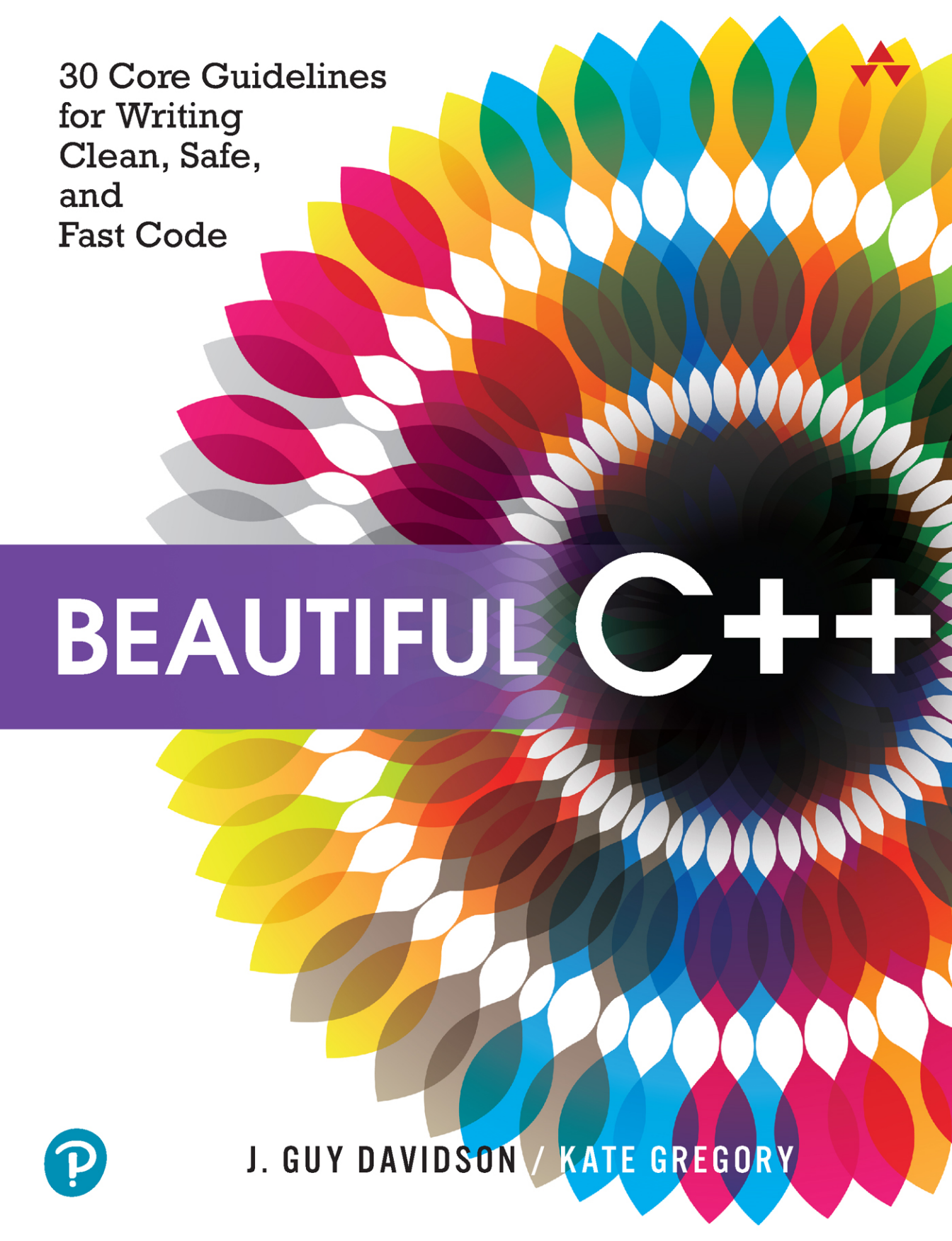
30 Core Guidelines
for Writing
Clean, Safe,
and
Fast Code
BEAUTIFUL C++
J. GUY DAVIDSON / KATE GREGORY

Beautiful C++

This page intentionally left blank

Beautiful C++

30 Core Guidelines for Writing Clean, Safe, and Fast Code

J. Guy Davidson
Kate Gregory

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw.

Library of Congress Control Number: 2021947544

Cover image: IROOM STOCK/Shutterstock

ISBN-13: 978-0-13-764784-2
ISBN-10: 0-13-764784-0

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at
  https://www.pearson.com/report-bias.html.

