinVals' declaration

…

};

… // no defn. for MinVals

std::vector<int> widgetData;

widgetData.reserve(Widget::MinVals); // use of MinVals *” there is no need to define static const data members.*

*Subtle problem concerning pointer to static const memebers. Compiler work: “*Here, we’re using Widget::MinVals (henceforth simply MinVals) to specify widget Data’s initial capacity, even though MinVals lacks a definition. Compilers work around the missing definition (as they are required to do) by plopping the value 28 into all places where MinVals is mentioned. The fact that no storage has been set aside for MinVals’ value is unproblematic. If MinVals’ address were to be taken (e.g., if somebody created a pointer to MinVals), then MinVals would require storage (so that the pointer had something to point to), and the code above, though it would compile, would fail at link-time until a definition for MinVals was provided.*” Program compilation structure.*

*Problem demonstration on the example: “*With that in mind, imagine that f (the function fwd forwards its argument to) is declared like this:

void f(std::size\_t val);

Calling f with MinVals is fine, because compilers will just replace MinVals with its value:

f(Widget::MinVals); // fine, treated as "f(28)"

Alas, things may not go so smoothly if we try to call f through fwd:

fwd(Widget::MinVals); // error! shouldn't link

This code will compile, but it shouldn’t link. If that reminds you of what happens if we write code that takes MinVals’ address, that’s good, because the underlying problem is the same.*”*

*Problem explanation and C++ structure: treatment of references: “*Although nothing in the source code takes MinVals’ address, fwd’s parameter is a universal reference, and references, in the code generated by compilers, are usually treated like pointers. In the program’s underlying binary code (and on the hardware), pointers and references are essentially the same thing. At this level, there’s truth to the adage that references are simply pointers that are automatically dereferenced. That being the case, passing MinVals by reference is effectively the same as passing it by pointer, and as such, there has to be some memory for the pointer to point to. Passing integral static const data members by reference, then, generally requires that they be defined, and that requirement can cause code using perfect forwarding to fail where the equivalent code without perfect forwarding succeeds.*”*

*p. 229*

*C++ structure. Code correctness of which depends on compiler realization: “*But perhaps you noticed the weasel words I sprinkled through the preceding discussion. The code “shouldn’t” link. References are “usually” treated like pointers. Passing integral static const data members by reference “generally” requires that they be defined. It’s almost like I know something I don’t really want to tell you… That’s because I do. According to the Standard, passing MinVals by reference requires that it be defined. But not all implementations enforce this requirement. So, depending on your compilers and linkers, you may find that you can perfect-forward integral static const data members that haven’t been defined. If you do, congratulations, but there is no reason to expect such code to port. To make it portable, simply provide a definition for the integral static const data member in question. For MinVals, that’d look like this:

const std::size\_t Widget::MinVals; // in Widget's .cpp file

Note that the definition doesn’t repeat the initializer (28, in the case of MinVals). Don’t stress over this detail, however. If you forget and provide the initializer in both places, your compilers will complain, thus reminding you to specify it only once.*”*

### Overloaded function names and template names

*Problem demonstration on example: “*Suppose our function f (the one we keep wanting to forward arguments to via fwd) can have its behavior customized by passing it a function that does some of its work. Assuming this function takes and returns ints, f could be declared like this:

void f(int (\*pf)(int)); // pf = "processing function"

It’s worth noting that f could also be declared using a simpler non-pointer syntax. Such a declaration would look like this, though it’d have the same meaning as the declaration above:

void f(int pf(int)); // declares same f as above

Either way, now suppose we have an overloaded function, processVal:

int processVal(int value);

int processVal(int value, int priority);

We can pass processVal to f,

f(processVal); // fine

but it’s something of a surprise that we can. f demands a pointer to a function as its argument, but processVal isn’t a function pointer or even a function, it’s the name of two different functions. However, compilers know which processVal they need: the one matching f’s parameter type. They thus choose the processVal taking one int, and they pass that function’s address to f. *” C++ structure. Example with function pointer.*

*p. 230*

*example continue and error description: “*What makes this work is that f’s declaration lets compilers figure out which version of processVal is required. fwd, however, being a function template, doesn’t have any information about what type it needs, and that makes it impossible for compilers to determine which overload should be passed:

fwd(processVal); // error! which processVal?

processVal alone has no type. Without a type, there can be no type deduction, and without type deduction, we’re left with another perfect forwarding failure case. *”*

*Another example of the same problem: “*The same problem arises if we try to use a function template instead of (or in addition to) an overloaded function name. A function template doesn’t represent one function, it represents many functions:

template<typename T>

T workOnVal(T param) // template for processing values

{ … }

fwd(workOnVal); // error! which workOnVal

// instantiation?*” function template structure and fundamental difference from function.*

*Solution of problem and example of type deduction and using of pointer function type: “*The way to get a perfect-forwarding function like fwd to accept an overloaded function name or a template name is to manually specify the overload or instantiation you want to have forwarded. For example, you can create a function pointer of the same type as f’s parameter, initialize that pointer with processVal or workOnVal (thus causing the proper version of processVal to be selected or the proper instantiation of workOnVal to be generated), and pass the pointer to fwd:

using ProcessFuncType = // make typedef;

int (\*)(int); // see Item 9

ProcessFuncType processValPtr = processVal; // specify needed

// signature for

// processVal

fwd(processValPtr); // fine

fwd(static\_cast<ProcessFuncType>(workOnVal)); // also fine*”*

*p. 231*

*perfect forwarding pattern and its quality marker: “*Of course, this requires that you know the type of function pointer that fwd is forwarding to. It’s not unreasonable to assume that a perfect-forwarding function will document that. After all, perfect-forwarding functions are designed to accept anything, so if there’s no documentation telling you what to pass, how would you know?*”*

### *Bitfields*

*Intbu this aspect: “*This assumes that bitfields are laid out lsb (least significant bit) to msb (most significant bit). C++ doesn’t guarantee that, but compilers often provide a mechanism that allows programmers to control bitfield layout.*”*

*Problem demonstration on example: “*The final failure case for perfect forwarding is when a bitfield is used as a function argument. To see what this means in practice, observe that an IPv4 header can be modeled as follows:3

struct IPv4Header {

std::uint32\_t version:4,

IHL:4,

DSCP:6,

ECN:2,

totalLength:16;

…

};

If our long-suffering function f (the perennial target of our forwarding function fwd) is declared to take a std::size\_t parameter, calling it with, say, the totalLength field of an IPv4Header object compiles without fuss:

void f(std::size\_t sz); // function to call

IPv4Header h;

…

f(h.totalLength); // fine

Trying to forward h.totalLength to f via fwd, however, is a different story:

fwd(h.totalLength); // error!

The problem is that fwd’s parameter is a reference, and h.totalLength is a nonconst bitfield. *” good style of placing together variables with the same type.*

*C++ structure: “*That may not sound so bad, but the C++ Standard condemns the combination in unusually clear prose: “A non-const reference shall not be bound to a bit-field.” There’s an excellent reason for the prohibition. Bitfields may consist of arbitrary parts of machine words (e.g., bits 3-5 of a 32-bit int), but there’s no way to directly address such things. I mentioned earlier that references and pointers are the same thing at the hardware level, and just as there’s no way to create a pointer to arbitrary bits (C++ dictates that the smallest thing you can point to is a char), there’s no way to bind a reference to arbitrary bits, either. *” C++ structure and realization. Sameness of pointers and references on the low level. Does std::uint32\_t indicate bit?*

*p. 232*

*C++ structure. Work with bitfields. Pointer realization: “*Working around the impossibility of perfect-forwarding a bitfield is easy, once you realize that any function that accepts a bitfield as an argument will receive a copy of the bitfield’s value. After all, no function can bind a reference to a bitfield, nor can any function accept pointers to bitfields, because pointers to bitfields don’t exist. The only kinds of parameters to which a bitfield can be passed are by-value parameters and, interestingly, references-to-const. In the case of by-value parameters, the called function obviously receives a copy of the value in the bitfield, and it turns out that in the case of a reference-to-const parameter, the Standard requires that the reference actually bind to a copy of the bitfield’s value that’s stored in an object of some standard integral type (e.g., int). References-to-const don’t bind to bitfields, they bind to “normal” objects into which the values of the bitfields have been copied.*” C++ demands and possibilities concerning bitfields.*

*Solution of the problem: “*The key to passing a bitfield into a perfect-forwarding function, then, is to take advantage of the fact that the forwarded-to function will always receive a copy of the bitfield’s value. You can thus make a copy yourself and call the forwarding function with the copy. In the case of our example with IPv4Header, this code would do the trick:

// copy bitfield value; see Item 6 for info on init. form

auto length = static\_cast<std::uint16\_t>(h.totalLength);

fwd(length); // forward the copy*”*

### Upshot

*Perfect forwarding structure (work with it and nuances): “*In most cases, perfect forwarding works exactly as advertised. You rarely have to think about it. But when it doesn’t work—when reasonable-looking code fails to compile or, worse, compiles, but doesn’t behave the way you anticipate—it’s important to know about perfect forwarding’s imperfections. Equally important is knowing how to work around them. In most cases, this is straightforward.*”*

*Book structure. Resume: “*Things to Remember

• Perfect forwarding fails when template type deduction fails or when it deduces the wrong type.

• The kinds of arguments that lead to perfect forwarding failure are braced initializers, null pointers expressed as 0 or NULL, declaration-only integral const static data members, template and overloaded function names, and bitfields.*”*

*About bitfields:* [*http://en.cppreference.com/w/cpp/language/bit\_field*](http://en.cppreference.com/w/cpp/language/bit_field)

# CHAPTER 6 Lambda Expressions

*Influence, advantage, essentials and purpose of lambda expressions. They represent function objects: “*Lambda expressions—lambdas—are a game changer in C++ programming. That’s somewhat surprising, because they bring no new expressive power to the language. Everything a lambda can do is something you can do by hand with a bit more typing. But lambdas are such a convenient way to create function objects, the impact on daytoday C++ software development is enormous.*”*

*STL structure \_if algorithms and predicates. Using context of lambdas: “*Without lambdas, the STL “\_if” algorithms (e.g., std::find\_if, std::remove\_if, std::count\_if, etc.) tend to be employed with only the most trivial predicates, but when lambdas are available, use of these algorithms with nontrivial conditions blossoms.*” Kind of algorithms in C++ and meaning of lambda expressions: “*The same is true of algorithms that can be customized with comparison functions (e.g., std::sort, std::nth\_element, std::lower\_bound, etc.).*”*

*using context of lambda expressions. Smart pointers. Condition variables in the threading API: “*Outside the STL, lambdas make it possible to quickly create custom deleters for std::unique\_ptr and std::shared\_ptr (see Items 18 and 19), and they make the specification of predicates for condition variables in the threading API equally straightforward (see Item 39).*”*

*Intbu this lambdas using context: “*Beyond the Standard Library, lambdas facilitate the on-the-fly specification of callback functions, interface adaption functions, and context-specific functions for one-off calls.*”*

*Lambdas changed C++ programming style: “*Lambdas really make C++ a more pleasant programming language.*” C++ history.*

*Lambda expression definition: “*The vocabulary associated with lambdas can be confusing. Here’s a brief refresher:

• A lambda expression is just that: an expression. It’s part of the source code. In

std::find\_if(container.begin(), container.end(),

[](int val) { return 0 < val && val < 10; });

the highlighted expression is the lambda.*”*

*Lambda expression structure: definition of “closure”: “*A closure is the runtime object created by a lambda. Depending on the capture mode, closures hold copies of or references to the captured data. In the call to std::find\_if above, the closure is the object that’s passed at runtime as the third argument to std::find\_if. *” what is capture mode? Example of closure.*

*Lambda expression structure and compiler work: closure class: “*A closure class is a class from which a closure is instantiated. Each lambda causes compilers to generate a unique closure class. The statements inside a lambda become executable instructions in the member functions of its closure class. *”*

*Using context of lambdas: “*A lambda is often used to create a closure that’s used only as an argument to a function. That’s the case in the call to std::find\_if above.*”*

*Closure property: “*However, closures may generally be copied, so it’s usually possible to have multiple closures of a closure type corresponding to a single lambda. For example, in the following code,

{

int x; // x is local variable

…

auto c1 = // c1 is copy of the

[x](int y) { return x \* y > 55; }; // closure produced

// by the lambda

auto c2 = c1; // c2 is copy of c1

auto c3 = c2; // c3 is copy of c2

…

}

c1, c2, and c3 are all copies of the closure produced by the lambda. Informally, it’s perfectly acceptable to blur the lines between lambdas, closures, and closure classes. But in the Items that follow, it’s often important to distinguish what exists during compilation (lambdas and closure classes), what exists at runtime (closures), and how they relate to one another. *” book structure. Lambdas structure.*

*This knowledge have two racurses: what exists during compilation and what exist during runtime.*

## Item 31: Avoid default capture modes.

*C++ structure: capture modes: “*There are two default capture modes in C++11: by-reference and by-value. Default by-reference capture can lead to dangling references. Default by-value capture lures you into thinking you’re immune to that problem (you’re not), and it lulls you into thinking your closures are self-contained (they may not be).*” Dangers connected with them. Intbu what is capture modes?*

*Book structure and Mayers language: “*That’s the executive summary for this Item. If you’re more engineer than executive, you’ll want some meat on those bones, so let’s start with the danger of default byreference capture.*”*

*p. 235*

*problem of dangling references in by-reference capture: “*A by-reference capture causes a closure to contain a reference to a local variable or to a parameter that’s available in the scope where the lambda is defined. If the lifetime of a closure created from that lambda exceeds the lifetime of the local variable or parameter, the reference in the closure will dangle.*”*

*Example of the problem: “*For example, suppose we have a container of filtering functions, each of which takes an int and returns a bool indicating whether a passed-in value satisfies the filter:

using FilterContainer = // see Item 9 for

std::vector<std::function<bool(int)>>; // "using", Item 2

// for std::function

FilterContainer filters; // filtering funcs

We could add a filter for multiples of 5 like this:

filters.emplace\_back( // see Item 42 for

[](int value) { return value % 5 == 0; } // info on

); // emplace\_back

However, it may be that we need to compute the divisor at runtime, i.e., we can’t just hard-code 5 into the lambda. So adding the filter might look more like this:

void addDivisorFilter()

{

auto calc1 = computeSomeValue1();

auto calc2 = computeSomeValue2();

auto divisor = computeDivisor(calc1, calc2);

filters.emplace\_back( // danger!

[&](int value) { return value % divisor == 0; } // ref to

); // divisor

} // will

// dangle!

This code is a problem waiting to happen. The lambda refers to the local variable divisor, but that variable ceases to exist when addDivisorFilter returns. That’s immediately after filters.emplace\_back returns, so the function that’s added to filters is essentially dead on arrival. Using that filter yields undefined behavior from virtually the moment it’s created. *” what is std::function? What is std::emplace\_back? Comment: “We add function to the vector but this function contains variable which doesn’t exist. It’s rubbish…. Big problem.”*

*Example of explicit by-reference capture. There is gangling problem too: “*Now, the same problem would exist if divisor’s by-reference capture were explicit,

filters.emplace\_back(

[&divisor](int value) // danger! ref to

{ return value % divisor == 0; } // divisor will

); // still dangle!*”*

*For understanding what is capture. It’s capture of local variables to lambda: “*but with an explicit capture, it’s easier to see that the viability of the lambda is dependent on divisor’s lifetime. Also, writing out the name, “divisor,” reminds us to ensure that divisor lives at least as long as the lambda’s closures. That’s a more specific memory jog than the general “make sure nothing dangles” admonition that “[&]” conveys.*”*

*p. 236*

*bad C++ practice concerning lambdas: “*If you know that a closure will be used immediately (e.g., by being passed to an STL algorithm) and won’t be copied, there is no risk that references it holds will outlive the local variables and parameters in the environment where its lambda is created. In that case, you might argue, there’s no risk of dangling references, hence no reason to avoid a default by-reference capture mode. For example, our filtering lambda might be used only as an argument to C++11’s std::all\_of, which returns whether all elements in a range satisfy a condition:

template<typename C>

void workWithContainer(const C& container)

{

auto calc1 = computeSomeValue1(); // as above

auto calc2 = computeSomeValue2(); // as above

auto divisor = computeDivisor(calc1, calc2); // as above

using ContElemT = typename C::value\_type; // type of

// elements in

// container

using std::begin; // for

using std::end; // genericity;

// see Item 13

if (std::all\_of( // if all values

begin(container), end(container), // in container

[&](const ContElemT& value) // are multiples

{ return value % divisor == 0; }) // of divisor...

) {

… // they are...

} else {

… // at least one

} // isn't...

}

It’s true, this is safe, but its safety is somewhat precarious. If the lambda were found to be useful in other contexts (e.g., as a function to be added to the filters container) and was copy-and-pasted into a context where its closure could outlive divisor, you’d be back in dangle-city, and there’d be nothing in the capture clause to specifically remind you to perform lifetime analysis on divisor. *” example of not-portable code. We can’t use such function in other context.*

*p. 237*

*Using technique of lambdas and book recommendation: “*Long-term, it’s simply better software engineering to explicitly list the local variables and parameters that a lambda depends on.*”*

*Intbu type deducting in this example: “*By the way, the ability to use auto in C++14 lambda parameter specifications means that the code above can be simplified in C++14. The ContElemT typedef can be elimi- nated, and the if condition can be revised as follows:

if (std::all\_of(begin(container), end(container),

[&](const auto& value) // C++14

{ return value % divisor == 0; }))*”*

*example and using of default by-value capture in order to resist dangling: “*One way to solve our problem with divisor would be a default by-value capture mode. That is, we could add the lambda to filters as follows:

filters.emplace\_back( // now

[=](int value) { return value % divisor == 0; } // divisor

); // can't

// dangle

This suffices for this example, but, in general, default by-value capture isn’t the antidangling elixir you might imagine. The problem is that if you capture a pointer by value, you copy the pointer into the closures arising from the lambda, but you don’t prevent code outside the lambda from deleteing the pointer and causing your copies to dangle.*” But dangling is possible and in this case.*

*“*That could never happen!*”…*

*p. 237*

*12.01.2017*

*Recommendation of Mayers. It is necessary to be attentive: “*“That could never happen!” you protest. “Having read Chapter 4, I worship at the house of smart pointers. Only loser C++98 programmers use raw pointers and delete.” That may be true, but it’s irrelevant because you do, in fact, use raw point- ers, and they can, in fact, be deleted out from under you. It’s just that in your modern C++ programming style, there’s often little sign of it in the source code.*”*

*Example of this: “*Suppose one of the things Widgets can do is add entries to the container of filters:

class Widget {

public:

… // ctors, etc.

void addFilter() const; // add an entry to filters

private:

int divisor; // used in Widget's filter

};

Widget::addFilter could be defined like this: *”*

*p. 238*

*example continue: “*void Widget::addFilter() const

{

filters.emplace\_back(

[=](int value) { return value % divisor == 0; }

);

}

To the blissfully uninitiated, this looks like safe code. The lambda is dependent on divisor, but the default by-value capture mode ensures that divisor is copied into any closures arising from the lambda, right? Wrong. Completely wrong. Horribly wrong. Fatally wrong.*” Lambda expressions nuanes.*

*Lambdas structure: “*Captures apply only to non-static local variables (including parameters) visible in the scope where the lambda is created.*”*

*Example explanation: “*In the body of Widget::addFilter, divisor is not a local variable, it’s a data member of the Widget class. It can’t be captured. Yet if the default capture mode is eliminated, the code won’t compile:

void Widget::addFilter() const

{

filters.emplace\_back( // error!