This page intentionally left blank

To Bryn

To Sinead

To Rory and Lois

C.47: Much love, JGD

To Jim Allison, though he is unlikely to see it. Research works. And to Chloe and Aisha who have not been at the front of books before, KMG

This page intentionally left blank

Contents

This page intentionally left blank

Selected C++ Core Guidelines

P.2: Write in ISO Standard C++ (*Chapter 1.1*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-Cplusplus

P.4: Ideally, a program should be statically type safe (*Chapter 5.1*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-typesafe

P.10: Prefer immutable data to mutable data (*Chapter 5.2*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-mutable

P.11: Encapsulate messy constructs, rather than spreading through the code (*Chapter 2.1*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-library

I.3: Avoid singletons (*Chapter 3.2*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-singleton

I.11: Never transfer ownership by a raw pointer (T*) or reference (T&) (*Chapter 3.1*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-raw

I.23: Keep the number of function arguments low (*Chapter 2.2*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-nargs

I.26: If you want a cross-compiler ABI, use a C-style subset (*Chapter 2.3*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abi

I.30: Encapsulate rule violations (*Chapter 5.3*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-encapsulate

F.21: To return multiple "out" values, prefer returning a struct or tuple (*Chapter 4.1*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-out-multi

F.51: Where there is a choice, prefer default arguments over overloading (*Chapter 1.2*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-default-args

C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead (*Chapter 1.3*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-default

C.47: Define and initialize member variables in the order of member declaration (*Chapter 2.4*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-order

C.90: Rely on constructors and assignment operators, not memset and memcpy (*Chapter 3.3*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-memset

C.131: Avoid trivial getters and setters (*Chapter 1.4*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-default-args

Enum.3: Prefer class enums over "plain" enums (*Chapter 4.2*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-class

ES.5: Keep scopes small (*Chapter 4.3*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-scope

ES.10: Declare one name (only) per declaration (*Chapter 1.5*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-task

ES.22: Don't declare a variable until you have a value to initialize it with (*Chapter 5.4*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-typesafe

ES.50: Don't cast away const (*Chapter 3.4*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const

Per.7: Design to enable optimization (*Chapter 5.5*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rper-efficiency

CP.3: Minimize explicit sharing of writable data (*Chapter 2.5*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-data

E.6: Use RAII to prevent leaks (*Chapter 5.6*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-raii

E.28: Avoid error handling based on global state (e.g. errno) (*Chapter 3.5*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-no-throw

Con.5: Use constexpr for values that can be computed at compile time (*Chapter 4.4*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconst-constexpr

T.1: Use templates to raise the level of abstraction of code (*Chapter 4.5*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-raise

T.10: Specify concepts for all template arguments (*Chapter 4.6*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-concepts

T.120: Use template metaprogramming only when you really need to (*Chapter 2.6*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-metameta

SF.7: Don't write using namespace at global scope in a header file (*Chapter 3.6*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rs-using-directive

NR.2: Don't insist to have only a single return-statement in a function (*Chapter 1.6*)
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rnr-single-return

Foreword

I enjoyed reading this book. I enjoyed it especially because it presents the C++ Core Guidelines (CG) very differently from how the CG itself does it. The CG presents its rules relatively tersely in a fixed format. The CG rules are often expressed in language-technical terms with an emphasis on enforcement through static analysis. This book tells stories, many coming from the games industry based on the evolution of code and techniques over decades. It presents the rules from a developer's point of view with an emphasis on what benefits can be obtained from following the rules and what nightmares can result from ignoring them. There are more extensive discussions of the motivation for rules than the CG themselves can offer.

The CG aims for a degree of completeness. Naturally, a set of rules for writing good code in general cannot be complete, but the necessary degree of completeness implies that the CG are not meant for a systematic read. I recommend the introduction and the philosophy section to get an impression of the aims of the CG and its conceptual framework. However, for a selective tour of the CG guided by taste, perspective, and experience, read the book. For true geeks, it is an easy and entertaining read. For most software developers, it offers something new and useful.

—Bjarne Stroustrup
June 2021

This page intentionally left blank

Preface

The complexity of writing C++ is diminishing with each new standard and each new piece of teaching literature. Conferences, blogs, and books abound, and this is a good thing. The world does not have enough engineers of sufficient quality to solve the very real problems we face.

Despite the continuing simplification of the language, there is still much to learn about how to write good C++. Bjarne Stroustrup, the inventor of C++, and Herb Sutter, the convenor of the standards body that maintains C++, have devoted considerable resources to creating teaching materials for both learning C++ and writing better C++. These volumes include *The C++ Programming Language*[1] and *A Tour of C++*,[2] as well as *Exceptional C++*[3] and *C++ Coding Standards*.[4]

The problem with books, even this modest volume, is that they represent a snapshot in time of the state of affairs, yet C++ is a continuously evolving language. What was good advice in 1998 may no longer be such a smart idea. An evolving language needs an evolving guide.

An online resource, C++ Core Guidelines,[5] was launched at the CppCon Conference in 2015 by Bjarne Stroustrup and Herb Sutter during their two[6] keynote[7] talks. The guidelines provide excellent, simple advice for improving your C++ style such that you can write correct, performant, and efficient code at your first attempt. It is the evolving guide that C++ practitioners need, and the authors will be delighted to review pull requests with corrections and improvements. Everyone, from beginners to veterans, should be able to follow its advisories.

1. Stroustrup, B, 2013. *The C++ Programming Language, Fourth Edition*. Boston: Addison-Wesley.
2. Stroustrup, B, 2018. *A Tour of C++, Second Edition*. Boston: Addison-Wesley.
3. Sutter, H, 1999. *Exceptional C++*. Reading, MA: Addison-Wesley.
4. Sutter, H, and Alexandrescu, A, 2004. *C++ Coding Standards*. Boston: Addison-Wesley.
5. Isocpp.github.io. 2021. C++ Core Guidelines. Copyright © Standard C++ Foundation and its contributors. Available at: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines Accessed 16 July 2021.
6. Youtube.com. 2021. CppCon 2015: Bjarne Stroustrup "Writing Good C++14". Available at: https://www.youtube.com/watch?v=1OEu9C51K2A Accessed 16 July 2021.
7. Youtube.com. 2021. CppCon 2015: Herb Sutter "Writing Good C++14... By Default." Available at: https://www.youtube.com/watch?v=hEx5DNLWGgA Accessed 16 July 2021.

The guidelines provide excellent, simple advice for improving your C++ style such that you can write correct, performant, and efficient code at your first attempt.

At the end of February 2020, on the #include discord,[8] Kate Gregory canvassed interest in producing a book about the Core Guidelines and I cautiously jumped at the chance. Kate gave a talk at CppCon 2017[9] where she looked at just 10 of the Core Guidelines. I share her enthusiasm for promoting better programming. I am the Head of Engineering Practice at Creative Assembly, Britain's oldest and largest game development studio, where I have spent a lot of the past 20-plus years helping to turn our fine engineers into even greater engineers. It is our observation that, despite the accessibility and simplicity of the Core Guidelines, many developers are not especially familiar with them. We want to promote their use, and we decided to write this book because there is not enough literature about them.

The Core Guidelines can be found at https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. They are absolutely jam-packed with excellent advice: indeed, it is hard to know where to start. Reading from the top to the bottom is feasible, but it is a tall order to grasp the entire set of advisories without repeated reading. They are organized into 22 major sections with titles like "Interfaces," "Functions," "Concurrency," and so on. Each section is composed of individual guidelines, sometimes a few, sometimes dozens. The guidelines are identified by their major section letter, then their number within the section, separated by a period. For example, "F.3: Keep functions short and simple" is the third guideline in section F, "Functions."

Each guideline is ordered in a similar way. It starts with the title of the guideline, which is presented as an action (do this, don't do this, avoid this, prefer this) followed by a reason and some examples, and possibly an exception to the guideline. Finally, there is a note on how to enforce the guideline. Enforcement notes range from advice to authors of static analysis tools to hints on how to conduct a code review. There is a skill to reading them, it turns out; deciding which ones to prioritize in your own code is a matter of personal discovery. Let us show you how to start taking advantage of their wisdom.

There are some sharp edges in C++ as well as some dusty corners that are not visited so often in modern C++. We want to steer you away from these. We want to show you that C++ does not have to be difficult, complex, or something that most developers cannot be trusted with.

8. #include <C++>. 2021. #include <C++>. Available at: https://www.includecpp.org/ Accessed 16 July 2021.
9. Youtube.com. 2021. CppCon 2017: Kate Gregory "10 Core Guidelines You Need to Start Using Now." Available at: https://www.youtube.com/watch?v=XkDEzfpdcSg Accessed 16 July 2021.

About This Book

In this book we offer what we consider to be 30 of the best C++ Core Guidelines. By thoroughly explaining these guidelines we hope that you will at least abide by them, even if you decide against investigating the remainder. The set that we have chosen are not necessarily the most important. However, they are certainly the set that will change your code for the better immediately. Of course, we hope that you will also see that there are many other good guidelines you could also follow. We hope that you will read the remainder and try them out in your code. Just as the Core Guidelines are aimed at all C++ developers with all levels of experience, so is this book aimed at the same set of people. The material does not increase in complexity as the book progresses, nor is there a required order in which to read the chapters. They are independent of one another, although they may explicitly refer to other chapters. We kept each chapter to about three thousand words, so you may decide that this is a bedside volume rather than a textbook. The purpose is not to teach you C++, but to advise you how to improve your style.