[](int value) { return value % divisor == 0; } // divisor

); // not

} // available

Furthermore, if an attempt is made to explicitly capture divisor (either by value or by reference—it doesn’t matter), the capture won’t compile, because divisor isn’t a local variable or a parameter:

void Widget::addFilter() const

{

filters.emplace\_back(

[divisor](int value) // error! no local

{ return value % divisor == 0; } // divisor to capture

);

}

So if the default by-value capture clause isn’t capturing divisor, yet without the default by-value capture clause, the code won’t compile, what’s going on?*”*

*C++ structure, compiler work and example explanation: “*The explanation hinges on the implicit use of a raw pointer: this. Every non-static member function has a this pointer, and you use that pointer every time you mention a data member of the class. Inside any Widget member function, for example, compilers internally replace uses of divisor with this->divisor. In the version of Widget::addFilter with a default by-value capture,*”*

*p. 239*

*example continue: “*The explanation hinges on the implicit use of a raw pointer: this. Every non-static member function has a this pointer, and you use that pointer every time you mention a data member of the class. Inside any Widget member function, for example, compilers internally replace uses of divisor with this->divisor. In the version of Widget::addFilter with a default by-value capture, *” lambdas structure.*

*Lambdas structure: “*Understanding this is tantamount to understanding that the viability of the closures arising from this lambda is tied to the lifetime of the Widget whose this pointer they contain a copy of.*”*

*Another example: “*Understanding this is tantamount to understanding that the viability of the closures arising from this lambda is tied to the lifetime of the Widget whose this pointer they contain a copy of lifetime (see Item 18). From that point on, filters contains an entry with a dangling pointer. *” dangling pointer problem. Book structure (Item 18)*

*p. 240*

*solution of this problem: making a copy of local variable: “*This particular problem can be solved by making a local copy of the data member you want to capture and then capturing the copy:

void Widget::addFilter() const

{

auto divisorCopy = divisor; // copy data member

filters.emplace\_back(

[divisorCopy](int value) // capture the copy

{ return value % divisorCopy == 0; } // use the copy

);

}

To be honest, if you take this approach, default by-value capture will work, too,

void Widget::addFilter() const

{

auto divisorCopy = divisor; // copy data member

filters.emplace\_back(

[=](int value) // capture the copy

{ return value % divisorCopy == 0; } // use the copy

);

}

but why tempt fate? A default capture mode is what made it possible to accidentally capture this when you thought you were capturing divisor in the first place.*” Mayers doesn’t like default capture modes?*

*Generalized lambda expressions using context and advantage: “*In C++14, a better way to capture a data member is to use generalized lambda capture (see Item 32):

void Widget::addFilter() const

{

filters.emplace\_back( // C++14:

[divisor = divisor](int value) // copy divisor to closure

{ return value % divisor == 0; } // use the copy

);

}

There’s no such thing as a default capture mode for a generalized lambda capture, however, so even in C++14, the advice of this Item—to avoid default capture modes —stands. *” yes Mayers recommends to aboid default capture modes.*

*Another drawback of default by-value captures: “*An additional drawback to default by-value captures is that they can suggest that the corresponding closures are self-contained and insulated from changes to data outside the closures. In general, that’s not true, because lambdas may be dependent not just on local variables and parameters (which may be captured), but also on objects with static storage duration. Such objects are defined at global or namespace scope or are declared static inside classes, functions, or files. These objects can be used inside lambdas, but they can’t be captured.*” Lambdas structure. Kind of program objects – objects with static storage duration.*

*p. 141*

*example of this: “*Yet specification of a default by-value capture mode can lend the impression that they are. Consider this revised version of the add DivisorFilter function we saw earlier:

void addDivisorFilter()

{

static auto calc1 = computeSomeValue1(); // now static

static auto calc2 = computeSomeValue2(); // now static

static auto divisor = // now static

computeDivisor(calc1, calc2);

filters.emplace\_back(

[=](int value) // captures nothing!

{ return value % divisor == 0; } // refers to above static

);

++divisor; // modify divisor

}

A casual reader of this code could be forgiven for seeing “[=]” and thinking, “Okay, the lambda makes a copy of all the objects it uses and is therefore self-contained.” But it’s not self-contained. This lambda doesn’t use any non-static local variables, so nothing is captured. Rather, the code for the lambda refers to the static variable divisor. When, at the end of each invocation of addDivisorFilter, divisor is incremented, any lambdas that have been added to filters via this function will exhibit new behavior (corresponding to the new value of divisor). Practically speaking, this lambda captures divisor by reference, a direct contradiction to what the default by-value capture clause seems to imply. If you stay away from default byvalue capture clauses, you eliminate the risk of your code being misread in this way.*”*

*Book structure. Resume: “*Things to Remember

• Default by-reference capture can lead to dangling references.

• Default by-value capture is susceptible to dangling pointers (especially this), and it misleadingly suggests that lambdas are self-contained.*”*

*p. 242*

## Item 32: Use init capture to move objects into closures

*Task: how to move object to lambda expression: “*Sometimes neither by-value capture nor by-reference capture is what you want. If you have a move-only object (e.g., a std::unique\_ptr or a std::future) that you want to get into a closure, C++11 offers no way to do it. If you have an object that’s expensive to copy but cheap to move (e.g., most containers in the Standard Library), and you’d like to get that object into a closure, you’d much rather move it than copy it. Again, however, C++11 gives you no way to accomplish that.*” Movable objects: std::unique\_ptr and std::future. Characteristic of most STL containers.*

*C++ history: “*But that’s C++11. C++14 is a different story. It offers direct support for moving objects into closures. If your compilers are C++14-compliant, rejoice and read on. If you’re still working with C++11 compilers, you should rejoice and read on, too, because there are ways to approximate move capture in C++11.*”*

*C++ history and init capture concept: “*The absence of move capture was recognized as a shortcoming even as C++11 was adopted. The straightforward remedy would have been to add it in C++14, but the Standardization Committee chose a different path. They introduced a new capture mechanism that’s so flexible, capture-by-move is only one of the tricks it can perform. The new capability is called init capture. It can do virtually everything the C++11 capture forms can do, plus more. The one thing you can’t express with an init capture is a default capture mode, but Item 31 explains that you should stay away from those, anyway. (For situations covered by C++11 captures, init capture’s syntax is a bit wordier, so in cases where a C++11 capture gets the job done, it’s perfectly reasonable to use it.)*” book structure (Item 31). Using context of C++11 captures and init captures.*

*Purpose of init capture: “*Using an init capture makes it possible for you to specify 1. the name of a data member in the closure class generated from the lambda and 2. an expression initializing that data member.*”*

*Demonstrational example: “*Here’s how you can use init capture to move a std::unique\_ptr into a closure:

class Widget { // some useful type

public:

…

bool isValidated() const;

bool isProcessed() const;

bool isArchived() const;

private:

…

};

auto pw = std::make\_unique<Widget>(); // create Widget; see

// Item 21 for info on

// std::make\_unique

… // configure \*pw

auto func = [pw = std::move(pw)] // init data mbr

{ return pw->isValidated() // in closure w/

&& pw->isArchived(); }; // std::move(pw)*” example of capture by move.*

*p. 243*

*example description: “*The highlighted text comprises the init capture. To the left of the “=” is the name of the data member in the closure class you’re specifying, and to the right is the initializing expression.*”*

*Interesting init capture property: “*Interestingly, the scope on the left of the “=” is different from the scope on the right. The scope on the left is that of the closure class. The scope on the right is the same as where the lambda is being defined.*”*

*Example description: “*In the example above, the name pw on the left of the “=” refers to a data member in the closure class, while the name pw on the right refers to the object declared above the lambda, i.e., the variable initialized by the call to std::make\_unique. So “pw = std::move(pw)” means “create a data member pw in the closure, and initialize that data member with the result of applying std::move to the local variable pw.”*” Lambda structure: “*As usual, code in the body of the lambda is in the scope of the closure class, so uses of pw there refer to the closure class data member. The comment “configure \*pw” in this example indicates that after the Widget is created by std::make\_unique and before the std::unique\_ptr to that Widget is captured by the lambda, the Widget is modified in some way. If no such configuration is necessary, i.e., if the Widget created by std::make\_unique is in a state suitable to be captured by the lambda, the local variable pw is unnecessary, because the closure class’s data member can be directly initialized by std::make\_unique:

auto func = [pw = std::make\_unique<Widget>()] // init data mbr

{ return pw->isValidated() // in closure w/

&& pw->isArchived(); }; // result of call

// to make\_unique

*” example continue.*

*Another name of init capture. Its advantage. C++ history: “*This should make clear that the C++14 notion of “capture” is considerably generalized from C++11, because in C++11, it’s not possible to capture the result of an expression. As a result, another name for init capture is generalized lambda capture.*”*

*Book question: “*But what if one or more of the compilers you use lacks support for C++14’s init capture? How can you accomplish move capture in a language lacking support for move capture?*”*

*p. 244*

*lambda expression structure: “*Remember that a lambda expression is simply a way to cause a class to be generated and an object of that type to be created. There is nothing you can do with a lambda that you can’t do by hand.*”*

*Example of answer this question: “*The example C++14 code we just saw, for example, can be written in C++11 like this:

class IsValAndArch { // "is validated

public: // and archived"

using DataType = std::unique\_ptr<Widget>;

explicit IsValAndArch(DataType&& ptr) // Item 25 explains

: pw(std::move(ptr)) {} // use of std::move

bool operator()() const

{ return pw->isValidated() && pw->isArchived(); }

private:

DataType pw;

};

auto func = IsValAndArch(std::make\_unique<Widget>());

That’s more work than writing the lambda, but it doesn’t change the fact that if you want a class in C++11 that supports move-initialization of its data members, the only thing between you and your desire is a bit of time with your keyboard.*” Intbu this code and overloading of operator ().*

*Methods of move captures in C++11: “*If you want to stick with lambdas (and given their convenience, you probably do), move capture can be emulated in C++11 by 1. moving the object to be captured into a function object produced by std::bind and 2. giving the lambda a reference to the “captured” object.*”. intbu.*

*Std::bind using context. Intbu both: “*If you’re familiar with std::bind, the code is pretty straightforward. If you’re not familiar with std::bind, the code takes a little getting used to, but it’s worth the trouble.*”*

*Demonstrational example: “*Suppose you’d like to create a local std::vector, put an appropriate set of values into it, then move it into a closure. In C++14, this is easy:

std::vector<double> data; // object to be moved

// into closure

… // populate data

auto func = [data = std::move(data)] // C++14 init capture

{ /\* uses of data \*/ };

*”*

*P. 245*

*Example continue: “*I’ve highlighted key parts of this code: the type of object you want to move (std::vector<double>), the name of that object (data), and the initializing expression for the init capture (std::move(data)). The C++11 equivalent is as follows, where I’ve highlighted the same key things:

std::vector<double> data; // as above

… // as above

auto func =

std::bind( // C++11 emulation

[](const std::vector<double>& data) // of init capture

{ /\* uses of data \*/ },

std::move(data)

);

Like lambda expressions, std::bind produces function objects.*” Lambda expressions and std::bind common property. What is function object in C++?*

*Std::bind structure: “*I call function objects returned by std::bind bind objects. The first argument to std::bind is a callable object. Subsequent arguments represent values to be passed to that object.*”*

*Std::bind structure and description of the first solution of question: “*A bind object contains copies of all the arguments passed to std::bind. For each lvalue argument, the corresponding object in the bind object is copy constructed. For each rvalue, it’s move constructed. In this example, the second argument is an rvalue (the result of std::move—see Item 23), so data is move constructed into the bind object. This move construction is the crux of move capture emulation, because moving an rvalue into a bind object is how we work around the inability to move an rvalue into a C++11 closure.*”*

*Std::bind and function object structure: “*When a bind object is “called” (i.e., its function call operator is invoked) the arguments it stores are passed to the callable object originally passed to std::bind. In this example, that means that when func (the bind object) is called, the moveconstructed copy of data inside func is passed as an argument to the lambda that was passed to std::bind.*”*

*First solution description: “*This lambda is the same as the lambda we’d use in C++14, except a parameter, data, has been added to correspond to our pseudo-move-captured object. This parameter is an lvalue reference to the copy of data in the bind object. (It’s not an rvalue reference, because although the expression used to initialize the copy of data (“std::move(data)”) is an rvalue, the copy of data itself is an lvalue.) Uses of data inside the lambda will thus operate on the move-constructed copy of data inside the bind object.*”*

*Lambdas structure: “*By default, the operator() member function inside the closure class generated from a lambda is const. That has the effect of rendering all data members in the closure const within the body of the lambda.*” Intbu this.*

*p. 246*

*continue of first solution description and lambda property (mutable removes constness): “*The move-constructed copy of data inside the bind object is not const, however, so to prevent that copy of data from being modified inside the lambda, the lambda’s parameter is declared reference-to-const. If the lambda were declared mutable, operator() in its closure class would not be declared const, and it would be appropriate to omit const in the lambda’s parameter declaration:

auto func =

std::bind( // C++11 emulation

[](std::vector<double>& data) mutable // of init capture

{ /\* uses of data \*/ }, // for mutable lambda

std::move(data)

);*”*

*Description of first solution: “*Because a bind object stores copies of all the arguments passed to std::bind, the bind object in our example contains a copy of the closure produced by the lambda that is its first argument. The lifetime of the closure is therefore the same as the lifetime of the bind object. That’s important, because it means that as long as the closure exists, the bind object containing the pseudo-move-captured object exists, too.*”*

*Good example of reader orientation which should be placed before discussion better: “*If this is your first exposure to std::bind, you may need to consult your favorite C++11 reference before all the details of the foregoing discussion fall into place.*”*

*Resume of first solution: “*Even if that’s the case, these fundamental points should be clear:

• It’s not possible to move-construct an object into a C++11 closure, but it is possible to move-construct an object into a C++11 bind object.

• Emulating move-capture in C++11 consists of move-constructing an object into a bind object, then passing the move-constructed object to the lambda by reference.

• Because the lifetime of the bind object is the same as that of the closure, it’s possible to treat objects in the bind object as if they were in the closure.*”*

*Another example of first solution: “*As a second example of using std::bind to emulate move capture, here’s the C++14 code we saw earlier to create a std::unique\_ptr in a closure:

auto func = [pw = std::make\_unique<Widget>()] // as before,

{ return pw->isValidated() // create pw

&& pw->isArchived(); }; // in closure

And here’s the C++11 emulation:

auto func = std::bind(

[](const std::unique\_ptr<Widget>& pw)

{ return pw->isValidated()

&& pw->isArchived(); },

std::make\_unique<Widget>()

);*”*

*p. 247*

*std::bind using context in different C++ versions. Using context comparisons of std::bind and lambdas: “*It’s ironic that I’m showing how to use std::bind to work around limitations in C++11 lambdas, because in Item 34, I advocate the use of lambdas over std::bind. However, that Item explains that there are some cases in C++11 where std::bind can be useful, and this is one of them. (In C++14, features such as init capture and auto parameters eliminate those cases.)*”*

*book structure. Resume: “*Things to Remember • Use C++14’s init capture to move objects into closures. • In C++11, emulate init capture via hand-written classes or std::bind.*”*

*Was there description of the second solution?*

## Item 33: Use decltype on auto&& paramteres to std::forward them.

*Intbu the name of this Item.*

*C++ history and generic lambdas: “*One of the most exciting features of C++14 is generic lambdas—lambdas that use auto in their parameter specifications. The implementation of this feature is straightforward: operator()

in the lambda’s closure class is a template.*”*

*Example of it: “*Given this lambda, for example,

auto f = [](auto x){ return func(normalize(x)); };

the closure class’s function call operator looks like this:

class SomeCompilerGeneratedClassName {

public:

template<typename T> // see Item 3 for

auto operator()(T x) const // auto return type

{ return func(normalize(x)); }

… // other closure class

}; // functionality

In this example, the only thing the lambda does with its parameter x is forward it to normalize. If normalize treats lvalues differently from rvalues, this lambda isn’t written properly, because it always passes an lvalue (the parameter x) to normalize, even if the argument that was passed to the lambda was an rvalue.*” I feel forwarding here. Std::forward is necessary to distinguish rvalues from lvalues.*

*p. 248*

*Example continue: “*The correct way to write the lambda is to have it perfect-forward x to normalize. Doing that requires two changes to the code. First, x has to become a universal reference (see Item 24), and second, it has to be passed to normalize via std::forward (see Item 25). In concept, these are trivial modifications:

auto f = [](auto&& x)

{ return func(normalize(std::forward<???>(x))); };

Between concept and realization, however, is the question of what type to pass to std::forward, i.e., to determine what should go where I’ve written ??? above. *” example continue and book question.*

Normally, when you employ perfect…

*p. 248*

*13.01.2017*

*Perfect forwarding complexity in lambdas: “*Normally, when you employ perfect forwarding, you’re in a template function taking a type parameter T, so you just write std::forward<T>. In the generic lambda, though, there’s no type parameter T available to you. There is a T in the templatized operator() inside the closure class generated by the lambda, but it’s not possible to refer to it from the lambda, so it does you no good.*”*

*Decltype structure and using context. Perfect forwarding in lambdas: “*Item 28 explains that if an lvalue argument is passed to a universal reference parameter, the type of that parameter becomes an lvalue reference. If an rvalue is passed, the parameter becomes an rvalue reference. This means that in our lambda, we can determine whether the argument passed was an lvalue or an rvalue by inspecting the type of the parameter x. decltype gives us a way to do that (see Item 3). If an lvalue was passed in, decltype(x) will produce a type that’s an lvalue reference. If an rvalue was passed, decltype(x) will produce an rvalue reference type. Item 28 also explains that when calling std::forward, convention dictates that the type argument be an lvalue reference to indicate an lvalue and a non-reference to indicate an rvalue. In our lambda, if x is bound to an lvalue, decltype(x) will yield an lvalue reference. That conforms to convention. However, if x is bound to an rvalue, decltype(x) will yield an rvalue reference instead of the customary nonreference. *” another complexity of perfect forwarding in lambdas.*

*Std::forward structure: “*But look at the sample C++14 implementation for std::forward from Item 28:

template<typename T> // in namespace

T&& forward(remove\_reference\_t<T>& param) // std

{

return static\_cast<T&&>(param);

}

If client code wants to perfect-forward an rvalue of type Widget, it normally instantiates std::forward with the type Widget (i.e, a non-reference type), and the std::forward template yields this function:

Widget&& forward(Widget& param) // instantiation of

{ // std::forward when

return static\_cast<Widget&&>(param); // T is Widget

}*” intbu this template instantiation.*

*p. 249*

*std::forward structure: “*But consider what would happen if the client code wanted to perfect-forward the same rvalue of type Widget, but instead of following the convention of specifying T to be a non-reference type, it specified it to be an rvalue reference. That is, consider what would happen if T were specified to be Widget&&. After initial instantiation of std::forward and application of std::remove\_reference\_t, but before reference collapsing (once again, see Item 28), std::forward would look like this:

Widget&& && forward(Widget& param) // instantiation of

{ // std::forward when

return static\_cast<Widget&& &&>(param); // T is Widget&&

} // (before reference

// collapsing)

Applying the reference-collapsing rule that an rvalue reference to an rvalue reference becomes a single rvalue reference, this instantiation emerges:

Widget&& forward(Widget& param) // instantiation of

{ // std::forward when

return static\_cast<Widget&&>(param); // T is Widget&&

} // (after reference

// collapsing)

If you compare this instantiation with the one that results when std::forward is called with T set to Widget, you’ll see that they’re identical. That means that instantiating std::forward with an rvalue reference type yields the same result as instantiating it with a non-reference type. *” intbu this examples by remind of Item 28. Book structure.*

*Decltype structure: “*That’s wonderful news, because decltype(x) yields an rvalue reference type when an rvalue is passed as an argument to our lambda’s parameter x.*”*

*perfect forwarding lambda structure: “*We established above that when an lvalue is passed to our lambda, decltype(x) yields the customary type to pass to std::forward, and now we realize that for rvalues, decltype(x) yields a type to pass to std::forward that’s not conventional, but that nevertheless yields the same outcome as the conventional type. So for both lvalues and rvalues, passing decltype(x) to std::forward gives us the result we want. Our perfectforwarding lambda can therefore be written like this:

auto f =

[](auto&& param)

{

return

func(normalize(std::forward<decltype(param)>(param)));

};

From there, it’s just a hop, skip, and six dots to a perfect-forwarding lambda that accepts not just a single parameter, but any number of parameters, because C++14 lambdas can also be variadic:

auto f =

[](auto&&... params)

{

return

func(normalize(std::forward<decltype(params)>(params)...));

};*” C++14 lambdas characteristics.*

*p. 250*

*book structure. Resume: “*Things to Remember

• Use decltype on auto&& parameters to std::forward them.*”*

## Item 34: Prefer lambdas to std::bind

*C++ history and explanation of this book recommendation: “*std::bind is the C++11 successor to C++98’s std::bind1st and std::bind2nd, but, informally, it’s been part of the Standard Library since 2005. That’s when the Standardization Committee adopted a document known as TR1, which included bind’s specification. (In TR1, bind was in a different namespace, so it was std::tr1::bind, not std::bind, and a few interface details were different.) This history means that some programmers have a decade or more of experience using std::bind. If you’re one of them, you may be reluctant to abandon a tool that’s served you well. That’s understandable, but in this case, change is good, because in C++11, lambdas are almost always a better choice than std::bind. As of C++14, the case for lambdas isn’t just stronger, it’s downright ironclad.*”*

*Book structure. Example of necessary knowledge basis: “*This Item assumes that you’re familiar with std::bind. If you’re not, you’ll want to acquire a basic understanding before continuing. Such an understanding is worth- while in any case, because you never know when you might encounter uses of std::bind in a code base you have to read or maintain.*”*

*Book terminology: “*As in Item 32, I refer to the function objects returned from std::bind as bind objects.*”*

*Advantage of lambdas over std::bind: “*The most important reason to prefer lambdas over std::bind is that lambdas are more readable.”

*Example of it:* “Suppose, for example, we have a function to set up an audible alarm:

// typedef for a point in time (see Item 9 for syntax)

using Time = std::chrono::steady\_clock::time\_point;

// see Item 10 for "enum class"

enum class Sound { Beep, Siren, Whistle };

// typedef for a length of time

using Duration = std::chrono::steady\_clock::duration;

// at time t, make sound s for duration d

void setAlarm(Time t, Sound s, Duration d);*”*

*p. 251*

*example continue: “*Further suppose that at some point in the program, we’ve determined we’ll want an alarm that will go off an hour after it’s set and that will stay on for 30 seconds. The alarm sound, however, remains undecided. We can write a lambda that revises setAlarm’s interface so that only a sound needs to be specified:

// setSoundL ("L" for "lambda") is a function object allowing a

// sound to be specified for a 30-sec alarm to go off an hour

// after it's set

auto setSoundL =

[](Sound s)

{

// make std::chrono components available w/o qualification

using namespace std::chrono;

setAlarm(steady\_clock::now() + hours(1), // alarm to go off

s, // in an hour for

seconds(30)); // 30 seconds

};

I’ve highlighted the call to setAlarm inside the lambda. This is a normal-looking function call, and even a reader with little lambda experience can see that the parameter s passed to the lambda is passed as an argument to setAlarm. We can streamline this code in C++14 by availing ourselves of the standard suffixes for seconds (s), milliseconds (ms), hours (h), etc., that build on C++11’s support for user-defined literals. These suffixes are implemented in the std::literals namespace, so the above code can be rewritten as follows:

auto setSoundL =

[](Sound s)

{

using namespace std::chrono;

using namespace std::literals; // for C++14 suffixes

setAlarm(steady\_clock::now() + 1h, // C++14, but

s, // same meaning

30s); // as above

};

Item 34 | 233

www.it-ebooks.info*” std::literals using context.*

*p. 252*

*example continue: “*Our first attempt to write the corresponding std::bind call is below. It has an error that we’ll fix in a moment, but the correct code is more complicated, and even this simplified version brings out some important issues:

using namespace std::chrono; // as above

using namespace std::literals;

using namespace std::placeholders; // needed for use of "\_1"

auto setSoundB = // "B" for "bind"

std::bind(setAlarm,

steady\_clock::now() + 1h, // incorrect! see below

\_1,

30s);*” what are std::placeholders??? It is if we always want to pass concrete value to function? Why is it incorrect?*

*Explain description: “*I’d like to highlight the call to setAlarm here as I did in the lambda, but there’s no call to highlight. Readers of this code simply have to know that calling setSoundB invokes setAlarm with the time and duration specified in the call to std::bind. To the uninitiated, the placeholder “\_1” is essentially magic, but even readers in the know have to mentally map from the number in that placeholder to its position in the std::bind parameter list in order to understand that the first argument in a call to setSoundB is passed as the second argument to setAlarm. The type of this argument is not identified in the call to std::bind, so readers have to consult the setAlarm declaration to determine what kind of argument to pass to setSoundB. *” intbu what is std::placeholder? Firs parameter of setBoundB will be sound and will be passed as second parameter for setAlarm(). Maybe I understand.*

*Example explanation: “*But, as I said, the code isn’t quite right. In the lambda, it’s clear that the expression “steady\_clock::now() + 1h” is an argument to setAlarm. It will be evaluated when setAlarm is called. That makes sense: we want the alarm to go off an hour after invoking setAlarm. In the std::bind call, however, “steady\_clock::now() + 1h” is passed as an argument to std::bind, not to setAlarm. That means that the expression will be evaluated when std::bind is called, and the time resulting from that expression will be stored inside the resulting bind object. As a consequence, the alarm will be set to go off an hour after the call to std::bind, not an hour after the call to setAlarm!*”*

*example continue: “*Fixing the problem requires telling std::bind to defer evaluation of the expression until setAlarm is called, and the way to do that is to nest a second call to std::bind inside the first one:

auto setSoundB =

std::bind(setAlarm,

std::bind(std::plus<>(), steady\_clock::now(), 1h),

\_1,

30s);*” yes… this code is not readable enough.*

*p. 253*

*C++ history and standard operators templates: “*If you’re familiar with the std::plus template from C++98, you may be surprised to see that in this code, no type is specified between the angle brackets, i.e., the code contains “std::plus<>”, not “std::plus<type>”. In C++14, the template type argument for the standard operator templates can generally be omitted, so there’s no need to provide it here. C++11 offers no such feature, so the C++11 std::bind equivalent to the lambda is*”*

*example continue: “*using namespace std::chrono; // as above

using namespace std::placeholders;

auto setSoundB =

std::bind(setAlarm,

std::bind(std::plus<steady\_clock::time\_point>(),

steady\_clock::now(),

hours(1)),

\_1,

seconds(30));

If, at this point, the lambda’s not looking a lot more attractive, you should probably have your eyesight checked.*”*

*The next situation. If function is overloaded: “*When setAlarm is overloaded, a new issue arises. Suppose there’s an overload taking a fourth parameter specifying the alarm volume:

enum class Volume { Normal, Loud, LoudPlusPlus };

void setAlarm(Time t, Sound s, Duration d, Volume v);

The lambda continues to work as before, because overload resolution chooses the three-argument version of setAlarm:

auto setSoundL = // same as before

[](Sound s)

{

using namespace std::chrono;

setAlarm(steady\_clock::now() + 1h, // fine, calls

s, // 3-arg version

30s); // of setAlarm

};

The std::bind call, on the other hand, now fails to compile:

auto setSoundB = // error! which

std::bind(setAlarm, // setAlarm?

std::bind(std::plus<>(),

steady\_clock::now(),

1h),

\_1,

30s);

The problem is that compilers have no way to determine which of the two setAlarm functions they should pass to std::bind. All they have is a function name, and the name alone is ambiguous. *” disadvantage of std::bind comparing with lambdas. Its less scalable.*

*p. 254*

*example continue: “*To get the std::bind call to compile, setAlarm must be cast to the proper function pointer type:

using SetAlarm3ParamType = void(\*)(Time t, Sound s, Duration d);

auto setSoundB = // now

std::bind(static\_cast<SetAlarm3ParamType>(setAlarm), // okay

std::bind(std::plus<>(),

steady\_clock::now(),

1h),

\_1,

30s);

But this brings up another difference between lambdas and std::bind. Inside the function call operator for setSoundL (i.e., the function call operator of the lambda’s closure class), the call to setAlarm is a normal function invocation that can be inlined by compilers in the usual fashion:

setSoundL(Sound::Siren); // body of setAlarm may

// well be inlined here

The call to std::bind, however, passes a function pointer to setAlarm, and that means that inside the function call operator for setSoundB (i.e., the function call operator for the bind object), the call to setAlarm takes place through a function pointer. Compilers are less likely to inline function calls through function pointers, and that means that calls to setAlarm through setSoundB are less likely to be fully inlined than those through setSoundL:

setSoundB(Sound::Siren); // body of setAlarm is less

// likely to be inlined here

It’s thus possible that using lambdas generates faster code than using std::bind. *” another advantage of lambdas over std::bind. Due to inline it can be faster.*

*Another example: “*The setAlarm example involves only a simple function call. If you want to do anything more complicated, the scales tip even further in favor of lambdas. For example, consider this C++14 lambda, which returns whether its argument is between a minimum value (lowVal) and a maximum value (highVal), where lowVal and highVal are local variables:*”*

*p. 255*

*example continue: “*auto betweenL =

[lowVal, highVal]

(const auto& val) // C++14

{ return lowVal <= val && val <= highVal; };

*” this is good example of auto. There is evident what type is.*

*Example continue: “*std::bind can express the same thing, but the construct is an example of job security through code obscurity:

using namespace std::placeholders; // as above

auto betweenB =

std::bind(std::logical\_and<>(), // C++14

std::bind(std::less\_equal<>(), lowVal, \_1),

std::bind(std::less\_equal<>(), \_1, highVal));

In C++11, we’d have to specify the types we wanted to compare, and the std::bind call would then look like this:

auto betweenB = // C++11 version

std::bind(std::logical\_and<bool>(),

std::bind(std::less\_equal<int>(), lowVal, \_1),

std::bind(std::less\_equal<int>(), \_1, highVal));

Of course, in C++11, the lambda couldn’t take an auto parameter, so it’d have to commit to a type, too:

auto betweenL = // C++11 version

[lowVal, highVal]

(int val)

{ return lowVal <= val && val <= highVal; };

Either way, I hope we can agree that the lambda version is not just shorter, but also more comprehensible and maintainable.

*” three advantages of lambda expressions.*

*Another example. Placeholders demonstration: “*Earlier, I remarked that for those with little std::bind experience, its placeholders (e.g., \_1, \_2, etc.) are essentially magic. But it’s not just the behavior of the placeholders that’s opaque. Suppose we have a function to create compressed copies of Widgets,

enum class CompLevel { Low, Normal, High }; // compression

// level

Widget compress(const Widget& w, // make compressed

CompLevel lev); // copy of w

and we want to create a function object that allows us to specify how much a particular Widget w should be compressed. This use of std::bind will create such an object:

Widget w;

using namespace std::placeholders;

auto compressRateB = std::bind(compress, w, \_1);*”*

*p. 256*

*example description and std::bind structure: “*Now, when we pass w to std::bind, it has to be stored for the later call to compress. It’s stored inside the object compressRateB, but how is it stored—by value or by reference? It makes a difference, because if w is modified between the call to std::bind and a call to compressRateB, storing w by reference will reflect the changes, while storing it by value won’t. The answer is that it’s stored by value,1 but the only way to know that is to memorize how std::bind works; there’s no sign of it in the call to std::bind.*”*

*Example continue and better readability of lambdas: “*Contrast that with a lambda approach, where whether w is captured by value or by reference is explicit:

auto compressRateL = // w is captured by

[w](CompLevel lev) // value; lev is

{ return compress(w, lev); }; // passed by value

Equally explicit is how parameters are passed to the lambda. Here, it’s clear that the parameter lev is passed by value. Hence:

compressRateL(CompLevel::High); // arg is passed

// by value

But in the call to the object resulting from std::bind, how is the argument passed?

compressRateB(CompLevel::High); // how is arg

// passed?

Again, the only way to know is to memorize how std::bind works. (The answer is that all arguments passed to bind objects are passed by reference, because the function call operator for such objects uses perfect forwarding.)*” intbu this structural aspect of std::bind.*

*Using context of std::bind and C++ history: “*Compared to lambdas, then, code using std::bind is less readable, less expressive, and possibly less efficient. In C++14, there are no reasonable use cases for std::bind. In C++11, however, std::bind can be justified in two constrained situations:*”*

*std::ref using context: “*std::bind always copies its arguments, but callers can achieve the effect of having an argument stored by reference by applying std::ref to it. The result of

auto compressRateB = std::bind(compress, std::ref(w), \_1);

is that compressRateB acts as if it holds a reference to w, rather than a copy. *” intbu if arguments are passed by reference or by value?*

*p. 257*

*std::bind using context in C++11: “*Move capture. C++11 lambdas don’t offer move capture, but it can be emulated through a combination of a lambda and std::bind. For details, consult Item 32, which also explains that in C++14, lambdas’ support for init capture eliminates the need for the emulation. • Polymorphic function objects. Because the function call operator on a bind object uses perfect forwarding, it can accept arguments of any type (modulo the restrictions on perfect forwarding described in Item 30). This can be useful when you want to bind an object with a templatized function call operator.*” Remind perfect forwarding restrictions.*

*Example of the last case: “*For example, given this class,

class PolyWidget {

public:

template<typename T>

void operator()(const T& param);

…

};

std::bind can bind a PolyWidget as follows:

PolyWidget pw;

auto boundPW = std::bind(pw, \_1);

boundPW can then be called with different types of arguments:

boundPW(1930); // pass int to

// PolyWidget::operator()

boundPW(nullptr); // pass nullptr to

// PolyWidget::operator()

boundPW("Rosebud"); // pass string literal to

// PolyWidget::operator()*” what is using context of templatized constructor?*

*C++ history and lambdas using context. What is conceptual using context of lambdas? Intbu: “*There is no way to do this with a C++11 lambda. In C++14, however, it’s easily achieved via a lambda with an auto parameter:

auto boundPW = [pw](const auto& param) // C++14

{ pw(param); };*”*

*Kind of lifecycle of some technology. C++ concept: “*These are edge cases, of course, and they’re transient edge cases at that, because compilers supporting C++14 lambdas are increasingly common. When bind was unofficially added to C++ in 2005, it was a big improvement over its 1998 predecessors. The addition of lambda support to C++11 rendered std::bind all but obsolete, however, and as of C++14, there are just no good use cases for it.*”*

*p. 258*

*book structure. Resume: “*Things to Remember

• Lambdas are more readable, more expressive, and may be more efficient than using std::bind.

• In C++11 only, std::bind may be useful for implementing move capture or for binding objects with templatized function call operators.*”*

*p. 259*

# CHAPTER 7 The Concurrency API

*C++ history and new feature – concurrency: “*One of C++11’s great triumphs is the incorporation of concurrency into the language and library. Programmers familiar with other threading APIs (e.g., pthreads or Windows threads) are sometimes surprised at the comparatively Spartan feature set that C++ offers, but that’s because a great deal of C++’s support for concurrency is in the form of constraints on compiler-writers. The resulting language assurances mean that for the first time in C++’s history, programmers can write multithreaded programs with standard behavior across all platforms. This establishes a solid foundation on which expressive libraries can be built, and the concurrency elements of the Standard Library (tasks, futures, threads, mutexes, condition variables, atomic objects, and more) are merely the beginning of what is sure to become an increasingly rich set of tools for the development of concurrent C++ software.*” C++ concurrency structure.*

*Future templates of Standard Library. Book structure: “*In the Items that follow, bear in mind that the Standard Library has two templates for futures: std::future and std::shared\_future. In many cases, the distinction is not important, so I often simply talk about futures, by which I mean both kinds.*”*

## Item 35: Prefer task-based programming to thread-based

*Two kinds of programming. Intbu their structure.*

*p. 259*

*16.01.2017*

*Example of thread-based approach of realization of asynchronous functions: “*If you want to run a function doAsyncWork asynchronously, you have two basic choices. You can create a std::thread and run doAsyncWork on it, thus employing a thread-based approach:

int doAsyncWork();

std::thread t(doAsyncWork);*” std::thread example.*

*Example of task-based strategy of asynchronous programming: “*Or you can pass doAsyncWork to std::async, a strategy known as task-based: auto fut = std::async(doAsyncWork); // "fut" for "future"

In such calls, the function object passed to std::async (e.g., doAsyncWork) is considered a task.*” Std::async example.*

*p. 260*

*advantage of task-based programming over thread-based: “*The task-based approach is typically superior to its thread-based counterpart, and the tiny amount of code we’ve seen already demonstrates some reasons why. Here, doAsyncWork produces a return value, which we can reasonably assume the code invoking doAsyncWork is interested in. With the thread-based invocation, there’s no straightforward way to get access to it. With the task-based approach, it’s easy, because the future returned from std::async offers the get function. The get function is even more important if doAsyncWork emits an exception, because get

provides access to that, too. With the thread-based approach, if doAsyncWork throws, the program dies (via a call to std::terminate).*” Future structure.*

*Another advantage. Higher level of abstraction that task-based embodies: “*A more fundamental difference between thread-based and task-based programming is the higher level of abstraction that task-based embodies. It frees you from the details of thread management, an observation that reminds me that I need to summarize the three meanings of “thread” in concurrent C++ software:*” for explanation of concept “level of astraction”. The more abstract the level the less details is to take into account.*

*Three meanings of thread in concurrent C++ software: “*Hardware threads are the threads that actually perform computation. Contemporary machine architectures offer one or more hardware threads per CPU core.” *Hardware threads definition.*

“Software threads (also known as OS threads or system threads) are the threads that the operating system1 manages across all processes and schedules for execution on hardware threads. It’s typically possible to create more software threads than hardware threads, because when a software thread is blocked (e.g., on I/O or waiting for a mutex or condition variable), throughput can be improved by executing other, unblocked, threads.*” Software threads definition. OS structure.*

*Std::thread definition: “*std::threads are objects in a C++ process that act as handles to underlying software threads. Some std::thread objects represent “null” handles, i.e., correspond to no software thread, because they’re in a default-constructed state (hence have no function to execute), have been moved from (the moved-to std::thread then acts as the handle to the underlying software thread), have been joined (the function they were to run has finished), or have been detached (the connection between them and their underlying software thread has been severed).*” Events which can occur with std::thread.*

*OS limitation and reason of std::system\_error exception: “*Software threads are a limited resource. If you try to create more than the system can provide, a std::system\_error exception is thrown. This is true even if the function you want to run can’t throw.*”*

*p. 243*

*there is example of this: “*For example, even if doAsyncWork

is noexcept, int doAsyncWork() noexcept; // see Item 14 for noexcept

this statement could result in an exception:

std::thread t(doAsyncWork); // throws if no more

// threads are available*”*

*Book question: ‘’* Well-written software must somehow deal with this possibility, but how?*”*

*Using context of multithreading: “*One approach is to run doAsyncWork on the current thread, but that could lead to unbalanced loads and, if the current thread is a GUI thread, responsiveness issues.*”*

*Problem of thread blocking: “*Another option is to wait for some existing software threads to complete and then try to create a new std::thread again, but it’s possible that the existing threads are waiting for an action that doAsyncWork is supposed to perform (e.g., produce a result or notify a condition variable).*”*

*Phenomenon of oversubscription and OS structure: thread scheduler: “*Even if you don’t run out of threads, you can have trouble with oversubscription. That’s when there are more ready-to-run (i.e., unblocked) software threads than hardware threads. When that happens, the thread scheduler (typically part of the OS) time-slices the software threads on the hardware. When one thread’s time-slice is finished and another’s begins, a context switch is performed. Such context switches increase the overall thread management overhead of the system, and they can be particularly costly when the hardware thread on which a software thread is scheduled is on a different core than was the case for the software thread during its last time-slice.*” This phenomenon can be costly in myltithread systems.*

*Oversubscription description. Intbu these problems with CPU caches: “*In that case, (1) the CPU caches are typically cold for that software thread (i.e., they contain little data and few instructions useful to it) and (2) the running of the “new” software thread on that core “pollutes” the CPU caches for “old” threads that had been running on that core and are likely to be scheduled to run there again.*”*

*Oversubscription is difficult to avoid: “*Avoiding oversubscription is difficult, because the optimal ratio of software to hardware threads depends on how often the software threads are runnable, and that can change dynamically, e.g., when a program goes from an I/O-heavy region to a computation-heavy region.*” Intbu concepts of program performing concepts: I/O-heave region and computation0heavy region.*

*Oversubscription structure and example of program behavior strongly dependent on machine architecture: “*The best ratio of software to hardware threads is also dependent on the cost of context switches and how effectively the software threads use the CPU caches. Furthermore, the number of hardware threads and the details of the CPU caches (e.g., how large they are and their relative speeds) depend on the machine architecture, so even if you tune your application to avoid oversubscription (while still keeping the hardware busy) on one platform, there’s no guarantee that your solution will work well on other kinds of machines.*”*

*Important parameter of multithreading: relation between of system thread number and hardware thread number.*

*Advantages of std::async: “*Your life will be easier if you dump these problems on somebody else, and using std::async does exactly that:

auto fut = std::async(doAsyncWork); // onus of thread mgmt is

// on implementer of

// the Standard Library*”*

*p. 262*

*std::async realization: “*Your life will be easier if you dump these problems on somebody else, and using std::async does exactly that:

auto fut = std::async(doAsyncWork); // onus of thread mgmt is

// on implementer of

// the Standard Library*” intbu text about scheduler.*

*Intbu translation: “*If you pulled this “run it on the thread needing the result” trick yourself, I remarked that it could lead to load-balancing issues, and those issues don’t go away simply because it’s std::async and the runtime scheduler that confront them instead of you.*”*

*advantages of std::async which treat as far as I understood OS thread scheduler: “*When it comes to load balancing, however, the runtime scheduler is likely to have a more comprehensive picture of what’s happening on the machine than you do, because it manages the threads from all processes, not just the one your code is running in.*”*

*intbu what is std::launch::async and std::async realization: “*With std::async, responsiveness on a GUI thread can still be problematic, because the scheduler has no way of knowing which of your threads has tight responsiveness requirements. In that case, you’ll want to pass the std::launch::async launch policy to std::async. That will ensure that the function you want to run really executes on a different thread (see Item 36).*” Function can not be implemented in a different thread? Intbu.*

*Advantage of std::async: “*State-of-the-art thread schedulers employ system-wide thread pools to avoid oversubscription, and they improve load balancing across hardware cores through workstealing algorithms.*”*

*Intbu this two multithreading techniques: “*The C++ Standard does not require the use of thread pools or work-stealing, and, to be honest, there are some technical aspects of the C++11 concurrency specification that make it more difficult to employ them than we’d like.*” C++11 disadvantage or nuance? Intbu!*

*STD implementations difference. Intbu: “*Nevertheless, some vendors take advantage of this technology in their Standard Library implementations, and it’s reasonable to expect that progress will continue in this area.*”*

*Advantages of using C++11 threading API. Three problems of multithreading programming. What is thread exhaustion?: “*If you take a task-based approach to your concurrent programming, you automatically reap the benefits of such technology as it becomes more widespread. If, on the other hand, you program directly with std::threads, you assume the burden of dealing with thread exhaustion, oversubscription, and load balancing yourself, not to mention how your solutions to these problems mesh with the solutions implemented in programs running in other processes on the same machine.*”*

*Two advantages of task-based approach over thread-based approach: “*Compared to thread-based programming, a task-based design spares you the travails of manual thread management, and it provides a natural way to examine the results of asynchronously executed functions (i.e., return values or exceptions).*”*

*p. 263*

*using context of thread-based strategy: “*Nevertheless, there are some situations where using threads directly may be appropriate. They include:

You need access to the API of the underlying threading implementation. The C++ concurrency API is typically implemented using a lower-level platformspecific API, usually pthreads or Windows’ Threads. Those APIs are currently richer than what C++ offers. (For example, C++ has no notion of thread priorities or affinities.) To provide access to the API of the underlying threading implementation, std::thread objects typically offer the native\_handle member function. There is no counterpart to this functionality for std::futures (i.e., for what std::async returns). *” realization of C++ threading API. Disadvantage of std::future.*

*Using context of thread-based strategy: “*You need to and are able to optimize thread usage for your application. This could be the case, for example, if you’re developing server software with a known execution profile that will be deployed as the only significant process on a machine with fixed hardware characteristics.*” What is execution profile (characteristic of program performing)*

*using context of thread-based strategy: “*You need to implement threading technology beyond the C++ concurrency API, e.g., thread pools on platforms where your C++ implementations don’t offer them.*” How is it possible to realize thread pool in C++?*

*Frequency of using thread-based and task-based API: “*These are uncommon cases, however. Most of the time, you should choose taskbased designs instead of programming with threads.*”*

*Book structure. Resume: “*Things to Remember

• The std::thread API offers no direct way to get return values from asynchronously run functions, and if those functions throw, the program is terminated.