We divided the guidelines into five sections of six chapters, following Kate's original presentation to CppCon in 2017. In Section 1, "Bikeshedding is bad," we present guidelines that allow you to simply make a decision about when to do A or B, for some particular set of As and Bs, and move on with the minimum of fuss and argument. "Bikeshedding"[10] derives from C. Northcote Parkinson's "law of triviality," an argument that organization members typically give disproportionate weight to trivial issues, such as the color to paint a bikeshed compared to the testing criteria for the nuclear power station to which it is attached, because it is the one thing everyone knows something about.

In Section 2, "Don't hurt yourself," we present guidelines for preventing personal injury while writing code. One of the problems with the residual complexity of C++ is that there are several places where you can shoot yourself in the foot with ease. For example, while it is legal to populate a constructor initialization list in any order, it is never wise to do so.

Section 3 is named "Stop using that" and deals with parts of the language that are retained for backward compatibility reasons, along with pieces of advice that used to be valuable, but which have been superseded by developments in the language. As C++ evolves, things that seemed like a good idea at the time occasionally reveal themselves as rather less valuable than was originally expected. The standardization process fixes these things, but everyone needs to stay informed about them because you may come across examples if you find yourself working with a legacy codebase. C++ offers a guarantee of backward compatibility: code written 50 years ago in C should still compile today.

10. 2021. Available at: https://exceptionnotfound.net/bikeshedding-the-daily-software-anti-pattern/ Accessed 16 July 2021.

Section 4 follows on from this with the title "Use this new thing properly." Things like concepts, `constexpr`, structured binding, and so on need care when being deployed. Again, C++ is an evolving standard and new things appear with each release, all of which require some teaching to back them up. Although this text does not aim to teach you the new features of C++20, these guidelines do give you a flavor of how to apprehend novel features.

Section 5, the final section, is titled "Write code well by default." These are simple guidelines that, if followed, will result in you generating good code without having to think too hard about what is going on. They lead to the production of good idiomatic C++ which will be understood and appreciated by your colleagues.

Throughout the book, as with any good text, themes emerge and are developed. Part of the fun of writing this book, which I hope will translate to the reading of it too, has been seeing what motivates the guidelines and introspecting about the wider application of these motivations. Many of the guidelines, when squinted at carefully with the sun in the right place, restate some of the fundamental truths of software engineering in different ways. Extracting those truths will greatly improve your programming practice.

We truly hope you enjoy and profit from this book.

Access the Code

All of the code is available at the Compiler Explorer website. Matt Godbolt has kindly reserved stable links for each chapter which are formed by joining https://godbolt.org/z/cg30-ch and the chapter number. For example, https://godbolt.org/z/cg30-ch1.3 will take you to the complete code for Chapter 1.3. We recommend you start with https://godbolt.org/z/cg30-ch0.0 for instructions on how to use the website and interact with the code.

—Guy Davidson, @hatcat01 hatcat.com
—Kate Gregory, @gregcons gregcons.com
October 2021

Acknowledgments

The years 2020 and 2021 proved to be quite turbulent, and we have many people to thank for their support, both elective and incidental, during the development of this book.

Of course, we would like to thank Bjarne Stroustrup and Herb Sutter for the existence of the Core Guidelines and for their encouragement to write about them. We would also like to thank the attendees of CppCon for providing an audience to explore some of this.

Our families have provided vital support during what is a somewhat solitary process, and without them this effort would have been considerably harder.

The legion of friends on the #include discord, headquartered at includecpp.org, have continued to buoy us up in our daily C++ life since July 2017.[11] We will be donating one-tenth of our earnings from this book to you. All of you, please take a bow.

Several members of the ISO WG21 C++ committee, the body that maintains the standard, offered their help. We would like to thank Michael Wong and Tony van Eerd for their insight.

All the code examples are available at Compiler Explorer[12] with stable and intelligible links thanks to the generous efforts of Matt Godbolt, creator of this fine service. We extend our gratitude and remind him that the C++ community has profited greatly from his exertions.

Cppreference.com[13] was an excellent research tool during the initial preparation of each chapter, so we acknowledge the continuing efforts of the creator and host Nate Kohl, admins Povilas Kanapickas and Sergey Zubkov, along with Tim Song and all the other contributors, and thank them for maintaining this fine resource. They are heroes of the community.