• Thread-based programming calls for manual management of thread exhaustion, oversubscription, load balancing, and adaptation to new platforms. Task-based programming via std::async with the default launch policy handles most of these issues for you.*” task of myltithread programming.*

## Item 36: Specigy std::launch::async if asynchronicity is essential

*What is asynchronicity? Intbu. Book recommendation.*

*Function is kind of callable* *object. What is another? Lambda? And another? Pointer to function? Intbu: “*When you call std::async to execute a function (or other callable object), you’re generally intending to run the function asynchronously. But that’s not necessarily what you’re asking std::async to do. You’re really requesting that the function be run in accord with a std::async launch policy. There are two standard policies, each represented by an enumerator in the std::launch scoped enum. See Item 10 for information on scoped enums.)*” std::async structure. Structure and purpose of std::launch::async. Structure of concept Scoped enum.*

*p. 264*

*Two policies of std::async: “*Assuming a function f is passed to std::async for execution,

• The std::launch::async launch policy means that f must be run asynchro- nously, i.e., on a different thread.

• The std::launch::deferred launch policy means that f may run only when get or wait is called on the future returned by std::async.2 That is, f’s execution is deferred until such a call is made. When get or wait is invoked, f will execute synchronously, i.e., the caller will block until f finishes running. If nei- ther get nor wait is called, f will never run. *” definition of asynchronicity and synchronicity.*

*Default std::async policy: “*Perhaps surprisingly, std::async’s default launch policy—the one it uses if you don’t expressly specify one—is neither of these. Rather, it’s these or-ed together. The following two calls have exactly the same meaning:

auto fut1 = std::async(f); // run f using

// default launch

// policy

auto fut2 = std::async(std::launch::async | // run f either

std::launch::deferred, // async or

f); // deferred

The default policy thus permits f to be run either asynchronously or synchronously.*”*

Foundation of this default policy: “As Item 35 points out, this flexibility permits std::async and the threadmanagement components of the Standard Library to assume responsibility for thread creation and destruction, avoidance of oversubscription, and load balancing. That’s among the things that make concurrent programming with std::async so convenient.”

*intbu this nuance: “*This is a simplification. What matters isn’t the future on which get or wait is invoked, it’s the shared state to which the future refers. (Item 38 discusses the relationship between futures and shared states.) Because std::futures support moving and can also be used to construct std::shared\_futures, and because std::shared\_futures can be copied, the future object referring to the shared state arising from the call to std::async to which f was passed is likely to be different from the one returned by std::async. That’s a mouthful, however, so it’s common to fudge the truth and simply talk about invoking get or wait on the future returned from std::async.*”*

*Std::async and default policy structure: “*But using std::async with the default launch policy has some interesting implications. Given a thread t executing this statement,

auto fut = std::async(f); // run f using default launch policy *”*

*p. 265*

*implication of default policy of std::thread: “*It’s not possible to predict whether f will run concurrently with t, because f might be scheduled to run deferred.

• It’s not possible to predict whether f runs on a thread different from the thread invoking get or wait on fut. If that thread is t, the implication is that it’s not possible to predict whether f runs on a thread different from t.

• It may not be possible to predict whether f runs at all, because it may not be possible to guarantee that get or wait will be called on fut along every path through the program. *” intbu the last implication.*

*What is thread-local storage (TLS): “*The default launch policy’s scheduling flexibility often mixes poorly with the use of thread\_local variables, because it means that if f reads or writes such thread-local storage (TLS), it’s not possible to predict which thread’s variables will be accessed:

auto fut = std::async(f); // TLS for f possibly for

// independent thread, but

// possibly for thread

// invoking get or wait on fut*” limitations of std::async. Intbu.*

*Wait\_for and until\_for nuances and std::async limitations. Intbu this example: “*It also affects wait-based loops using timeouts, because calling wait\_for or wait\_until on a task (see Item 35) that’s deferred yields the value std::launch::deferred. This means that the following loop, which looks like it should eventually terminate, may, in reality, run forever:

using namespace std::literals; // for C++14 duration

// suffixes; see Item 34

void f() // f sleeps for 1 second,

{ // then returns

std::this\_thread::sleep\_for(1s);

}

auto fut = std::async(f); // run f asynchronously

// (conceptually)

while (fut.wait\_for(100ms) != // loop until f has

std::future\_status::ready) // finished running...

{ // which may never happen!

…

}*”*

*p. 266*

*example explanation: “*If f runs concurrently with the thread calling std::async (i.e., if the launch policy chosen for f is std::launch::async), there’s no problem here (assuming f eventually finishes), but if f is deferred, fut.wait\_for will always return std::future\_status::deferred. That will never be equal to std::future\_status:: ready, so the loop will never terminate.*”*

*Complexity of this bug. Intbu the last sentence: “*This kind of bug is easy to overlook during development and unit testing, because it may manifest itself only under heavy loads. Those are the conditions that push the machine towards oversubscription or thread exhaustion, and that’s when a task may be most likely to be deferred. After all, if the hardware isn’t threatened by oversubscription or thread exhaustion, there’s no reason for the runtime system not to schedule the task for concurrent execution.*”*

*Possible solution of this bug and std::future disadvantage: “*The fix is simple: just check the future corresponding to the std::async call to see whether the task is deferred, and, if so, avoid entering the timeout-based loop. Unfortunately, there’s no direct way to ask a future whether its task is deferred. Instead, you have to call a timeout-based function—a function such as wait\_for. In this case, you don’t really want to wait for anything, you just want to see if the return value is std::future\_status::deferred, so stifle your mild disbelief at the necessary circumlocution and call wait\_for with a zero timeout:

auto fut = std::async(f); // as above

if (fut.wait\_for(0s) == // if task is

std::future\_status::deferred) // deferred...

{

// ...use wait or get on fut

… // to call f synchronously

} else { // task isn't deferred

while (fut.wait\_for(100ms) != // infinite loop not

std::future\_status::ready) { // possible (assuming

// f finishes)

… // task is neither deferred nor ready,

// so do concurrent work until it's ready

}

… // fut is ready

}*”*

*p. 267*

*using context of std::async default policy: “*The upshot of these various considerations is that using std::async with the default launch policy for a task is fine as long as the following conditions are fulfilled:

• The task need not run concurrently with the thread calling get or wait.

• It doesn’t matter which thread’s thread\_local variables are read or written.

• Either there’s a guarantee that get or wait will be called on the future returned by std::async or it’s acceptable that the task may never execute.

• Code using wait\_for or wait\_until takes the possibility of deferred status into account.*”*

*Using context of std::launch::async policy: “*The upshot of these various considerations is that using std::async with the default launch policy for a task is fine as long as the following conditions are fulfilled:

• The task need not run concurrently with the thread calling get or wait.

• It doesn’t matter which thread’s thread\_local variables are read or written.

• Either there’s a guarantee that get or wait will be called on the future returned by std::async or it’s acceptable that the task may never execute.

• Code using wait\_for or wait\_until takes the possibility of deferred status into account. This function receives a callable object f and zero or more parameters params and perfect-forwards them (see Item 25) to std::async, passing std::launch::async as the launch policy. Like std::async, it returns a std::future for the result of invoking f on params. Determining the type of that result is easy, because the type trait std::result\_of gives it to you. (See Item 9 for general information on type traits.) reallyAsync is used just like std::async:

auto fut = reallyAsync(f); // run f asynchronously;

// throw if std::async

// would throw*” Example of really asynchronous function solution. Example of perfect-forwarding. Intbu type trait std::result\_of.*

*p. 268*

*example continue: “*In C++14, the ability to deduce reallyAsync’s return type streamlines the function declaration:

template<typename F, typename... Ts>

inline

auto // C++14

reallyAsync(F&& f, Ts&&... params)

{

return std::async(std::launch::async,

std::forward<F>(f),

std::forward<Ts>(params)...);

}

This version makes it crystal clear that reallyAsync does nothing but invoke std::async with the std::launch::async launch policy.*” Remind why is it not possible in C++14. C++ history.*

*Book structure. Resume:”* Things to Remember • The default launch policy for std::async permits both asynchronous and synchronous task execution.

• This flexibility leads to uncertainty when accessing thread\_locals, implies that the task may never execute, and affects program logic for timeout-based wait calls.

• Specify std::launch::async if asynchronous task execution is essential.*”*

## Item 37: Make std::threads unjoinable on all pahts.

*Intbu this recommendation.*

*States of std::thread object: “*Every std::thread object is in one of two states: joinable or unjoinable. A joinable std::thread corresponds to an underlying asynchronous thread of execution that is or could be running. A std::thread corresponding to an underlying thread that’s blocked or waiting to be scheduled is joinable, for example. std::thread objects corresponding to underlying threads that have run to completion are also considered joinable.*”*

*Unjoinable std::thread: “*An unjoinable std::thread is what you’d expect: a std::thread that’s not joinable. Unjoinable std::thread objects include:

• Default-constructed std::threads. Such std::threads have no function to execute, hence don’t correspond to an underlying thread of execution. *”*

*p. 269*

*other kinds of unjoinable threads: “*std::thread objects that have been moved from. The result of a move is that the underlying thread of execution a std::thread used to correspond to (if any) now corresponds to a different std::thread.

• std::threads that have been joined. After a join, the std::thread object no longer corresponds to the underlying thread of execution that has finished running.

• std::threads that have been detached. A detach severs the connection between a std::thread object and the underlying thread of execution it corresponds to. *” operations with std::thread.*

One reason a std::thread’s…

*p. 269*

*17.01.2017*

*Std::thread joinability structure: “*One reason a std::thread’s joinability is important is that if the destructor for a joinable thread is invoked, execution of the program is terminated.*” Reason of program execution termination.*

*Example of this: “*For example, suppose we have a function doWork that takes a filtering function, filter, and a maximum value, maxVal, as parameters. doWork checks to make sure that all conditions necessary for its computation are satisfied, then performs the computation with all the values between 0 and maxVal that pass the filter. If it’s time-consuming to do the filtering and it’s also time-consuming to determine whether doWork’s conditions are satisfied, it would be reasonable to do those two things concurrently.*” Time consuming is the reason of concurrency.*

*Example of using context of thread-based API: “*Our preference would be to employ a task-based design for this (see Item 35), but let’s assume we’d like to set the priority of the thread doing the filtering. Item 35 explains that that requires use of the thread’s native handle, and that’s accessible only through the std::thread API; the task-based API (i.e., futures) doesn’t provide it. Our approach will therefore be based on threads, not tasks.*”*

*Example continue. remin constexpr and std::function: “*We could come up with code like this:

constexpr auto tenMillion = 10000000; // see Item 15

// for constexpr

bool doWork(std::function<bool(int)> filter, // returns whether

int maxVal = tenMillion) // computation was

{ // performed; see

// Item 2 for

// std::function

std::vector<int> goodVals; // values that

// satisfy filter

std::thread t([&filter, maxVal, &goodVals] // populate

{ // goodVals

for (auto i = 0; i <= maxVal; ++i)

{ if (filter(i)) goodVals.push\_back(i); }

});

auto nh = t.native\_handle(); // use t's native

… // handle to set

// t's priority

if (conditionsAreSatisfied()) {

t.join(); // let t finish

performComputation(goodVals);

return true; // computation was

} // performed

return false; // computation was

} // not performed*”*

*p. 270*

*C++ history and aspects of number representation. We better see number of zeros: “*Before I explain why this code is problematic, I’ll remark that tenMillion’s initializing value can be made more readable in C++14 by taking advantage of C++14’s ability to use an apostrophe as a digit separator:

constexpr auto tenMillion = 10'000'000; // C++14*”*

*bad practice of multithreading: “*I’ll also remark that setting t’s priority after it has started running is a bit like closing the proverbial barn door after the equally proverbial horse has bolted. A better design would be to start t in a suspended state (thus making it possible to adjust its priority before it does any computation), but I don’t want to distract you with that code. If you’re more distracted by the code’s absence, turn to Item 39, because it shows how to start threads suspended.*” Book structure. Item (39). Start threads suspended. Intbu this.*

*Explanation of code badness: “*But back to doWork. If conditionsAreSatisfied() returns true, all is well, but if it returns false or throws an exception, the std::thread object t will be joinable when its destructor is called at the end of doWork. That would cause program execution to be terminated.*”*

*Std::thread structure: “*You might wonder why the std::thread destructor behaves this way. It’s because the two other obvious options are arguably worse. They are: • An implicit join. In this case, a std::thread’s destructor would wait for its underlying asynchronous thread of execution to complete. That sounds reasonable, but it could lead to performance anomalies that would be difficult to track down. For example, it would be counterintuitive that doWork would wait for its filter to be applied to all values if conditionsAreSatisfied() had already returned false.*” Intbu this property. I didn’t understand.*

*std::thread structure: “*An implicit detach. In this case, a std::thread’s destructor would sever the connection between the std::thread object and its underlying thread of execution. The underlying thread would continue to run. This sounds no less reasonable than the join approach, but the debugging problems it can lead to are worse.*”*

*p. 271*

*“*In doWork, for example, goodVals is a local variable that is captured by reference. It’s also modified inside the lambda (via the call to push\_back). Suppose, then, that while the lambda is running asynchronously, conditionsAreSatisfied() returns false. In that case, doWork would return, and its local variables (including goodVals) would be destroyed. Its stack frame would be popped, and execution of its thread would continue at doWork’s call site.*” Explanation of problem and example. There are choose of goodVlas in separate thread.*

*I understood implicit join. Thread doWork even if conditions are not satisfied will be continue up to finishing of thread t.*

*The second problem is if thread doWork will be destroyed asynchronous thread t can work with rubbish.*

“Statements following that call site would, at some point, make additional function calls, and at least one such call would probably end up using some or all of the memory that had once been occupied by the doWork stack frame. Let’s call such a function f. While f was running, the lambda that doWork initiated would still be running asynchronously. That lambda could call push\_back on the stack memory that used to be goodVals but that is now somewhere inside f’s stack frame. Such a call would modify the memory that used to be goodVals, and that means that from f’s perspective, the content of memory in its stack frame could spontaneously change! Imagine the fun you’d have debugging that.” *intbu this problem. What is stack frame?*

*C++ structure concerning multithreading: “*The Standardization Committee decided that the consequences of destroying a joinable thread were sufficiently dire that they essentially banned it (by specifying that destruction of a joinable thread causes program termination).*”*

*Problems of multithreading programming in C++. What is path? Intbu on examples: “*This puts the onus on you to ensure that if you use a std::thread object, it’s made unjoinable on every path out of the scope in which it’s defined. But covering every path can be complicated. It includes flowing off the end of the scope as well as jumping out via a return, continue, break, goto or exception. That can be a lot of paths.*” C++ multithreading programming.*

*Intbu this approach: “*Any time you want to perform some action along every path out of a block, the normal approach is to put that action in the destructor of a local object. Such objects are known as RAII objects, and the classes they come from are known as RAII classes. (RAII itself stands for “Resource Acquisition Is Initialization,” although the crux of the technique is destruction, not initialization).*” What are RAII objects?*

*RAII classes example and using context: “*RAII classes are common in the Standard Library. Examples include the STL containers (each container’s destructor destroys the container’s contents and releases its memory), the standard smart pointers (Items 18–20 explain that std::unique\_ptr’s destructor invokes its deleter on the object it points to, and the destructors in std::shared\_ptr and std::weak\_ptr decrement reference counts), std::fstream objects (their destructors close the files they correspond to), and many more.*” These classes while destruction clean memory.*

*Std::thread structure: “*And yet there is no standard RAII class for std::thread objects, perhaps because the Standardization Committee, having rejected both join and detach as default options, simply didn’t know what such a class should do.*” Intbu.*

*About RAII (*[*https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5\_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D0%B0\_%D0%B5%D1%81%D1%82%D1%8C\_%D0%B8%D0%BD%D0%B8%D1%86%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F*](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D0%B0_%D0%B5%D1%81%D1%82%D1%8C_%D0%B8%D0%BD%D0%B8%D1%86%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F)*)*

*p. 272*

*solution proposed by Mayers: “*Fortunately, it’s not difficult to write one yourself. For example, the following class allows callers to specify whether join or detach should be called when a Threa dRAII object (an RAII object for a std::thread) is destroyed:

class ThreadRAII {

public:

enum class DtorAction { join, detach }; // see Item 10 for

// enum class info

ThreadRAII(std::thread&& t, DtorAction a) // in dtor, take

: action(a), t(std::move(t)) {} // action a on t

~ThreadRAII()

{ // see below for

if (t.joinable()) { // joinability test

if (action == DtorAction::join) {

t.join();

} else {

t.detach();

}

}

}

std::thread& get() { return t; } // see below

private:

DtorAction action;

std::thread t;

};*” what is join and what is detach? Intbu these aspects of myltithreading.*

*About detach:* [*http://www.cplusplus.com/reference/thread/thread/detach/*](http://www.cplusplus.com/reference/thread/thread/detach/) *there is a very informative example.*

*Code description: “*I hope this code is largely self-explanatory, but the following points may be helpful:

• The constructor accepts only std::thread rvalues, because we want to move the passed-in std::thread into the ThreadRAII object. (Recall that std::thread objects aren’t copyable.)*” std::thread is move-only object.*

*Aspects of synchronization when std::thread is constructor parameter: “*The parameter order in the constructor is designed to be intuitive to callers (specifying the std::thread first and the destructor action second makes more sense than vice versa), but the member initialization list is designed to match the order of the data members’ declarations. That order puts the std::thread object last. In this class, the order makes no difference, but in general, it’s possible for the initialization of one data member to depend on another, and because std::thread objects may start running a function immediately after they are initialized, it’s a good habit to declare them last in a class. That guarantees that at the time they are constructed, all the data members that precede them have already been initialized and can therefore be safely accessed by the asynchronously running thread that corresponds to the std::thread data member.*”*

*p. 273*

*description of this solution: “*ThreadRAII offers a get function to provide access to the underlying std::thread object. This is analogous to the get functions offered by the standard smart pointer classes that give access to their underlying raw pointers. Providing get avoids the need for ThreadRAII to replicate the full std::thread interface, and it also means that ThreadRAII objects can be used in contexts where std::thread objects are required.*”*

*C++ undefined behavior: “*Before the ThreadRAII destructor invokes a member function on the std::thread object t, it checks to make sure that t is joinable. This is necessary, because invoking join or detach on an unjoinable thread yields undefined behavior.*”*

*Intbu this information about simultaneous member function calls safety: “*It’s possible that a client constructed a std::thread, created a ThreadRAII object from it, used get to acquire access to t, and then did a move from t or called join or detach on it. Each of those actions would render t unjoinable. If you’re worried that in this code,

if (t.joinable()) {

if (action == DtorAction::join) {

t.join();

} else {

t.detach();

}

}

a race exists, because between execution of t.joinable() and invocation of join or detach, another thread could render t unjoinable, your intuition is commendable, but your fears are unfounded. A std::thread object can change state from joinable to unjoinable only through a member function call, e.g., join, detach, or a move operation. At the time a ThreadRAII object’s destructor is invoked, no other thread should be making member function calls on that object. If there are simultaneous calls, there is certainly a race, but it isn’t inside the destructor, it’s in the client code that is trying to invoke two member functions (the destructor and something else) on one object at the same time. In general, simultaneous member function calls on a single object are safe only if all are to const member functions (see Item 16).*” Intbu this fragment.*

*p. 274*

*“*Employing ThreadRAII in our doWork example would look like this:

bool doWork(std::function<bool(int)> filter, // as before

int maxVal = tenMillion)

{

std::vector<int> goodVals; // as before

ThreadRAII t( // use RAII object

std::thread([&filter, maxVal, &goodVals]

{

for (auto i = 0; i <= maxVal; ++i)

{ if (filter(i)) goodVals.push\_back(i); }

}),

ThreadRAII::DtorAction::join // RAII action

);

auto nh = t.get().native\_handle();

…

if (conditionsAreSatisfied()) {

t.get().join();

performComputation(goodVals);

return true;

}

return false;

}

In this case, we’ve chosen to do a join on the asynchronously running thread in the ThreadRAII destructor, because, as we saw earlier, doing a detach could lead to some truly nightmarish debugging. We also saw earlier that doing a join could lead to performance anomalies (that, to be frank, could also be unpleasant to debug), but given a choice between undefined behavior (which detach would get us), program termination (which use of a raw std::thread would yield), or performance anomalies, performance anomalies seems like the best of a bad lot.*” Example of RAII std::thread objects using. Intbu about performance anomalies.*

*Pitfall of std::thread RAII objects: “*Alas, Item 39 demonstrates that using ThreadRAII to perform a join on std::thread destruction can sometimes lead not just to a performance anomaly, but to a hung program.*” Big programming error – hung program.*

*Solution of the problem: “*The “proper” solution to these kinds of problems would be to communicate to the asynchronously running lambda that we no longer need its work and that it should return early, but there’s no support in C++11 for interruptible threads. They can be implemented by hand, but that’s a topic beyond the scope of this book.*” C++ multithreading limitations – interruptible threads. Know reference 3 where there is information about their modeling.*

*p. 275*

*RAII std::thread objects structure: “*Item 17 explains that because ThreadRAII declares a destructor, there will be no compiler-generated move operations, but there is no reason ThreadRAII objects shouldn’t be movable. If compilers were to generate these functions, the functions would do the right thing, so explicitly requesting their creation isappropriate:

class ThreadRAII {

public:

enum class DtorAction { join, detach }; // as before

ThreadRAII(std::thread&& t, DtorAction a) // as before

: action(a), t(std::move(t)) {}

~ThreadRAII()

{

… // as before

}

ThreadRAII(ThreadRAII&&) = default; // support

ThreadRAII& operator=(ThreadRAII&&) = default; // moving

std::thread& get() { return t; } // as before

private: // as before

DtorAction action;

std::thread t;

};*”*

*Book structure. Resume: “*Things to Remember

• Make std::threads unjoinable on all paths. • join-on-destruction can lead to difficult-to-debug performance anomalies.