After writing Chapter 3.6 it became clear that considerable inspiration came from an article by Arthur O'Dwyer. Many thanks to him for his continued service to the community. His blog also includes tales of his efforts to uncover some of the earliest computer-based text adventures from the 1970s and 1980s.[14]

A book like this requires an army of proofreaders, so we offer our thanks to Bjarne Stroustrup, Roger Orr, Clare Macrae, Arthur O'Dwyer, Ivan Čukić, Rainer Grimm, and Matt Godbolt.

The team at Addison-Wesley were invaluable, so we offer many thanks to Gregory Doench, Audrey Doyle, Aswini Kumar, Menka Mehta, Julie Nahil, and Mark Taber.

11. https://twitter.com/hatcat01/status/885973064600760320
12. https://godbolt.org/z/cg30-ch0.0
13. https://en.cppreference.com/w
14. https://quuxplusone.github.io/blog

This page intentionally left blank

About the Authors

J. Guy Davidson was first introduced to computing by way of the Acorn Atom in 1980. He spent most of his teenage years writing games on a variety of home computers: the Sinclair Research ZX81 and ZX Spectrum, as well as the Atari ST. After taking a mathematics degree from Sussex University, dabbling with theater, and playing keyboards in a soul band, he settled on writing presentation applications in the early 1990s and moved to the games industry in 1997 when he started working for Codemasters in their London office.

In 1999 he joined Creative Assembly where he is now the head of engineering practice. He works on the *Total War* franchise, curating the back catalogue, as well as improving the standard of programming among the engineering team. He serves on the IGGI advisory board, the BSI C++ panel, and the ISO C++ committee. He is the standards officer of the ACCU committee and serves on the program committee of the ACCU conference. He is a moderator on the #include<C++> discord server. He serves as code of conduct lead for several organizations. He can be found speaking at C++ conferences and meetups, particularly about adding linear algebra to the standard library.

In his bountiful spare time he offers C++ mentoring support through Prospela and BAME in Games; addresses schools, colleges, and universities through UKIE, STEMNet, and as a Video Game Ambassador; practices and teaches wu-style tai chi; studies the piano; sings first bass for the Brighton Festival Chorus; runs a local film club; is a voting member of BAFTA; has stood twice (unsuccessfully) for election to local council on behalf of The Green Party of England and Wales; and is trying to learn Spanish. You may occasionally find him at the card table playing bridge for a penny a point. There are probably other things: he is not one for letting the grass grow under his feet.

Kate Gregory met programming, some of her dearest friends, and the man she married all at the University of Waterloo in 1977 and has never looked back. Her degrees are in chemical engineering, which goes to show that you can't tell much about someone from what their degrees are in. Her rural Ontario basement has a small room with ancient computers: PET, C64, home-soldered 6502 system, and so on, as souvenirs of a simpler time. Since 1986 she has been running Gregory Consulting with her husband, helping clients across the world to be better at what they do.

Kate has done keynotes on five continents, loves finding brain-changing truths and then sharing them, and spends a great deal of time volunteering in various C++ activities. Dearest of these is #include <C++>, which is changing this industry to be more welcoming and inclusive. Their Discord server is a warm and gentle place to learn C++ as a beginner, to collaborate on a paper for WG21 to change the language we all use, or anything in between.

She is pulled from her keyboard by her grandchildren, Ontario lakes and campsites, canoe paddles and woodsmoke, and the lure of airports worldwide. A foodie, a board game player, and someone who cannot resist signing up to help with things, she is as active offline as online, but less visible. Since surviving stage IV melanoma in 2016, she worries less about what others think and what is expected, and more about what she wants for her own future. It's working well.

Chapter 3.2

I.3: Avoid singletons

Global objects are bad

Global objects are bad, m'kay? You will hear this all the time, from programmers young and old, recited as an article of faith. Let's look into why this is.

A global object lives in the global namespace. There is only one of these, hence the name "global." The global namespace is the outermost declarative region of a translation unit. A name with global namespace scope is said to be a global name. Any object with a global name is a global object.

A global object is not necessarily visible to every translation unit of a program; the one-definition rule means that it can only be defined in one translation unit. However, a declaration can be repeated in any number of translation units.

Global objects have no access restrictions. If you can see it, you can interact with it. Global objects have no owner other than the program itself, which means no single entity is responsible for it. Global objects have static storage duration, so they are initialized at startup (or static initialization) and destroyed at shutdown (or static deinitialization).

This is problematic. Ownership is fundamental to reasoning about objects. Since nothing owns a global object, how can you reason about its state at any time? You might be calling functions on that object and then, suddenly and without warning, another entity may call other functions on that object without your knowledge.

Worse still, since nothing owns global objects, their construction sequence is not determined by the standard. You have no idea in which order global objects will be constructed, which leads to a rather frustrating category of bug that we shall cover later.

Singleton Design Pattern

Having convinced you of the harm that global objects cause to your codebase, let us turn our attention to singletons. I first encountered this term in 1994 when the book *Design Patterns*[1] was published. This venerable tome was a tremendously exciting read at the time and is still a very useful book to have on your shelf or your e-reader. It describes patterns that recur in software engineering, in much the same way that patterns recur in conventional architecture, such as cupola, portico, or cloister. What was so welcome about this book was that it identified common patterns in programming and gave them names. Naming is hard, and having someone do the naming for us was a great boon.

The book categorizes the patterns in three ways, as creational, structural, or behavioral patterns. It is within the creational division that we find the singleton, which restricts object creation for a class to only one instance. Of course, with such a fabulous text outlining such a well-used pattern, it was taken for granted that using a singleton was A Good Thing. After all, we had all been using something like singletons for years, we just had not yet given them a name that we could all agree on.

A popular example of a singleton is the main window. The main window is where all the action happens, collecting user input and displaying results. You should only create one main window, so it might make sense to prevent the creation of another. Another example is the manager class. This is characterized by including the name "manager" in the identifier. This is a strong sign that in fact a singleton has been created, and that there are problems deciding about ownership of whatever is being managed.

Static initialization order fiasco

Singletons are prone to the static initialization order fiasco.[2] This term was coined by Marshall Cline in his C++ FAQ and characterizes the problem of dependent objects being constructed out of order. Consider two global objects, A and B, where the constructor of B uses some functionality provided by A and so A must be constructed first. At link time, the linker identifies the set of objects with static storage duration,

1. Gamma, E, Helm, R, Johnson, R, and Vlissides, J, 1994. *Design Patterns*. Reading, MA: Addison-Wesley.
2. "Fiasco" is possibly an unfair characterization. Static initialization was never supposed to offer a topological ordering of initialization. That was infeasible with separate compilation, incremental linking, and linkers from the 1980s. C++ had to live with the existing operating systems. This was a time when systems programmers were used to living with sharp tools.

sets aside an area of the memory for them to exist in, and creates a list of constructors to be called before `main` is called. At runtime, this is called static initialization.

Now, although you can identify that B depends on A and so A must be constructed first, there is no standard way to signal to the linker that this is the case. Indeed, how could you do that? You would need to find some way of exposing the dependency in the translation unit, but the compiler only knows about the translation unit it is compiling.

We can hear your brow furrowing. "Well, what if I told the linker what order to create them in? Could the linker be modified to accommodate that?" In fact, this has been tried. Long ago I used an IDE called Code Warrior, by Metrowerks. The edition I was using exposed a property that allowed me to dictate the order of construction of static objects. It worked fine, for a while, until I unwittingly created a subtle circular dependency that took me the better part of twenty hours to track down.

By keeping your object dependencies in a single translation unit, you avoid all of these problems while maintaining clarity of purpose and separation of concerns.

You aren't convinced. "Circular dependencies are part and parcel of engineering development. The fact that you managed to create one because you got your relationships wrong shouldn't preclude the option to dictate the creation order at static initialization." Indeed, I did actually resolve the problem and carried on, but then I needed to port the codebase to another toolchain which didn't support this feature. I was programming in nonstandard C++ and paid the price when I attempted portability.

"Nonetheless," you continue, "this is something the committee COULD standardize. Linkage specifications are already in the purview of the standard. Why not initialization order specification?" Well, another problem with static initialization order is that there is nothing to stop you starting multiple threads during static initialization and requiring an object before it has been created. It is far too easy to shoot yourself in the foot with dependencies between global static objects.

The committee is not in the habit of standardizing footguns. Dependency on the order of initialization is fraught with peril, as demonstrated in the prior paragraphs, and allowing programmers to command this facility is unwise at best. Additionally, it militates against modular design. Static initialization order IS specified per translation unit by order of declaration. Specification between translation units is where it all falls down. By keeping your object dependencies in a single translation unit, you avoid all of these problems while maintaining clarity of purpose and separation of concerns.

The word "linker" appears ONCE in the standard.[3] Linkers are not unique to C++; linkers will bind together anything of the appropriate format, regardless of what compiler emitted it, be it C, C++, Pascal, or other languages. It is a steep demand to require that linkers suddenly support a new feature solely for the benefit of promoting a dicey programming practice in one language. Cast the idea of standardizing initialization order from your mind. It is a fool's errand.