• detach-on-destruction can lead to difficult-to-debug undefined behavior.

• Declare std::thread objects last in lists of data members.*”*

## Item 38: Be aware of varying thread handle destructor behavior.

*Intbu what is “*thread handle destructor behavior*”*

*Std::thread and futures as hagnles to system threads: “*4 Item 39 explains that the kind of communications channel associated with a future can be employed for other purposes. For this Item, however, we’ll consider only its use as a mechanism for a callee to convey its result to a caller.

Item 38: Be aware of varying thread handle destructor

behavior.

Item 37 explains that a joinable std::thread corresponds to an underlying system thread of execution. A future for a non-deferred task (see Item 36) has a similar relationship to a system thread. As such, both std::thread objects and future objects can be thought of as handles to system threads.*”*

*Std::thread and future destructions: “*From this perspective, it’s interesting that std::threads and futures have such different behaviors in their destructors. As noted in Item 37, destruction of a joinable std::thread terminates your program, because the two obvious alternatives—an implicit join and an implicit detach—were considered worse choices. Yet the destructor for a future sometimes behaves as if it did an implicit join, sometimes as if it did an implicit detach, and sometimes neither. It never causes program termination. This thread handle behavioral bouillabaisse deserves closer examination.*”*

*Future structure. It is one end of communication chanell: “*We’ll begin with the observation that a future is one end of a communications channel through which a callee transmits a result to a caller.4 The callee (usually running asynchronously) writes the result of its computation into the communications channel (typically via a std::promise object), and the caller reads that result using a future. You can think of it as follows, where the dashed arrow shows the flow of information from callee to caller:

**

But where is the callee’s result stored? The callee could finish before the caller invokes get on a corresponding future, so the result can’t be stored in the callee’s std::promise. That object, being local to the callee, would be destroyed when the callee finished.*” Book question.*

*Std::future and std::shared\_future: “*The result can’t be stored in the caller’s future, either, because (among other reasons) a std::future may be used to create a std::shared\_future (thus transferring ownership of the callee’s result from the std::future to the std::shared\_future), which may then be copied many times after the original std::future is destroyed. Given that not all result types can be copied (i.e., move-only types) and that the result must live at least as long as the last future referring to it, which of the potentially many futures corresponding to the callee should be the one to contain its result?*” book question. What is purpose of std::shared\_future?*

*p. 277*

*future realization. Shared state: “*Because neither objects associated with the callee nor objects associated with the caller are suitable places to store the callee’s result, it’s stored in a location outside both. This location is known as the shared state. The shared state is typically represented by a heap-based object, but its type, interface, and implementation are not specified by the Standard. Standard Library authors are free to implement shared states in any way they like.*” Intbu why the result of callee can’t be stored in caller?*

*Future realization: “*We can envision the relationship among the callee, the caller, and the shared state as

follows, where dashed arrows once again represent the flow of information:

**

*”*

*Importance of the shared state and std::future destruction. This is the topic of this Item: “*The existence of the shared state is important, because the behavior of a future’s destructor—the topic of this Item—is determined by the shared state associated with the future. In particular, • The destructor for the last future referring to a shared state for a nondeferred task launched via std::async blocks until the task completes. In essence, the destructor for such a future does an implicit join on the thread on which the asynchronously executing task is running.

• The destructor for all other futures simply destroys the future object. For asynchronously running tasks, this is akin to an implicit detach on the underlying thread. For deferred tasks for which this is the final future, it means that the deferred task will never run.*” Intbu the last sentence about deferred task.*

*Normal future destruction: “*These rules sound more complicated than they are. What we’re really dealing with is a simple “normal” behavior and one lone exception to it. The normal behavior is that a future’s destructor destroys the future object. That’s it. It doesn’t join with anything, it doesn’t detach from anything, it doesn’t run anything. It just destroys the future’s data members.*” What is thread join to something? Intbu.*

*“*(Well, actually, it does one more thing. It decrements the reference count inside the shared state that’s manipulated by both the futures referring to it and the callee’s std::promise. This reference count makes it possible for the library to know when the shared state can be destroyed. For general information about reference counting, see Item 19.)*” future structure. Intbu. What is std::promise???*

*p. 278*

*special cases of future destruction: “*The exception to this normal behavior arises only for a future for which all of the following apply: It refers to a shared state that was created due to a call to std::async.

• The task’s launch policy is std::launch::async (see Item 36), either because that was chosen by the runtime system or because it was specified in the call to std::async.

• The future is the last future referring to the shared state. For std::futures, this will always be the case. For std::shared\_futures, if other std::shared\_ futures refer to the same shared state as the future being destroyed, the future being destroyed follows the normal behavior (i.e., it simply destroys its data members). *” difference between std::future and std::shared\_future.*

*Std::future destruction special case and for explanation of join concept: “*Only when all of these conditions are fulfilled does a future’s destructor exhibit special behavior, and that behavior is to block until the asynchronously running task completes. Practically speaking, this amounts to an implicit join with the thread running the std::async-created task.*” Join means to run the task?*

*Example of first approximation and other name of the given phenomenon: “*It’s common to hear this exception to normal future destructor behavior summarized as “Futures from std::async block in their destructors.” To a first approximation, that’s correct, but sometimes you need more than a first approximation. Now you know the truth in all its glory and wonder.*”*

*C++ history and examplanation of such std::future destructor behavior. Intbu what is implicit detach: “*Your wonder may take a different form. It may be of the “I wonder why there’s a special rule for shared states for non-deferred tasks that are launched by std::async”

variety. It’s a reasonable question. From what I can tell, the Standardization Committee wanted to avoid the problems associated with an implicit detach (see Item 37), but they didn’t want to adopt as radical a policy as mandatory program termination (as they did for joinable std::threads—again, see Item 37), so they compromised on an implicit join. The decision was not without controversy, and there was serious talk about abandoning this behavior for C++14. In the end, no change was made, so the behavior of destructors for futures is consistent in C++11 and C++14.*”*

*Std::future drawback: “*The API for futures offers no way to determine whether a future refers to a shared state arising from a call to std::async, so given an arbitrary future object, it’s not possible to know whether it will block in its destructor waiting for an asynchronously running task to finish. “

*example of this:* “This has some interesting implications:

// this container might block in its dtor, because one or more

// contained futures could refer to a shared state for a non//

deferred task launched via std::async

std::vector<std::future<void>> futs; // see Item 39 for info

// on std::future<void>

class Widget { // Widget objects might

public: // block in their dtors

…

private:

std::shared\_future<double> fut;

}; *”*

*p. 279*

*shared states creation cases: “*Of course, if you have a way of knowing that a given future does not satisfy the conditions that trigger the special destructor behavior (e.g., due to program logic), you’re assured that that future won’t block in its destructor. For example, only shared states arising from calls to std::async qualify for the special behavior, but there are other ways that shared states get created.*” Intbu what is shared state creation via std::async.*

*Another share state creation case: “*One is the use of std::packaged\_task. A std::packaged\_task object prepares a function (or other callable object) for asynchronous execution by wrapping it such that its result is put into a shared state. A future referring to that shared state can then be obtained via std::packaged\_task’s get\_future function:

int calcValue(); // func to run

std::packaged\_task<int()> // wrap calcValue so it

pt(calcValue); // can run asynchronously

auto fut = pt.get\_future(); // get future for pt

At this point, we know that the future fut doesn’t refer to a shared state created by a call to std::async, so its destructor will behave normally.*” Intbu the last sentence. What is std::packaged\_task???*

*Structure of std::packaged\_task: “*Once created, the std::packaged\_task pt can be run on a thread. (It could be run via a call to std::async, too, but if you want to run a task using std::async, there’s little reason to create a std::packaged\_task, because std::async does everything std::packaged\_task does before it schedules the task for execution.)*” intbu difference between std::package\_task and std::async.*

*Std::packaged\_task property: “*std::packaged\_tasks aren’t copyable, so when pt is passed to the std::thread constructor, it must be cast to an rvalue (via std::move—see Item 23):

std::thread t(std::move(pt)); // run pt on t*”*

This example…

p. 279

*18.01.2017*

*“*This example lends some insight into the normal behavior for future destructors, but it’s easier to see if the statements are put together inside a block:

{ // begin block

std::packaged\_task<int()>

pt(calcValue);

auto fut = pt.get\_future();

std::thread t(std::move(pt));

… // see below

} // end block *” I don’t understand pupose of this example. But I understand that we create std::packaged\_task. Then we save its future in fut variable. And then we move this task to std::thread. There is also some block.*

*p. 280*

*intbu this example and its behavior: “*The most interesting code here is the “…” that follows creation of the std::thread object t and precedes the end of the block. What makes it interesting is what can happen to t inside the “…” region. There are three basic possibilities:

• Nothing happens to t. In this case, t will be joinable at the end of the scope. That will cause the program to be terminated (see Item 37).

• A join is done on t. In this case, there would be no need for fut to block in its destructor, because the join is already present in the calling code.

• A detach is done on t. In this case, there would be no need for fut to detach in its destructor, because the calling code already does that.*”*

*Intbu these aspects of future, thread and packaged\_task: “*In other words, when you have a future corresponding to a shared state that arose due to a std::packaged\_task, there’s usually no need to adopt a special destruction policy, because the decision among termination, joining, or detaching will be made in the code that manipulates the std::thread on which the std::packaged\_task is typically run.*”*

*Book structure. Resume: “*Things to Remember

• Future destructors normally just destroy the future’s data members.

• The final future referring to a shared state for a non-deferred task launched via std::async blocks until the task completes.*”*

## Item 39: Consider void futures for one-shot event communication.

*What is one shot event communication?*

*What is void futures?*

*“*Sometimes it’s useful for a task to tell a second, asynchronously running task that a particular event has occurred, because the second task can’t proceed until the event has taken place. Perhaps a data structure has been initialized, a stage of computation has been completed, or a significant sensor value has been detected. When that’s the case, what’s the best way for this kind of inter-thread communication to take place?*” taks formulation. Inter-thread communication.*

*Condition variable (condvar) as solution: “*An obvious approach is to use a condition variable (condvar). If we call the task that detects the condition the detecting task and the task reacting to the condition the reacting task, the strategy is simple: the reacting task waits on a condition variable, and the detecting thread notifies that condvar when the event occurs.*” It’s variable accessible to both threads?*

*p. 281*

*std::condition\_variable, std::mutext. What are they?: “*Given

std::condition\_variable cv; // condvar for event

std::mutex m; // mutex for use with cv

the code in the detecting task is as simple as simple can be:

… // detect event

cv.notify\_one(); // tell reacting task*” intbu this code.*

*Variance of the task: “*If there were multiple reacting tasks to be notified, it would be appropriate to replace notify\_one with notify\_all, but for now, we’ll assume there’s only one reacting task.*” book assumption*

*intbu code here:* [*http://ru.cppreference.com/w/cpp/thread/condition\_variable*](http://ru.cppreference.com/w/cpp/thread/condition_variable)

The code for the…

p. 281

*19.01.2017*

*C++ concurrent API. Intbu std::unique\_lock and principle of mutex work: “*The code for the reacting task is a bit more complicated, because before calling wait on the condvar, it must lock a mutex through a std::unique\_lock object. (Locking a mutex before waiting on a condition variable is typical for threading libraries. The need to lock the mutex through a std::unique\_lock object is simply part of the C++11 API.)*”*

*Intbu critical section. Book example: “*Here’s the conceptual approach:

… // prepare to react

{ // open critical section

std::unique\_lock<std::mutex> lk(m); // lock mutex

cv.wait(lk); // wait for notify;

// this isn't correct!

… // react to event

// (m is locked)

} // close crit. section;

// unlock m via lk's dtor

… // continue reacting

// (m now unlocked)*”*

*Example explanation: “*The first issue with this approach is what’s sometimes termed a code smell: even if the code works, something doesn’t seem quite right. In this case, the odor emanates from the need to use a mutex. Mutexes are used to control access to shared data, but it’s entirely possible that the detecting and reacting tasks have no need for such mediation. For example, the detecting task might be responsible for initializing a global data structure, then turning it over to the reacting task for use. If the detecting task never accesses the data structure after initializing it, and if the reacting task never accesses it before the detecting task indicates that it’s ready, the two tasks will stay out of each other’s way through program logic. There will be no need for a mutex. The fact that the condvar approach requires one leaves behind the unsettling aroma of suspect design.*” mutex purpose and example of task interaction. Critica of C++ concurrent API. What is condvar conception? Intbu.*

*p. 282*

*example explanation and two another problems demonstrated in it: “*Even if you look past that, there are two other problems you should definitely pay attention to: *”*

*problem of hang thread: “*If the detecting task notifies the condvar before the reacting task waits, the reacting task will hang. In order for notification of a condvar to wake another task, the other task must be waiting on that condvar. If the detecting task happens to execute the notification before the reacting task executes the wait, the reacting task will miss the notification, and it will wait forever.*”*

*spurious wakeups – concept of threading programming : “*A fact of life in threading APIs (in many languages—not just C++) is that code waiting on a condition variable may be awakened even if the condvar wasn’t notified. Such awakenings are known as spurious wakeups. Proper code deals with them by confirming that the condition being waited for has truly occurred, and it does this as its first action after waking.*”*

*Intbu this realization of spurious wakeup: “*The C++ condvar API makes this exceptionally easy, because it permits a lambda (or other function object) that tests for the waited-for condition to be passed to wait. That is, the wait call in the reacting task could be written like this:

cv.wait(lk,

[]{ return whether the event has occurred; });*”*

*Example explanation. Does it concerns hang program problem too?: “*Taking advantage of this capability requires that the reacting task be able to determine whether the condition it’s waiting for is true. But in the scenario we’ve been considering, the condition it’s waiting for is the occurrence of an event that the detecting thread is responsible for recognizing. The reacting thread may have no way of determining whether the event it’s waiting for has taken place. That’s why it’s waiting on a condition variable!*”*

*intbu translation: “*There are many situations where having tasks communicate using a condvar is a good fit for the problem at hand, but this doesn’t seem to be one of them.*”*

*What is shared boolean flag?: “*For many developers, the next trick in their bag is a shared boolean flag. The flag is initially false. When the detecting thread recognizes the event it’s looking for, it sets the flag:*”*

*demonstrational example: “*std::atomic<bool> flag(false); // shared flag; see

// Item 40 for std::atomic

… // detect event

flag = true; // tell reacting task *” it’s implemented with std::atomic.*

*p. 283*

*example continue: “*For its part, the reacting thread simply polls the flag. When it sees that the flag is set, it knows that the event it’s been waiting for has occurred:

… // prepare to react

while (!flag); // wait for event

… // react to event*” I understood this mechanism.*

*Advantages of this mechanism: “*This approach suffers from none of the drawbacks of the condvar-based design. There’s no need for a mutex, no problem if the detecting task sets the flag before the reacting task starts polling, and nothing akin to a spurious wakeup. Good, good, good.*” Intbu condvar design.*

*Drawbacks of mechanism: “*Less good is the cost of polling in the reacting task. During the time the task is waiting for the flag to be set, the task is essentially blocked, yet it’s still running. As such, it occupies a hardware thread that another task might be able to make use of, it incurs the cost of a context switch each time it starts or completes its time-slice, and it could keep a core running that might otherwise be shut down to save power.*” This mechanism is blocking imitation.*

*Concurrency concept. Truly blocked task: “*A truly blocked task would do none of these things. That’s an advantage of the condvarbased approach, because a task in a wait call is truly blocked.*” Advantage of condvar-based approach.*

*Combined solution: “*It’s common to combine the condvar and flag-based designs. A flag indicates whether the event of interest has occurred, but access to the flag is synchronized by a mutex. Because the mutex prevents concurrent access to the flag, there is, as Item 40 explains, no need for the flag to be std::atomic; a simple bool will do.*”*

*It’s example: “*The detecting task would then look like this:

std::condition\_variable cv; // as before

std::mutex m;

bool flag(false); // not std::atomic

… // detect event

{

std::lock\_guard<std::mutex> g(m); // lock m via g's ctor

flag = true; // tell reacting task

// (part 1)

} // unlock m via g's dtor*” mutex realization in C++.*

*p. 284*

*example continue: “*cv.notify\_one(); // tell reacting task

// (part 2)

And here’s the reacting task:

… // prepare to react

{ // as before

std::unique\_lock<std::mutex> lk(m); // as before

cv.wait(lk, [] { return flag; }); // use lambda to avoid

// spurious wakeups

… // react to event

// (m is locked)

}

… // continue reacting

// (m now unlocked)*” intbu how do we avoid spurious wakeup and why is it bad?*

*Solution advantages (3): “*This approach avoids the problems we’ve discussed. It works regardless of whether the reacting task waits before the detecting task notifies, it works in the presence of spurious wakeups, and it doesn’t require polling.*”*

*Approach drawbacks: “*Yet an odor remains, because the

detecting task communicates with the reacting task in a very curious fashion. Notifying the condition variable tells the reacting task that the event it’s been waiting for has probably occurred, but the reacting task must check the flag to be sure. Setting the flag tells the reacting task that the event has definitely occurred, but the detecting task still has to notify the condition variable so that the reacting task will awaken and check the flag. The approach works, but it doesn’t seem terribly clean.*”*

An alternative is to avoid…

*19.01.2017*

*Another mechanism: “*An alternative is to avoid condition variables, mutexes, and flags by having the reacting task wait on a future that’s set by the detecting task. This may seem like an odd idea. After all, Item 38 explains that a future represents the receiving end of a communications channel from a callee to a (typically asynchronous) caller, and here there’s no callee-caller relationship between the detecting and reacting tasks.*”*

*С++ concurrent structure. Communication channel “future-promise” and its purpose: “*However, Item 38 also notes that a communications channel whose transmitting end is a std::promise and whose receiving end is a future can be used for more than just callee-caller communication. Such a communications channel can be used in any situation where you need to transmit information from one place in your program to another. In this case, we’ll use it to transmit information from the detecting task to the reacting task, and the information we’ll convey will be that the event of interest has taken place.*”*

*design of this channel purpose: “*The design is simple. The detecting task has a std::promise object (i.e., the writing end of the communications channel), and the reacting task has a corresponding future. When the detecting task sees that the event it’s looking for has occurred, it sets the std::promise (i.e., writes into the communications channel). Meanwhile, the reacting task waits on its future. That wait blocks the reacting task until the std::promise has been set.*” For explanation of future mechanism.*

*p. 285*

*realization of this design: “*Now, both std::promise and futures (i.e., std::future and std::shared\_future) are templates that require a type parameter. That parameter indicates the type of data to be transmitted through the communications channel. In our case, however, there’s no data to be conveyed. The only thing of interest to the reacting task is that its future has been set. What we need for the std::promise and future templates is a type that indicates that no data is to be conveyed across the communications channel. That type is void. The detecting task will thus use a std::promise<void>, and the reacting task a std::future<void> or std::shared\_future<void>. The detecting task will set its std::promise<void> when the event of interest occurs, and the reacting task will wait on its future. Even though the reacting task won’t receive any data from the detecting task, the communications channel will permit the reacting task to know when the detecting task has “written” its void data by calling set\_value on its std::promise.*” Std::future<void> explanation ans general explanation of future mechanism.*

*“*So given

std::promise<void> p; // promise for

// communications channel

the detecting task’s code is trivial,

… // detect event

p.set\_value(); // tell reacting task

and the reacting task’s code is equally simple:

… // prepare to react

p.get\_future().wait(); // wait on future

// corresponding to p

… // react to event*” realization of this design and future structure.*

*Advantages of this solution: “*Like the approach using a flag, this design requires no mutex, works regardless of whether the detecting task sets its std::promise before the reacting task waits, and is immune to spurious wakeups. (Only condition variables are susceptible to that problem.) Like the condvar-based approach, the reacting task is truly blocked after making the wait call, so it consumes no system resources while waiting. Perfect, right?*” what is spurious wakeups? Why are they so dangerous?*

*p. 286*

*nuances of this design:* “Not exactly. Sure, a future-based approach skirts those shoals, but there are other hazards to worry about. For example, Item 38 explains that between a std::promise and a future is a shared state, and shared states are typically dynamically allocated. You should therefore assume that this design incurs the cost of heap-based allocation and deallocation.”

*Characteristic of this design. Kind of informational chanel – one-shot mechanism: “*Perhaps more importantly, a std::promise may be set only once. The communications channel between a std::promise and a future is a one-shot mechanism: it can’t be used repeatedly. This is a notable difference from the condvar- and flag-based designs, both of which can be used to communicate multiple times. (A condvar can be repeatedly notified, and a flag can always be cleared and set again.)*” drawback of this mechanism and advantages of that.*

*Kind of thread state: “*The one-shot restriction isn’t as limiting as you might think. Suppose you’d like to create a system thread in a suspended state. That is, you’d like to get all the overhead associated with thread creation out of the way so that when you’re ready to execute something on the thread, the normal thread-creation latency will be avoided. Or you might want to create a suspended thread so that you could configure it before letting it run. Such configuration might include things like setting its priority or core affinity.*” Using context of suspended thread state (2). What are thread priority and core affinity?*

*C++ concurrency API: “*The C++ concurrency API offers no way to do those things, but std::thread objects offer the native\_handle member function, the result of which is intended to give you access to the platform’s underlying threading API (usually POSIX threads or Windows threads). The lower-level API often makes it possible to configure thread characteristics such as priority and affinity.*” Access to native API.*

*Essense of std::future<void> mechanism: “*Assuming you want to suspend a thread only once (after creation, but before it’s running its thread function), a design using a void future is a reasonable choice. Here’s the essence of the technique:

std::promise<void> p;

void react(); // func for reacting task

void detect() // func for detecting task

{

std::thread t([] // create thread

{

p.get\_future().wait(); // suspend t until

react(); // future is set

});

… // here, t is suspended

// prior to call to react

p.set\_value(); // unsuspend t (and thus

// call react)

… // do additional work

t.join(); // make t unjoinable

} // (see Item 37)*” there are two threads correspond two functions. Without join() program will terminate as far as I remember.*

*p. 287*

*nuances of RAII object using context: “*Because it’s important that t become unjoinable on all paths out of detect, use of an RAII class like Item 37’s ThreadRAII seems like it would be advisable. Code like this comes to mind:

void detect()

{

ThreadRAII tr( // use RAII object

std::thread([]

{

p.get\_future().wait();

react();

}),

ThreadRAII::DtorAction::join // risky! (see below)

);

… // thread inside tr

// is suspended here

p.set\_value(); // unsuspend thread

// inside tr

…

}*”*

*Problem of this solution. Exception structure: “*This looks safer than it is. The problem is that if in the first “…” region (the one with the “thread inside tr is suspended here” comment), an exception is emitted, set\_value will never be called on p. That means that the call to wait inside the lambda will never return. That, in turn, means that the thread running the lambda will never finish, and that’s a problem, because the RAII object tr has been configured to perform a join on that thread in tr’s destructor. In other words, if an exception is emitted from the first “…” region of code, this function will hang, because tr’s destructor will never complete.*” Example of error: hang fuction.*

*Book task: “*There are ways to address this problem, but I’ll leave them in the form of the hallowed exercise for the reader.5 Here, I’d like to show how the original code (i.e., not using ThreadRAII) can be extended to suspend and then unsuspend not just one reacting task, but many.*” using one-shot chanel for modelling of multy-shot channel.*

*p. 288*

*solution of this task: “*It’s a simple generalization, because the key is to use std::shared\_futures instead of a std::future in the react code. Once you know that the std::future’s share member function transfers ownership of its shared state to the std::shared\_future object produced by share, the code nearly writes itself. The only subtlety is that each reacting thread needs its own copy of the std::shared\_future that refers to the shared state, so the std::shared\_future obtained from share is captured by value by the lambdas running on the reacting threads:

std::promise<void> p; // as before

void detect() // now for multiple

{ // reacting tasks

auto sf = p.get\_future().share(); // sf's type is

// std::shared\_future<void>

std::vector<std::thread> vt; // container for

// reacting threads

for (int i = 0; i < threadsToRun; ++i) {

vt.emplace\_back([sf]{ sf.wait(); // wait on local

react(); }); // copy of sf; see

} // Item 42 for info

// on emplace\_back

… // detect hangs if

// this "…" code throws!

p.set\_value(); // unsuspend all threads

…

for (auto& t : vt) { // make all threads

t.join(); // unjoinable; see Item 2

} // for info on "auto&"

}

The fact that a design using futures can achieve this effect is noteworthy, and that’s why you should consider it for one-shot event communication.*” Book recommendation. Std::shared\_future structure. Like shared pointer but future. It’s evident from code.*

*p. 289*

*book structure. Resume: “*Things to Remember

• For simple event communication, condvar-based designs require a superfluous mutex, impose constraints on the relative progress of detecting and reacting tasks, and require reacting tasks to verify that the event has taken place.

• Designs employing a flag avoid those problems, but are based on polling, not blocking.

• A condvar and flag can be used together, but the resulting communications mechanism is somewhat stilted.

• Using std::promises and futures dodges these issues, but the approach uses heap memory for shared states, and it’s limited to one-shot communication.*” Example of ideally good solutions absence.*

## Item 40: Use std::atomic for concurrency, volatile for special memory.

*c. 289*

*20.01.2017*

*“*Poor volatile. So misunderstood. It shouldn’t even be in this chapter, because it has nothing to do with concurrent programming. But in other programming languages (e.g., Java and C#), it is useful for such programming, and even in C++, some compilers have imbued volatile with semantics that render it applicable to concurrent software (but only when compiled with those compilers). It’s thus worthwhile to discuss volatile in a chapter on concurrency if for no other reason than to dispel the confusion surrounding it.*” (aspects of C# and Java. Volatile and concurrent programming in different languages. Intbu.)*

*“*The C++ feature that programmers sometimes confuse volatile with—the feature that definitely does belong in this chapter—is the std::atomic template.*” (frequent misunderstanding.)*

“Instantiations of this template (e.g., std::atomic<int>, std::atomic<bool>, std::atomic<Widget\*>, etc.) offer operations that are guaranteed to be seen as atomic by other threads. Once a std::atomic object has been constructed, operations on it behave as if they were inside a mutex-protected critical section, but the operations are generally implemented using special machine instructions that are more efficient than would be the case if a mutex were employed.” (*std::atomic description and realization*)

*“*Consider this code using std::atomic:

std::atomic<int> ai(0); // initialize ai to 0

ai = 10; // atomically set ai to 10

std::cout << ai; // atomically read ai's value

++ai; // atomically increment ai to 11

--ai; // atomically decrement ai to 10

During execution of these statements, other threads reading ai may see only values of 0, 10, or 11. No other values are possible (assuming, of course, that this is the only thread modifying ai).*” (std::atomic demonstrating example)*

*p. 290*

*std::atomic description. Other threads can modify atomic values: “*Two aspects of this example are worth noting. First, in the “std::cout << ai;” statement, the fact that ai is a std::atomic guarantees only that the read of ai is atomic. There is no guarantee that the entire statement proceeds atomically. Between the time ai’s value is read and operator<< is invoked to write it to the standard output, another thread may have modified ai’s value. That has no effect on the behavior of the statement, because operator<< for ints uses a by-value parameter for the int to output (the outputted value will therefore be the one that was read from ai), but it’s important to understand that what’s atomic in that statement is nothing more than the read of ai.*” (but they can’t while operation with atomic is performed in the given thread)*

*“*The second noteworthy aspect of the example is the behavior of the last two statements—the increment and decrement of ai. These are each read-modify-write (RMW) operations, yet they execute atomically.*” (example description. Atomic structure. Class of operations – RMW (read-modify-write)).*

*“*This is one of the nicest characteristics of the std::atomic types: once a std::atomic object has been constructed, all member functions on it, including those comprising RMW operations, are guaranteed to be seen by other threads as atomic.*” (std::atomic structure and example)*

*“*In contrast, the corresponding code using volatile guarantees virtually nothing in a multithreaded context:

volatile int vi(0); // initialize vi to 0

vi = 10; // set vi to 10

std::cout << vi; // read vi's value

++vi; // increment vi to 11

--vi; // decrement vi to 10

During execution of this code, if other threads are reading the value of vi, they may see anything, e.g, -12, 68, 4090727—anything! Such code would have undefined behavior, because these statements modify vi, so if other threads are reading vi at the same time, there are simultaneous readers and writers of memory that’s neither std::atomic nor protected by a mutex, and that’s the definition of a data race.*” (data race problem definition. C++ undefined behavior. Volatile properties in the concurrency context)*

*p. 291*

*“*As a concrete example of how the behavior of std::atomics and volatiles can differ in a multithreaded program, consider a simple counter of each type that’s incremented by multiple threads. We’ll initialize each to 0:

std::atomic<int> ac(0); // "atomic counter"

volatile int vc(0); // "volatile counter"

We’ll then increment each counter one time in two simultaneously running threads:

/\*----- Thread 1 ----- \*/ /\*------- Thread 2 ------- \*/

++ac; ++ac;

++vc; ++vc;

When both threads have finished, ac’s value (i.e., the value of the std::atomic) must be 2, because each increment occurs as an indivisible operation. vc’s value, on the other hand, need not be 2, because its increments may not occur atomically. Each increment consists of reading vc’s value, incrementing the value that was read, and writing the result back into vc. But these three operations are not guaranteed to proceed atomically for volatile objects, so it’s possible that the component parts of the two increments of vc are interleaved as follows:

1. Thread 1 reads vc’s value, which is 0.

2. Thread 2 reads vc’s value, which is still 0.

3. Thread 1 increments the 0 it read to 1, then writes that value into vc.

4. Thread 2 increments the 0 it read to 1, then writes that value into vc.

vc’s final value is therefore 1, even though it was incremented twice.

This is not the only possible outcome. vc’s final value is, in general, not predictable, because vc is involved in a data race, and the Standard’s decree that data races cause undefined behavior means that compilers may generate code to do literally anything. Compilers don’t use this leeway to be malicious, of course. Rather, they perform optimizations that would be valid in programs without data races, and these optimizations yield unexpected and unpredictable behavior in programs where races are present.*” (classical example of data race. It’s undefined behavior in C++. Volatile is not atomic.)*

*“*The use of RMW operations isn’t the only situation where std::atomics comprise a concurrency success story and volatiles suffer failure. Suppose one task computes an important value needed by a second task. When the first task has computed the value, it must communicate this to the second task. Item 39 explains that one way for the first task to communicate the availability of the desired value to the second task is by using a std::atomic<bool>.*” (another using context of std::atomic and failure context of volatile)*

*p. 292*

*“*Code in the task computing the value would look something like this:

std::atomic<bool> valAvailable(false);

auto imptValue = computeImportantValue(); // compute value

valAvailable = true; // tell other task

// it's available*” (example of it)*

*“*As humans reading this code, we know it’s crucial that the assignment to imptValue take place before the assignment to valAvailable, but all compilers see is a pair of assignments to independent variables. As a general rule, compilers are permitted to reorder such unrelated assignments. That is, given this sequence of assignments (where a, b, x, and y correspond to independent variables),

a = b;

x = y;

compilers may generally reorder them as follows:

x = y;

a = b;*” (example description and interesting compiler feature. It can reorder independent statements)*

*“*Even if compilers don’t reorder them, the underlying hardware might do it (or might make it seem to other cores as if it had), because that can sometimes make the code run faster.*” (hardware also can reorder operations. Intbu this computer peculiarity)*

*“*However, the use of std::atomics imposes restrictions on how code can be reordered, and one such restriction is that no code that, in the source code, precedes a write of a std::atomic variable may take place (or appear to other cores to take place) afterwards.6 That means that in our code,

auto imptValue = computeImportantValue(); // compute value

valAvailable = true; // tell other task

// it's available

not only must compilers retain the order of the assignments to imptValue and valAvailable, they must generate code that ensures that the underlying hardware does, too. As a result, declaring valAvailable as std::atomic ensures that our critical ordering requirement—imptValue must be seen by all threads to change no later than valAvailable does—is maintained.*” (std::atomic realization. Restrictions on reordering)*

*“*This is true only for

std::atomics using sequential consistency, which is both the default and the only consistency model for std::atomic objects that use the syntax shown in this book. C++11 also supports consistency models with more flexible code-reordering rules. Such weak (aka relaxed) models make it possible to create software that runs faster on some hardware architectures, but the use of such models yields software that is much more difficult to get right, to understand, and to maintain. Subtle errors in code using relaxed atomics is not uncommon, even for experts, so you should stick to sequential consistency if at all possible.*” C++ concurrency realization and C++ std::atomic structure.*

*p. 293*

*“*Declaring valAvailable as volatile doesn’t impose the same code reordering restrictions:

volatile bool valAvailable(false);

auto imptValue = computeImportantValue();

valAvailable = true; // other threads might see this assignment

// before the one to imptValue!

Here, compilers might flip the order of the assignments to imptValue and valAvail able, and even if they don’t, they might fail to generate machine code that would prevent the underlying hardware from making it possible for code on other cores to see valAvailable change before imptValue. These two issues—no guarantee of operation atomicity and insufficient restrictions on code reordering—explain why volatile’s not useful for concurrent programming, but it doesn’t explain what it is useful for.*” (volatile failure context)*

*“*In a nutshell, it’s for telling compilers that they’re dealing with memory that doesn’t behave normally.*” (purpose of volatile)*

*“*“Normal” memory has the characteristic that if you write a value to a memory location, the value remains there until something overwrites it. So if I have a normal int,

int x;

and a compiler sees the following sequence of operations on it,

auto y = x; // read x

y = x; // read x again

the compiler can optimize the generated code by eliminating the assignment to y, because it’s redundant with y’s initialization.*” (compiler work and explanation of normal memory management in C++ programs)*

*“*Normal memory also has the characteristic that if you write a value to a memory location, never read it, and then write to that memory location again, the first write can be eliminated, because it was never used. So given these two adjacent statements,

x = 10; // write x

x = 20; // write x again

compilers can eliminate the first one.*” (another normal memory management in C++)*

*“*Normal memory also has the characteristic that if you write a value to a memory location, never read it, and then write to that memory location again, the first write can be eliminated, because it was never used. So given these two adjacent statements,

x = 10; // write x

x = 20; // write x again

compilers can eliminate the first one. That means that if we have this in the source code,

auto y = x; // read x

y = x; // read x again

x = 10; // write x

x = 20; // write x again

compilers can treat it as if it had been written like this:

auto y = x; // read x

x = 20; // write x*” (compiler memory government)*

*p. 294*

*“*Lest you wonder who’d write code that performs these kinds of redundant reads and superfluous writes (technically known as redundant loads and dead stores), the answer is that humans don’t write it directly—at least we hope they don’t.*” (C++ bad practices – redundant loads and dead stores)*

*“*However, after compilers take reasonable-looking source code and perform template instantiation, inlining, and various common kinds of reordering optimizations, it’s not uncommon for the result to have redundant loads and dead stores that compilers can get rid of.*” (compiler work. Such problems can be as a result of compiler work)*

*“*Such optimizations are valid only if memory behaves normally. “Special” memory doesn’t. Probably the most common kind of special memory is memory used for memory-mapped I/O. Locations in such memory actually communicate with peripherals, e.g., external sensors or displays, printers, network ports, etc. rather than reading or writing normal memory (i.e., RAM).*” (C++ structure. Special memory. Memory-mapped I/O. intbu.)*

*“*In such a context, consider again the code with seemingly redundant reads:

auto y = x; // read x

y = x; // read x again

If x corresponds to, say, the value reported by a temperature sensor, the second read of x is not redundant, because the temperature may have changed between the first and second reads. It’s a similar situation for seemingly superfluous writes. In this code, for example,

x = 10; // write x

x = 20; // write x again

if x corresponds to the control port for a radio transmitter, it could be that the code is issuing commands to the radio, and the value 10 corresponds to a different command from the value 20. Optimizing out the first assignment would change the sequence of commands sent to the radio.*” (examples of special memory)*

*“*volatile is the way we tell compilers that we’re dealing with special memory. Its meaning to compilers is “Don’t perform any optimizations on operations on this memory.” So if x corresponds to special memory, it’d be declared volatile:

volatile int x; *” (volatile purpose)*

*p. 295*

*“*Consider the effect that has on our original code sequence:

auto y = x; // read x

y = x; // read x again (can't be optimized away)

x = 10; // write x (can't be optimized away)

x = 20; // write x again

This is precisely what we want if x is memory-mapped (or has been mapped to a memory location shared across processes, etc.). *” (example of volatile)*

*“*Pop quiz! In that last piece of code, what is y’s type: int or volatile int?7*” (book structure.)*

*“*y’s type is auto-deduced, so it uses the rules described in Item 2. Those rules dictate that for the declaration of non-reference non-pointer types (which is the case for y), const and volatile qualifiers are dropped. y’s type is therefore simply int. This means that redundant reads of and writes to y can be eliminated. In the example, compilers must perform both the initialization of and the assignment to y, because x is volatile, so the second read of x might yield a different value from the first one.*” (auto type deducing. Volatile are dropped.)*

*“*The fact that seemingly redundant loads and dead stores must be preserved when dealing with special memory explains, by the way, why std::atomics are unsuitable for this kind of work. Compilers are permitted to eliminate such redundant operations on std::atomics. The code isn’t written quite the same way it is for vola tiles, but if we overlook that for a moment and focus on what compilers are permitted to do, we can say that, conceptually, compilers may take this,*” (std::atomic structure and failure context)*

*“*std::atomic<int> x;

auto y = x; // conceptually read x (see below)

y = x; // conceptually read x again (see below)

x = 10; // write x

x = 20; // write x again

and optimize it to this:

auto y = x; // conceptually read x (see below)

x = 20; // write x

For special memory, this is clearly unacceptable behavior.*” (example of it)*

*“*Now, as it happens, neither of these two statements will compile when x is std::atomic:

auto y = x; // error!

y = x; // error!

That’s because the copy operations for std::atomic are deleted (see Item 11). And with good reason.*” (std::atomic structure and limitations. C++ structure. Assignment operator structure.)*

*p. 296*

*“*Consider what would happen if the initialization of y with x compiled. Because x is std::atomic, y’s type would be deduced to be std::atomic, too (see Item 2). I remarked earlier that one of the best things about std::atomics is that all their operations are atomic, but in order for the copy construction of y from x to be atomic, compilers would have to generate code to read x and write y in a single atomic operation. Hardware generally can’t do that, so copy construction isn’t supported for std::atomic types. Copy assignment is deleted for the same reason, which is why the assignment from x to y won’t compile. (The move operations aren’t explicitly declared in std::atomic, so, per the rules for compiler-generated special functions described in Item 17, std::atomic offers neither move construction nor move assignment.)*” (std::atomic structure. Explanation of absence of move, copy and assignment operations)*

“It’s possible to get the value of x into y, but it requires use of std::atomic’s member functions load and store. The load member function reads a std::atomic’s value atomically, while the store member function writes it atomically. To initialize y with x, followed by putting x’s value in y, the code must be written like this:

std::atomic<int> y(x.load()); // read x

y.store(x.load()); // read x again

This compiles, but the fact that reading x (via x.load()) is a separate function call from initializing or storing to y makes clear that there is no reason to expect either statement as a whole to execute as a single atomic operation.” (*std::atomic structure*)

*“*Given that code, compilers could “optimize” it by storing x’s value in a register instead of reading it twice:

register = x.load(); // read x into register

std::atomic<int> y(register); // init y with register value

y.store(register); // store register value into y

The result, as you can see, reads from x only once, and that’s the kind of optimization that must be avoided when dealing with special memory. (The optimization isn’t permitted for volatile variables.)*” (alternative solution of copying of atomic values)*

*“*The situation should thus be clear:

• std::atomic is useful for concurrent programming, but not for accessing special memory.