Having said that, there is a way around the static initialization order fiasco, and that is to take the objects out of the global scope so that their initialization can be scheduled. The easiest way to do this is to create a simple function containing a static object of the type required, which the function returns by reference. This is sometimes known as the Meyers Singleton after Scott Meyers, who described this approach in his book *Effective C++*.[4] The technique itself is much older than that, having been used in the 1980s. For example:

```
Manager& manager() {
  static Manager m;
  return m;
}
```

Now the function is global, rather than the object. The `Manager` object will not be created until the function is called: static data at function scope falls under different initialization rules. "But," you may ask, "what about the concurrency problem? Surely, we still have the same issue of multiple threads trying to access the object before it has been fully created?"

Fortunately, since C++11 this is also thread safe. If you look at section [stmt.dcl][5] in the standard you will see the following: "If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization." This is not the end of your troubles, though: you are still distributing a handle to a single mutable object, with no guarantee of thread-safe access to that object.

How to hide a singleton

You might look at that and decide that we have simply hidden a singleton behind a function. Indeed, hiding singletons is easy and the Core Guidelines remarks that enforcing their nonuse is very hard in general. The first enforcement idea offered by

3. https://eel.is/c++draft/lex.name
4. Meyers, S, 1998. *Effective C++*. Reading, MA: Addison-Wesley.
5. https://eel.is/c++draft/stmt.dcl

this specific Core Guideline I.3: "Avoid singletons," is "look for classes with names that include singleton." This might seem somewhat specious, but since Singleton is one of the Design Patterns it is remarkably common for engineers to add it to the name of a class, to identify that "this is a singleton" or "I have read the *Design Patterns* book." Of course, doing so embeds the implementation in the interface, which is A Bad Thing, but that is another topic.

The second idea offered by the guideline is "look for classes for which only a single object is created (by counting objects or by examining constructors)." This requires a complete, manual, class-by-class audit of your codebase. Sometimes singletons are created by accident. An abstraction may be inducted and a class formed from it, and all the scaffolding required to manage the life cycle of and interactions with that class may be created, such as the special functions, public interface, and so on, but only one instance of the object may ever exist at one time. It may not have been the engineer's intention to create a singleton, but that is what has happened; a count of all the instances reveals the quantity to be one.

The final idea is "If a class X has a public static function that contains a function-local static of the class type X and returns a pointer or reference to it, ban that." This is exactly the technique described above to resolve the static initialization order fiasco. The class may have a superset of the following interface:

```
class Manager
{
public:
  static Manager& instance();

private:
  Manager();
};
```

The giveaway here is the private constructor. Nothing can create this object except a static member or a friend, and we see no friend declarations. Nothing can derive from it unless another constructor is added to the nonprivate interface. The private constructor indicates that "my construction is tightly controlled by other functions in my interface" and lo! And behold! The public interface contains a static function which returns a reference to an instance. You will no doubt be able to guess the general content of this member function by looking at the `manager()` example function above.

A subtle variation of this is the reference-counted singleton. Consider a class that is a huge resource hog. Not only do you not want two instances of these to exist at once, but you also want it to be destroyed the moment it is no longer needed. This is somewhat complex to manage, since you need a shared pointer, a mutex, and a

reference counter. However, this is still a singleton and falls under the "Avoid singletons" guideline.

You might be looking at that public static member function and saying to yourself "surely the guideline should say 'Avoid static storage duration objects.' They are singletons, after all." Hold that thought.

But only one of these should ever exist

Throughout the teaching of C++ there have been some popular examples to describe object orientation. Gas stations have cars, pumps, a cash desk, tankers delivering fuel, prices, and so on, yielding an ecosystem rich enough to describe many kinds of relationships. In the same vein, restaurants have tables, customers, menus, a serving hatch, wait staff, chefs, food deliveries, garbage collection, and other features. In today's textbooks they probably also have a website and a Twitter account.

Both examples have one thing in common: an abstraction that should only exist singly. The gas station has one cash desk. The restaurant has one serving hatch. Surely these are singletons? If not, what is to be done?

One solution we have seen to this problem is to create a class with an entirely static interface. All the public member functions and the private data are static. We now want to take a diversion and tell you about W. Heath Robinson. Born in 1872 in Finsbury Park, London, this English cartoonist was best known for his drawings of ludicrously elaborate machines that went to great lengths to solve simple problems. One of the automatic analysis machines built for Bletchley Park during the Second World War to assist in the decryption of German message traffic was named "Heath Robinson" in his honor. I was given a book of his cartoons as a young child and marveled at the intricacy of the operation of his devices. He had an American counterpart, Rube Goldberg, born in July 1883 in San Francisco, who also drew overly complex devices, and inspired the board game Mouse Trap. Their names have passed into common parlance in the English language to describe overengineering.