• volatile is useful for accessing special memory, but not for concurrent programming.*” (using context of std::atomic and volatile)*

*p. 297*

*“*Because std::atomic and volatile serve different purposes, they can even be used together:

volatile std::atomic<int> vai; // operations on vai are

// atomic and can't be

// optimized away

This could be useful if vai corresponded to a memory-mapped I/O location that was concurrently accessed by multiple threads.*” Atomic volatile variables in C++ and their using context.*

*“*As a final note, some developers prefer to use std::atomic’s load and store member functions even when they’re not required, because it makes explicit in the source code that the variables involved aren’t “normal.” Emphasizing that fact isn’t unreasonable.*” (programming practice. Intbu in examples.)*

*“*Accessing a std::atomic is typically much slower than accessing a nonstd:: atomic, and we’ve already seen that the use of std::atomics prevents compilers from performing certain kinds of code reorderings that would otherwise be permitted.*” (std::atomic performance slowness)*

*“*Calling out loads and stores of std::atomics can therefore help identify potential scalability chokepoints. From a correctness perspective, not seeing a call to store on a variable meant to communicate information to other threads (e.g., a flag indicating the availability of data) could mean that the variable wasn’t declared std::atomic when it should have been. This is largely a style issue, however, and as such is quite different from the choice between std::atomic and volatile. *” (intbu. I didn’t understand)*

*“*Things to Remember

• std::atomic is for data accessed from multiple threads without using mutexes. It’s a tool for writing concurrent software.

• volatile is for memory where reads and writes should not be optimized away. It’s a tool for working with special memory.*” (book structure. resume)*

*p. 299*

# CHAPTER 8 Tweaks

For every general technique or feature in C++, there are circumstances where it’s reasonable to use it, and there are circumstances where it’s not.*” (C++ fundamental structure)*

*“*Describing when it makes sense to use a general technique or feature is usually fairly straightforward, but this chapter covers two exceptions. The general technique is pass by value, and the general feature is emplacement. The decision about when to employ them is affected by so many factors, the best advice I can offer is to consider their use.*” (kind of using context – complicated. Peculiarities of pass by value and emplacement C++ features.)*

*“*Nevertheless, both are important players in effective modern C++ programming, and the Items that follow provide the information you’ll need to determine whether using them is appropriate for your software.*” (book structure and importance of these two features)*

## Item 41: Consider pass by value for copyable parameters that are cheap to move and always copied.

*(book recommendation and pass by value using context)*

*“*In this Item, to “copy” a parameter generally means to use it as the source of a copy or move operation. Recall on page 2 that C++ has no terminology to distinguish a copy made by a copy operation from one made by a move operation.*” (book terminology)*

*“*Some function parameters are intended to be copied.1*” (usgin context of pass by value)*

*“*For example, a member function addName might copy its parameter into a private container. For efficiency, such a function should copy lvalue arguments, but move rvalue arguments:

class Widget {

public:

void addName(const std::string& newName) // take lvalue;

{ names.push\_back(newName); } // copy it

void addName(std::string&& newName) // take rvalue;

{ names.push\_back(std::move(newName)); } // move it; see

… // Item 25 for use

// of std::move

private:

std::vector<std::string> names;

};*” (example of this using context. Good C++ practice)*

*p. 300*

*“*This works, but it requires writing two functions that do essentially the same thing. That chafes a bit: two functions to declare, two functions to implement, two functions to document, two functions to maintain. Ugh.*” (disadvantage of overloading) “*Furthermore, there will be two functions in the object code—something you might care about if you’re concerned about your program’s footprint. In this case, both functions will probably be inlined, and that’s likely to eliminate any bloat issues related to the existence of two functions, but if these functions aren’t inlined everywhere, you really will get two functions in your object code.*” (intbu what is program footprint? What is inlined and what is object code. C++ structure.)*

An alternative…

*p. 300*

*23.01.2017*

*“*An alternative approach is to make addName a function template taking a universal

reference (see Item 24):

class Widget {

public:

template<typename T> // take lvalues

void addName(T&& newName) // and rvalues;

{ // copy lvalues,

names.push\_back(std::forward<T>(newName)); // move rvalues;

} // see Item 25

// for use of

… // std::forward

};

This reduces the source code you have to deal with, but the use of universal references leads to other complications. As a template, addName’s implementation must typically be in a header file. It may yield several functions in object code, because it not only instantiates differently for lvalues and rvalues, it also instantiates differently for std::string and types that are convertible to std::string (see Item 25). At the same time, there are argument types that can’t be passed by universal reference (see Item 30), and if clients pass improper argument types, compiler error messages can be intimidating (see Item 27).*” (another solution of the problem. Limitation of universal references use. Remind. Intbu problems with object code. What is object code?)*

*“*Wouldn’t it be nice if there were a way to write functions like addName such that lvalues were copied, rvalues were moved, there was only one function to deal with (in both source and object code), and the idiosyncrasies of universal references were avoided?*” (task formulation. What is isiosyncrasies? Of universal references?)*

*“*As it happens, there is. All you have to do is abandon one of the first rules you probably learned as a C++ programmer. That rule was to avoid passing objects of user-defined types by value. For parameters like newName in functions like addName, pass by value may be an entirely reasonable strategy.*” (passing by value using context) p. 301*

*“*Before we discuss why pass-by-value may be a good fit for newName and addName, let’s see how it would be implemented:

class Widget {

public:

void addName(std::string newName) // take lvalue or

{ names.push\_back(std::move(newName)); } // rvalue; move it

…

};

The only non-obvious part of this code is the application of std::move to the parameter newName. Typically, std::move is used with rvalue references, but in this case, we know that (1) newName is a completely independent object from whatever the caller passed in, so changing newName won’t affect callers and (2) this is the final use of newName, so moving from it won’t have any impact on the rest of the function.*” (example and explanation of this technique. 2 conditions of passing by value: no influence on the caller and final usage of the object)*

*“*The fact that there’s only one addName function explains how we avoid code duplication, both in the source code and the object code. We’re not using a universal reference, so this approach doesn’t lead to bloated header files, odd failure cases, or confounding error messages.*” (intbu this avoidance and what is about object code? What is “odd failure cases”?)*

*“*But what about the efficiency of this design? We’re passing by value. Isn’t that expensive?*” (book question).*

*“*In C++98, it was a reasonable bet that it was. No matter what callers passed in, the parameter newName would be created by copy construction. In C++11, however, add Name will be copy constructed only for lvalues. For rvalues, it will be move constructed.*” (answer the question. C++ history. And C++ structure – pass by value realization)*

*“*Here, look:

Widget w;

…

std::string name("Bart");

w.addName(name); // call addName with lvalue

…

w.addName(name + "Jenne"); // call addName with rvalue

// (see below)*” (example of it)*

*p. 302*

*“*In the first call to addName (when name is passed), the parameter newName is initialized with an lvalue. newName is thus copy constructed, just like it would be in C++98. In the second call, newName is initialized with the std::string object resulting from a call to operator+ for std::string (i.e., the append operation). That object is an rvalue, and newName is therefore move constructed. Lvalues are thus copied, and rvalues are moved, just like we want. Neat, huh? *” (C++ structure: examples of rvalue and lvalue).*

*Nuances of pass by value: “*It is neat, but there are some caveats you need to keep in mind. Doing that will be easier if we recap the three versions of addName we’ve considered:

class Widget { // Approach 1:

public: // overload for

void addName(const std::string& newName) // lvalues and

{ names.push\_back(newName); } // rvalues

void addName(std::string&& newName)

{ names.push\_back(std::move(newName)); }

…

private:

std::vector<std::string> names;

};

class Widget { // Approach 2:

public: // use universal

template<typename T> // reference

void addName(T&& newName)

{ names.push\_back(std::forward<T>(newName)); }

…

};

class Widget { // Approach 3:

public: // pass by value

void addName(std::string newName)

{ names.push\_back(std::move(newName)); }

…

};

I refer to the first two versions as the “by-reference approaches,” because they’re both based on passing their parameters by reference.

284 | Item 41

[www.it-ebooks.info](http://www.it-ebooks.info)*” (book structure. Three cases of references)*

*p. 303*

*“*Here are the two calling scenarios we’ve examined:

Widget w;

…

std::string name("Bart");

w.addName(name); // pass lvalue

…

w.addName(name + "Jenne"); // pass rvalue*” (book structure. Calling scenario – important part of code analysis)*

*“*Now consider the cost, in terms of copy and move operations, of adding a name to a Widget for the two calling scenarios and each of the three addName implementations we’ve discussed. The accounting will largely ignore the possibility of compilers optimizing copy and move operations away, because such optimizations are context- and compiler-dependent and, in practice, don’t change the essence of the analysis.*” (book structure. Sub-topic of this topic)*

*“*Overloading: Regardless of whether an lvalue or an rvalue is passed, the caller’s argument is bound to a reference called newName. That costs nothing, in terms of copy and move operations. In the lvalue overload, newName is copied into Widget::names. In the rvalue overload, it’s moved. Cost summary: one copy for lvalues, one move for rvalues.*” (overloading approach analysis)*

*“*Using a universal reference: As with overloading, the caller’s argument is bound to the reference newName. This is a no-cost operation. Due to the use of std::forward, lvalue std::string arguments are copied into Widget::names, while rvalue std::string arguments are moved. The cost summary for std::string arguments is the same as with overloading: one copy for lvalues, one move for rvalues.*” (universal references approach structure)*

*“*Item 25 explains that if a caller passes an argument of a type other than std::string, it will be forwarded to a std::string constructor, and that could cause as few as zero std::string copy or move operations to be performed. Functions taking universal references can thus be uniquely efficient. However, that doesn’t affect the analysis in this Item, so we’ll keep things simple by assuming that callers always pass std::string arguments.*” (remind this. What is uniquely efficient?)*

*“*Passing by value: Regardless of whether an lvalue or an rvalue is passed, the parameter newName must be constructed. If an lvalue is passed, this costs a copy construction. If an rvalue is passed, it costs a move construction. In the body of the function, newName is unconditionally moved into Widget::names. The cost summary is thus one copy plus one move for lvalues, and two moves for rvalues. Compared to the by-reference approaches, that’s one extra move for both lvalues and rvalues.*” (passing by value structure and peculiarities)*

*p. 304*

*“*Look again at this Item’s title: Consider pass by value for copyable parameters that are cheap to move and always copied. It’s worded the way it is for a reason. Four reasons, in fact:*” (using context of pass by value approach)*

*“*You should only consider using pass by value. Yes, it requires writing only one function. Yes, it generates only one function in the object code. Yes, it avoids the issues associated with universal references. But it has a higher cost than the alternatives, and, as we’ll see below, in some cases, there are expenses we haven’t yet discussed.*” (using context structure of pass by value approach)*

*“*Consider pass by value only for copyable parameters. Parameters failing this test must have move-only types, because if they’re not copyable, yet the function always makes a copy, the copy must be created via the move constructor.2

Recall that the advantage of pass by value over overloading is that with pass by value, only one function has to be written. But for move-only types, there is no need to provide an overload for lvalue arguments, because copying an lvalue entails calling the copy constructor, and the copy constructor for move-only types is disabled. That means that only rvalue arguments need to be supported, and in that case, the “overloading” solution requires only one overload: the one taking an rvalue reference.*” (using context of pass by value and pass by reference approachs)*

*“*Consider a class with a std::unique\_ptr<std::string> data member and a setter for it. std::unique\_ptr is a move-only type, so the “overloading” approach to its setter consists of a single function:

class Widget {

public:

…

void setPtr(std::unique\_ptr<std::string>&& ptr)

{ p = std::move(ptr); }

private:

std::unique\_ptr<std::string> p;

};

A caller might use it this way:

Widget w;

…

w.setPtr(std::make\_unique<std::string>("Modern C++")); *” (example of last using context).*

*“” (C++ structure. Two kinds of copy operations and terminology development – widening because of differentiation).*

*p. 305*

*“*Here the rvalue std::unique\_ptr<std::string> returned from std::make\_unique (see Item 21) is passed by rvalue reference to setPtr, where it’s moved into the data member p. The total cost is one move.*” (example explanation)*

*“*If setPtr were to take its parameter by value,

class Widget {

public:

…

void setPtr(std::unique\_ptr<std::string> ptr)

{ p = std::move(ptr); }

…

};

the same call would move construct the parameter ptr, and ptr would then be move assigned into the data member p. The total cost would thus be two moves —twice that of the “overloading” approach.*” (example continue)*

*“*Pass by value is worth considering only for parameters that are cheap to move. When moves are cheap, the cost of an extra one may be acceptable, but when they’re not, performing an unnecessary move is analogous to performing an unnecessary copy, and the importance of avoiding unnecessary copy operations is what led to the C++98 rule about avoiding pass by value in the first place!*” (using context of pass by value)*

*“*You should consider pass by value only for parameters that are always copied. To see why this is important, suppose that before copying its parameter into the names container, addName checks to see if the new name is too short or too long. If it is, the request to add the name is ignored. A pass-by-value implementation could be written like this:

class Widget {

public:

void addName(std::string newName)

{

if ((newName.length() >= minLen) &&

(newName.length() <= maxLen))

{

names.push\_back(std::move(newName));

}

}

…

private:

std::vector<std::string> names;

};

This function incurs the cost of constructing and destroying newName, even if nothing is added to names. That’s a price the by-reference approaches wouldn’t be asked to pay.*” (using context of pass by value)*

*p. 306*

*“*Even when you’re dealing with a function performing an unconditional copy on a copyable type that’s cheap to move, there are times when pass by value may not be appropriate. That’s because a function can copy a parameter in two ways: via construction (i.e., copy construction or move construction) and via assignment (i.e., copy assignment or move assignment). addName uses construction: its parameter newName is passed to vector::push\_back, and inside that function, newName is copy con- structed into a new element created at the end of the std::vector. For functions that use construction to copy their parameter, the analysis we saw earlier is complete: using pass by value incurs the cost of an extra move for both lvalue and rvalue arguments.*” (using context of pass by value. Is this case of chain of functions calls?)*

*“*When a parameter is copied using assignment, the situation is more complicated.*” (using context structure of pass by value approach)*

*“*Suppose, for example, we have a class representing passwords. Because passwords can be changed, we provide a setter function, changeTo. Using a pass-by-value strategy, we could implement Password like this:

class Password {

public:

explicit Password(std::string pwd) // pass by value

: text(std::move(pwd)) {} // construct text

void changeTo(std::string newPwd) // pass by value

{ text = std::move(newPwd); } // assign text

…

private:

std::string text; // text of password

};

Storing the password as plain text will whip your software security SWAT team into a frenzy, but ignore that and consider this code:

std::string initPwd("Supercalifragilisticexpialidocious");

Password p(initPwd);*” (example of this. Intbu this aspect of computer security)*

*p. 307*

*“*There are no suprises here: p.text is constructed with the given password, and using pass by value in the constructor incurs the cost of a std::string move construction that would not be necessary if overloading or perfect forwarding were employed. All is well. A user of this program may not be as sanguine about the password, however, because “Supercalifragilisticexpialidocious” is found in many dictionaries. He or she may therefore take actions that lead to code equivalent to the following being executed:

std::string newPassword = "Beware the Jabberwock";

p.changeTo(newPassword);

Whether the new password is better than the old one is debatable, but that’s the user’s problem. Ours is that changeTo’s use of assignment to copy the parameter newPwd probably causes that function’s pass-by-value strategy to explode in cost. The argument passed to changeTo is an lvalue (newPassword), so when the parameter newPwd is constructed, it’s the std::string copy constructor that’s called. That constructor allocates memory to hold the new password. newPwd is then moveassigned to text, which causes the memory already held by text to be deallocated. There are thus two dynamic memory management actions within changeTo: one to allocate memory for the new password, and one to deallocate the memory for the old password.*” (example continue)*

*“*But in this case, the old password (“Supercalifragilisticexpialidocious”) is longer than the new one (“Beware the Jabberwock”), so there’s no need to allocate or deallocate anything. If the overloading approach were used, it’s likely that none would take place:

class Password {

public:

…

void changeTo(const std::string& newPwd) // the overload

{ // for lvalues

text = newPwd; // can reuse text's memory if

// text.capacity() >= newPwd.size()

}

…

private:

std::string text; // as above

};*” (example continue. variant with overloading approach. intbu why there is no need “to allocate or deallocate anything?)*

*p. 308*

*“*In this scenario, the cost of pass by value includes an extra memory allocation and deallocation—costs that are likely to exceed that of a std::string move operation by orders of magnitude. *” (example continue.)*

*“*Interestingly, if the old password were shorter than the new one, it would typically be impossible to avoid an allocation-deallocation pair during the assignment, and in that case, pass by value would run at about the same speed as pass by reference. The cost of assignment-based parameter copying can thus depend on the values of the objects participating in the assignment! This kind of analysis applies to any parameter type that holds values in dynamically allocated memory. Not all types qualify, but many—including std::string and std::vector—do.*” (using context structure of pass by value. Very-very interesting and complicated case.)*

*“*This potential cost increase generally applies only when lvalue arguments are passed, because the need to perform memory allocation and deallocation typically occurs only when true copy operations (i.e., not moves) are performed. For rvalue arguments, moves almost always suffice.*” (С++ memory management structure. Rvalues and lvalues.)*

*“*The upshot is that the extra cost of pass by value for functions that copy a parameter using assignment depends on the type being passed, the ratio of lvalue to rvalue arguments, whether the type uses dynamically allocated memory, and, if so, the implementation of that type’s assignment operators and the likelihood that the memory associated with the assignment target is at least as large as the memory associated with the assignment source. For std::string, it also depends on whether the implementation uses the small string optimization (SSO—see Item 29) and, if so, whether the values being assigned fit in the SSO buffer.*” (using context structure of pass by value. Very-very interesting and complicated case.)*

*“*So, as I said, when parameters are copied via assignment, analyzing the cost of pass by value is complicated. Usually, the most practical approach is to adopt a “guilty until proven innocent” policy, whereby you use overloading or universal references instead of pass by value unless it’s been demonstrated that pass by value yields acceptably efficient code for the parameter type you need.*” (book recommendation of this complicated case. Type of making decisions. Aspects of tkir.)*

*“*Now, for software that must be as fast as possible, pass by value may not be a viable strategy, because avoiding even cheap moves can be important. Moreover, it’s not always clear how many moves will take place. In the Widget::addName example, pass by value incurs only a single extra move operation, but suppose that Widget::add Name called Widget::validateName, and this function also passed by value. (Presumably it has a reason for always copying its parameter, e.g., to store it in a data structure of all values it validates.) And suppose that validateName called a third function that also passed by value…*” (pass by value using context structure. Intbu this book recommendation: “*Presumably it has a reason for always copying its parameter, e.g., to store it in a data structure of all values it validates.*”)*

*p. 309*

You can…

*24.01.2017*

*“*You can see where this is headed. When there are chains of function calls, each of which employs pass by value because “it costs only one inexpensive move,” the cost for the entire chain of calls may not be something you can tolerate. Using byreference parameter passing, chains of calls don’t incur this kind of accumulated overhead.*” (using context of pass by value. Chain of calls is a programming concept).*

*“*An issue unrelated to performance, but still worth keeping in mind, is that pass by value, unlike pass by reference, is susceptible to the slicing problem.*” (pass by value drawback and pass by reference advantage. What is slicing problem?)*

*“*This is well-trod C++98 ground, so I won’t dwell on it, but if you have a function that is designed to accept a parameter of a base class type or any type derived from it, you don’t want to declare a pass-by-value parameter of that type, because you’ll “slice off” the derivedclass characteristics of any derived type object that may be passed in:*” (the slicing problem. Inheritance aspects.)*

*“*class Widget { … }; // base class

class SpecialWidget: public Widget { … }; // derived class

void processWidget(Widget w); // func for any kind of Widget,

// including derived types;

… // suffers from slicing problem

SpecialWidget sw;

…

processWidget(sw); // processWidget sees a

// Widget, not a SpecialWidget!*” (example of it.)*

*“*If you’re not familiar with the slicing problem, search engines and the Internet are your friends; there’s lots of information available. You’ll find that the existence of the slicing problem is another reason (on top of the efficiency hit) why pass by value has a shady reputation in C++98. There are good reasons why one of the first things you probably learned about C++ programming was to avoid passing objects of userdefined types by value.*” (pass by value using context limitations.)*

*“*C++11 doesn’t fundamentally change the C++98 wisdom regarding pass by value. In general, pass by value still entails a performance hit you’d prefer to avoid, and pass by value can still lead to the slicing problem. What’s new in C++11 is the distinction between lvalue and rvalue arguments. Implementing functions that take advantage of move semantics for rvalues of copyable types requires either overloading or using universal references, both of which have drawbacks. For the special case of copyable, cheap-to-move types passed to functions that always copy them and where slicing is not a concern, pass by value can offer an easy-to-implement alternative that’s nearly as efficient as its pass-by-reference competitors, but avoids their disadvantages.*” (С++ history and using context of pass by value and pass by reference)*

*p. 310*

*book structure: resume: “*Things to Remember

• For copyable, cheap-to-move parameters that are always copied, pass by value may be nearly as efficient as pass by reference, it’s easier to implement, and it can generate less object code.

• Copying parameters via construction may be significantly more expensive than copying them via assignment.

• Pass by value is subject to the slicing problem, so it’s typically inappropriate for base class parameter types.*” (intbu difference between copying via construction and via assignment)*

## Item 42: Consider emplacement instead of insertion

*“*If you have a container holding, say, std::strings, it seems logical that when you add a new element via an insertion function (i.e., insert, push\_front, push\_back, or, for std::forward\_list, insert\_after), the type of element you’ll pass to the function will be std::string. After all, that’s what the container has in it.*” (expectation of programmer after insertion value to container)*

*“*Logical though this may be, it’s not always true. Consider this code:

std::vector<std::string> vs; // container of std::string

vs.push\_back("xyzzy"); // add string literal

Here, the container holds std::strings, but what you have in hand—what you’re actually trying to push\_back—is a string literal, i.e., a sequence of characters inside quotes.*” (definition of the literal. Example demonstating that these expectation are not always true)*

*“*A string literal is not a std::string, and that means that the argument you’re passing to push\_back is not of the type held by the container. push\_back for std::vector is overloaded for lvalues and rvalues as follows:

template <class T, // from the C++11

class Allocator = allocator<T>> // Standard

class vector {

public:

…

void push\_back(const T& x); // insert lvalue

void push\_back(T&& x); // insert rvalue

…

};

In the call

vs.push\_back("xyzzy");*” (example structure)*

*p. 311*

*“*compilers see a mismatch between the type of the argument (const char[6]) and the type of the parameter taken by push\_back (a reference to a std::string). They address the mismatch by generating code to create a temporary std::string object from the string literal, and they pass that temporary object to push\_back. In other words, they treat the call as if it had been written like this:

vs.push\_back(std::string("xyzzy")); // create temp. std::string

// and pass it to push\_back

The code compiles and runs, and everybody goes home happy. Everybody except the performance freaks, that is, because the performance freaks recognize that this code isn’t as efficient as it should be.*” (example continueю compiler work. Performance depends on object code as far as I understood)*

“To create a new element in a container of std::strings, they understand, a std::string constructor is going to have to be called, but the code above doesn’t make just one constructor call. It makes two. And it calls the std::string destructor, too. Here’s what happens at runtime in the call to push\_back:

1. A temporary std::string object is created from the string literal "xyzzy". This object has no name; we’ll call it temp. Construction of temp is the first std::string construction. Because it’s a temporary object, temp is an rvalue.

2. temp is passed to the rvalue overload for push\_back, where it’s bound to the rvalue reference parameter x. A copy of x is then constructed in the memory for the std::vector. This construction—the second one—is what actually creates a new object inside the std::vector. (The constructor that’s used to copy x into the std::vector is the move constructor, because x, being an rvalue reference, gets cast to an rvalue before it’s copied. For information about the casting of rvalue reference parameters to rvalues, see Item 25.)

3. Immediately after push\_back returns, temp is destroyed, thus calling the std::string destructor.” (*example explanation. Intbu information about move cast and move constructor. Move constructor for construction of object in std::vector. It is connected with move realization. Objetc must by saved inside std::vector. It is not operations with only references or pointers. Intbu this*).

*“*The performance freaks can’t help but notice that if there were a way to take the string literal and pass it directly to the code in step 2 that constructs the std::string object inside the std::vector, we could avoid constructing and destroying temp. That would be maximally efficient, and even the performance freaks could contentedly decamp. *” (interesting language of Mayers and possible solution.)*

*“*Because you’re a C++ programmer, there’s an above-average chance you’re a performance freak. If you’re not, you’re still probably sympathetic to their point of view. (If you’re not at all interested in performance, shouldn’t you be in the Python room down the hall?)*” (difference between Python and C++)*

*p. 312*

*“*So I’m pleased to tell you that there is a way to do exactly what is needed for maximal efficiency in the call to push\_back. It’s to not call push\_back. push\_back is the wrong function. The function you want is emplace\_back. emplace\_back does exactly what we desire: it uses whatever arguments are passed to it to construct a std::string directly inside the std::vector. No temporaries are involved:

vs.emplace\_back("xyzzy"); // construct std::string inside

// vs directly from "xyzzy"*” (emplace\_back purpose and structure)*

*“*emplace\_back uses perfect forwarding, so, as long as you don’t bump into one of perfect forwarding’s limitations (see Item 30), you can pass any number of arguments of any combination of types through emplace\_back. For example, if you’d like to create a std::string in vs via the std::string constructor taking a character and a repeat count, this would do it:

vs.emplace\_back(50, 'x'); // insert std::string consisting

// of 50 'x' characters

emplace\_back is available for every standard container that supports push\_back.*” (emplace\_back structure. Perfect forwarding aspects. Limitations of emplace\_back. Emplace\_back existing context and STL containers structure.)*

*“*Similarly, every standard container that supports push\_front supports emplace\_front. And every standard container that supports insert (which is all but std::forward\_list and std::array) supports emplace. The associative containers offer emplace\_hint to complement their insert functions that take a “hint” iterator, and std::forward\_list has emplace\_after to match its insert\_after.*” (kinds of emplace operations. What is “hint” iterator?)*

*“*What makes it possible for emplacement functions to outperform insertion functions is their more flexible interface. Insertion functions take objects to be inserted, while emplacement functions take constructor arguments for objects to be inserted. This difference permits emplacement functions to avoid the creation and destruction of temporary objects that insertion functions can necessitate.*” (semantic and realization differences between insert and emplace operations)*

*“*Because an argument of the type held by the container can be passed to an emplacement function (the argument thus causes the function to perform copy or move construction), emplacement can be used even when an insertion function would require no temporary. In that case, insertion and emplacement do essentially the same thing.*” (situation where emplacement and insertion are identical)*

*“*For example, given

std::string queenOfDisco("Donna Summer");

both of the following calls are valid, and both have the same net effect on the container:

vs.push\_back(queenOfDisco); // copy-construct queenOfDisco

// at end of vs

vs.emplace\_back(queenOfDisco); // ditto*” (example of this situation)*

*p. 313*

*“*Emplacement functions can thus do everything insertion functions can. They sometimes do it more efficiently, and, at least in theory, they should never do it less efficiently. So why not use them all the time?*” (book question)*

*“*Because, as the saying goes, in theory, there’s no difference between theory and practice, but in practice, there is. With current implementations of the Standard Library, there are situations where, as expected, emplacement outperforms insertion, but, sadly, there are also situations where the insertion functions run faster.*” (relations between effectiveness of insertion and emplacement. It is context dependent.)*

*“*Such situations are not easy to characterize, because they depend on the types of arguments

being passed, the containers being used, the locations in the containers where insertion or emplacement is requested, the exception safety of the contained types’ constructors, and, for containers where duplicate values are prohibited (i.e., std::set, std::map, std::unordered\_set, std::unordered\_map), whether the value to be added is already in the container.*” (complexity of this using contexts relation. 5 influentional factors.)*

*“*The usual performance-tuning advice thus applies: to determine whether emplacement or insertion runs faster, benchmark them both.*” (book recommendation. Common way how to make better choice where realization is cheap and both variants are possible)*

*“*That’s not very satisfying, of course, so you’ll be pleased to learn that there’s a heuristic that can help you identify situations where emplacement functions are most likely to be worthwhile.*” (using context of heuristic)*

*“*If all the following are true, emplacement will almost certainly outperform insertion:*” (heuristic of emplacement superiority over insertion)*

*“*The value being added is constructed into the container, not assigned.*” (the first heuristic condition)*

*“*The example that opened this Item (adding a std::string with the value "xyzzy" to a std::vector vs) showed the value being added to the end of vs—to a place where no object yet existed. The new value therefore had to be constructed into the std::vector. If we revise the example such that the new std::string goes into a location already occupied by an object, it’s a different story. Consider:

std::vector<std::string> vs; // as before

… // add elements to vs

vs.emplace(vs.begin(), "xyzzy"); // add "xyzzy" to

// beginning of vs

For this code, few implementations will construct the added std::string into the memory occupied by vs[0]. Instead, they’ll move-assign the value into place. But move assignment requires an object to move from, and that means that a temporary object will need to be created to be the source of the move. Because the primary advantage of emplacement over insertion is that temporary objects are neither created nor destroyed, when the value being added is put into the container via assignment, emplacement’s edge tends to disappear.*” (example of opposite situation where temporary object will be constructed in the case of emplacement too)*

*“*Alas, whether adding a value to a container is accomplished by construction or assignment is generally up to the implementer. But, again, heuristics can help.*” (dependence of emplacement superiority on compiler implementation. Using context structure of emplacement and insertion).*

*p. 314*

*“*Node-based containers virtually always use construction to add new values, and most standard containers are node-based. The only ones that aren’t are std::vector, std::deque, and std::string. (std::array isn’t, either, but it doesn’t support insertion or emplacement, so it’s not relevant here.) Within the non-node-based containers, you can rely on emplace\_back to use construction instead of assignment to get a new value into place, and for std::deque, the same is true of emplace\_front.*” (using context of emplacement. What are node-based containers and non node-based containers? Intbu. Intbu the last sentence. Construction is good or not?)*

*“*The argument type(s) being passed differ from the type held by the container.*” (another heuristic condition)*

*“*Again, emplacement’s advantage over insertion generally stems from the fact that its interface doesn’t require creation and destruction of a temporary object when the argument(s) passed are of a type other than that held by the container. When an object of type T is to be added to a container<T>, there’s no reason to expect emplacement to run faster than insertion, because no temporary needs to be created to satisfy the insertion interface.*” (second heuristic condition. Insertion and emplacement using context).*

*“*The container is unlikely to reject the new value as a duplicate.*” (third heuristic condition)*

*“*This means that the container either permits duplicates or that most of the values you add will be unique. The reason this matters is that in order to detect whether a value is already in the container, emplacement implementations typically create a node with the new value so that they can compare the value of this node with existing container nodes. If the value to be added isn’t in the container, the node is linked in. However, if the value is already present, the emplacement is aborted and the node is destroyed, meaning that the cost of its construction and destruction was wasted. Such nodes are created for emplacement functions more often than for insertion functions.*” (its explanation. Intbu about frequency of node creation in the case of emplacement and insertion. Why are they different?)*

*“*The following calls from earlier in this Item satisfy all the criteria above. They also run faster than the corresponding calls to push\_back.

vs.emplace\_back("xyzzy"); // construct new value at end of

// container; don't pass the type in

// container; don't use container

// rejecting duplicates

vs.emplace\_back(50, 'x'); // ditto*” (example of heuristic satisfaction)*

*“*When deciding whether to use emplacement functions, two other issues are worth keeping in mind. The first regards resource management.*” (another influential issue. Using context of emplacement. Resource management aspects)*

*“*Suppose you have a container of std::shared\_ptr<Widget>s,

std::list<std::shared\_ptr<Widget>> ptrs;

and you want to add a std::shared\_ptr that should be released via a custom deleter (see Item 19). Item 21 explains that you should use std::make\_shared to create std::shared\_ptrs whenever you can, but it also concedes that there are situations where you can’t. One such situation is when you want to specify a custom deleter. In that case, you must use new directly to get the raw pointer to be managed by the std::shared\_ptr.*” (explanation of this case with example. Example is connected with shared\_pointers).*

*p. 315*

*“*If the custom deleter is this function,

void killWidget(Widget\* pWidget);

the code using an insertion function could look like this:

ptrs.push\_back(std::shared\_ptr<Widget>(new Widget, killWidget));

It could also look like this, though the meaning would be the same:

ptrs.push\_back({ new Widget, killWidget });

Either way, a temporary std::shared\_ptr would be constructed before calling push\_back. push\_back’s parameter is a reference to a std::shared\_ptr, so there has to be a std::shared\_ptr for this parameter to refer to. The creation of the temporary std::shared\_ptr is what emplace\_back would avoid, but in this case, that temporary is worth far more than it costs.*” (example continue. insertion avoids memory leaks.)*

*“*Consider the following potential sequence of events: 1. In either call above, a temporary std::shared\_ptr<Widget> object is constructed to hold the raw pointer resulting from “new Widget”. Call this object temp.

2. push\_back takes temp by reference. During allocation of a list node to hold a copy of temp, an out-of-memory exception gets thrown.

3. As the exception propagates out of push\_back, temp is destroyed. Being the sole std::shared\_ptr referring to the Widget it’s managing, it automatically releases that Widget, in this case by calling killWidget.

Even though an exception occurred, nothing leaks: the Widget created via “new Widget” in the call to push\_back is released in the destructor of the std::shared\_ptr that was created to manage it (temp). Life is good. Now consider what happens if emplace\_back is called instead of push\_back:

ptrs.emplace\_back(new Widget, killWidget);

1. The raw pointer resulting from “new Widget” is perfect-forwarded to the point inside emplace\_back where a list node is to be allocated. That allocation fails, and an out-of-memory exception is thrown.*” (example continue. Situation with emplace back.)*

*p. 316*

*“*2. As the exception propagates out of emplace\_back, the raw pointer that was the only way to get at the Widget on the heap is lost. That Widget (and any resources it owns) is leaked. In this scenario, life is not good, and the fault doesn’t lie with std::shared\_ptr.*” (example continue. intbu why exception is propogates out of emplacement. Because of there is no temporary object inside emplacent because of moving?)*

*“*The same kind of problem can arise through the use of std::unique\_ptr with a custom deleter.*” (intbu this analogy)*

*“*Fundamentally, the effectiveness of resource-managing classes like std::shared\_ptr and std::unique\_ptr is predicated on resources (such as raw pointers from new) being immediately passed to constructors for resource-managing objects. The fact that functions like std::make\_shared and std::make\_unique automate this is one of the reasons they’re so important.*” (intbu this advantage of make functions. What do they automate?)*

*“*In calls to the insertion functions of containers holding resource-managing objects (e.g., std::list<std::shared\_ptr<Widget>>), the functions’ parameter types generally ensure that nothing gets between acquisition of a resource (e.g., use of new) and construction of the object managing the resource.*” (intbu translation of this sentence)*

*“*In the emplacement functions, perfect-forwarding defers the creation of the resource-managing objects until they can be constructed in the container’s memory, and that opens a window during which exceptions can lead to resource leaks. All standard containers are susceptible to this problem. When working with containers of resource-managing objects, you must take care to ensure that if you choose an emplacement function over its insertion counterpart, you’re not paying for improved code efficiency with diminished exception safety.*” (for explanation of this resource management problem. Using context of emplacement. Aspects of exception safety).*

*“*Frankly, you shouldn’t be passing expressions like “new Widget” to emplace\_back or push\_back or most any other function, anyway, because, as Item 21 explains, this leads to the possibility of exception safety problems of the kind we just examined. Closing the door requires taking the pointer from “new Widget” and turning it over to a resource-managing object in a standalone statement, then passing that object as an rvalue to the function you originally wanted to pass “new Widget” to. (Item 21 covers this technique in more detail.) The code using push\_back should therefore be written more like this:

std::shared\_ptr<Widget> spw(new Widget, // create Widget and

killWidget); // have spw manage it

ptrs.push\_back(std::move(spw)); // add spw as rvalue

The emplace\_back version is similar:

std::shared\_ptr<Widget> spw(new Widget, killWidget);

ptrs.emplace\_back(std::move(spw));*” (solution of this exception safety problem).*

*“*Either way, the approach incurs the cost of creating and destroying spw. Given that the motivation for choosing emplacement over insertion is to avoid the cost of a temporary object of the type held by the container, yet that’s conceptually what spw is, emplacement functions are unlikely to outperform insertion functions when you’re adding resource-managing objects to a container and you follow the proper practice of ensuring that nothing can intervene between acquiring a resource and turning it over to a resource-managing object.*” (intbu this resource-management rule better. As far as I understood additional construction can often help to avoid some safety problems)*

*p. 317*

*“*A second noteworthy aspect of emplacement functions is their interaction with explicit constructors.*” (the second aspect of emplacement using context. Intbu what is explicit keyword)*

*“*In honor of C++11’s support for regular expressions,*” (C++ structure and history. Std::regex)*

*“*suppose you create a container of regular expression objects:

std::vector<std::regex> regexes;

Distracted by your colleagues’ quarreling over the ideal number of times per day to check one’s Facebook account, you accidentally write the following seemingly meaningless code:

regexes.emplace\_back(nullptr); // add nullptr to container

// of regexes?

You don’t notice the error as you type it, and your compilers accept the code without complaint, so you end up wasting a bunch of time debugging. At some point, you discover that you have inserted a null pointer into your container of regular expressions. But how is that possible? Pointers aren’t regular expressions, and if you tried to do something like this,

std::regex r = nullptr; // error! won't compile

compilers would reject your code. Interestingly, they would also reject it if you called push\_back instead of

emplace\_back:

regexes.push\_back(nullptr); // error! won't compile*” (demonstration of problem on the example)*

*“*The curious behavior you’re experiencing stems from the fact that std::regex objects can be constructed from character strings. That’s what makes useful code like this legal:

std::regex upperCaseWord("[A-Z]+");

Creation of a std::regex from a character string can exact a comparatively large runtime cost, so, to minimize the likelihood that such an expense will be incurred unintentionally, the std::regex constructor taking a const char\* pointer is explicit.*” (std::regex structure. Intbu what is explicit. What is its purpose?)*

[*http://en.cppreference.com/w/cpp/language/explicit, http://hitfounder.livejournal.com/20152.html*](http://en.cppreference.com/w/cpp/language/explicit,%20http://hitfounder.livejournal.com/20152.html) *(about explicit. Intbu why it is important to forbid implicit constructor conversion?)*

*“*That’s why these lines don’t compile:

std::regex r = nullptr; // error! won't compile

regexes.push\_back(nullptr); // error! won't compile

In both cases, we’re requesting an implicit conversion from a pointer to a std::regex, and the explicitness of that constructor prevents such conversions.*” (example explanation)*

*p. 318*

*“*In the call to emplace\_back, however, we’re not claiming to pass a std::regex object. Instead, we’re passing a constructor argument for a std::regex object. That’s not considered an implicit conversion request. Rather, it’s viewed as if you’d written this code:

std::regex r(nullptr); // compiles

If the laconic comment “compiles” suggests a lack of enthusiasm, that’s good, because this code, though it will compile, has undefined behavior. The std::regex constructor taking a const char\* pointer requires that the pointed-to string comprise a valid regular expression, and the null pointer fails that requirement. If you write and compile such code, the best you can hope for is that it crashes at runtime. If you’re not so lucky, you and your debugger could be in for a special bonding experience.*” (example explanation. Emplacement structure. Undefined behavior case)*

*“*Setting aside push\_back, emplace\_back, and bonding for a moment, notice how these very similar initialization syntaxes yield different results:

std::regex r1 = nullptr; // error! won't compile

std::regex r2(nullptr); // compiles

In the official terminology of the Standard, the syntax used to initialize r1 (employing the equals sign) corresponds to what is known as copy initialization. In contrast, the syntax used to initialize r2 (with the parentheses, although braces may be used instead) yields what is called direct initialization.*” (direct initialization and copy initialization definition)*

*“*Copy initialization is not permitted to use explicit constructors. Direct initialization is. That’s why the line initializing r1 doesn’t compile, but the line initializing r2 does.*” (explicit structure. Example explanation).*

*“*But back to push\_back and emplace\_back and, more generally, the insertion functions versus the emplacement functions. Emplacement functions use direct initialization, which means they may use explicit constructors. Insertion functions employ copy initialization, so they can’t. Hence:

regexes.emplace\_back(nullptr); // compiles. Direct init permits

// use of explicit std::regex

// ctor taking a pointer

regexes.push\_back(nullptr); // error! copy init forbids

// use of that ctor*” (differences in realizations of emplacement and insertion)*

*“*The lesson to take away is that when you use an emplacement function, be especially careful to make sure you’re passing the correct arguments, because even explicit constructors will be considered by compilers as they try to find a way to interpret your code as valid.*” (purpose of explicit – to control constructor arguments types. Book recommendation about using of emplacement).*

*“*Things to Remember

• In principle, emplacement functions should sometimes be more efficient than their insertion counterparts, and they should never be less efficient.

• In practice, they’re most likely to be faster when (1) the value being added is constructed into the container, not assigned; (2) the argument type(s) passed differ from the type held by the container; and (3) the container won’t reject the value being added due to it being a duplicate.

• Emplacement functions may perform type conversions that would be rejected by insertion functions. *” (book structure. Resume)*