This is precisely what a class with an entirely static interface is an example of. When you create a class, you create a public interface for viewing and controlling the abstraction, and a pile of data and nonpublic functions for modeling the abstraction. However, if there is only one instance of all the data, why do you need to attach it to a class? You can simply implement all the public member functions in one source file and put the single instance of the data and all the nonpublic functions in an anonymous namespace.

In fact, why are you bothering with a class at all?

What we have arrived at, in a self-referentially convoluted way, is the correct solution to the problem of singletons (small s). They should be implemented as namespaces rather than classes. Rather than this:

```
class Manager
{
public:
  static int blimp_count();
  static void add_more_blimps(int);
  static void destroy_blimp(int);

private:
  static std::vector<Blimp> blimps;
  static void deploy_blimp();
};
```

you should declare this:

```
namespace Manager
{
  int blimp_count();
  void add_more_blimps(int);
  void destroy_blimp(int);
}
```

The implementation does not need to be exposed to the client like some Heath Robinson drawing of marvelous and fascinating complexity. It can be hidden away in the dark recesses of a private implementation file. This has the additional advantage of improving the stability of the file in which the namespace is declared, minimizing large-scale dependent recompilation. Of course, the data used to model the abstraction will not be owned by an object, so it will be static. Beware of the static initialization order fiasco as described above.

Wait a moment…

You might be looking at this namespace solution and remarking to yourself "but this is still a Singleton."

It is not a Singleton. It is a singleton. The problem that the guideline is warning about is the Singleton pattern, not the existence of single-instance abstractions. Indeed, in an interview with InformIT in 2009, Erich Gamma, one of the four authors of *Design Patterns,* remarked that he wanted to remove Singleton from the catalogue.[6]

6. https://www.informit.com/articles/article.aspx?p=1404056

There are two problems that we have with C++ advice. The first is that what was smart advice once may not remain smart advice forever.

At the moment, a new version of C++ is released every three years. The introduction of `std::unique_ptr` and `std::shared_ptr` in 2011 changed the advice on how we matched `new` and `delete` pairs ("Don't delete an object in a different module from where it was created") by making it entirely feasible to never use raw `new` and `delete`, as advised by Core Guideline R.11: "Avoid calling `new` and `delete` explicitly." Learning a set of advisories and then moving on with your life is not sufficient: you need to continually review advice as the language grows and changes.

What was smart advice once may not remain smart advice forever.

An immediate manifestation of this problem is that you may have a favorite framework that you use extensively, which may contain idiomatic use of C++ that has been deprecated. Perhaps it contains a Singleton for capturing and manipulating environment variables, or settings informed by the command-line parameters which may be subject to change. You might feel that your favorite framework can do no wrong, but that is not the case. Just as scientific opinion changes with the arrival of new information, so does best C++ practice. This book that you are reading today may contain some timeless advice, but it would be supremely arrogant and foolish of me to suggest that the entire text is wisdom for the ages, with stone-carved commandments about how you should write C++.

The second problem is that advisories are the distillation of several motivations, often hidden entirely from the snappy and memorable phrase that sits in our immediate recall. "Avoid singletons" is much easier to remember than "avoid overengineering single-instance abstractions into a class and abusing access levels to prevent multiple instantiations." Learning the advice is not enough. You must learn the motivations so that you know why you are taking a particular approach, and when it is safe not to do so.

C++ Core Guidelines is a living document with a GitHub repository on which you can make pull requests. It contains hundreds of advisories with varying amounts of motivation, and the purpose of this book is to highlight some of the deeper motivations for 30 of them.

Earlier we remarked that you may be thinking that all static objects are Singletons, so all static objects should be avoided. You should be able to see now that static objects are not Singletons, nor are they necessarily singletons. They are an instance of an object whose duration is the entire duration of the program. Nor are they necessarily globals: static data members have class scope, not global scope.

Similarly, "Globals are bad, m'kay?" is not universally the case. It is global mutable state that can hurt you, as revealed in Core Guideline I.2: "Avoid non-`const` global

variables." If your global object is immutable, then it is merely a property of your program. For example, while writing a physics simulation for a space game we could quite reasonably declare an object of type `float` called `G`, which is the gravitational constant, in the global namespace like this:

```
constexpr float G = 6.674e-11; // Gravitational constant
```

After all, it is a universal constant. Nobody should be changing this. Of course, you might decide that the global namespace is not the right place for such a thing, and declare a namespace called `universe` like this:

```
namespace universe {
  constexpr float G = 6.674e-11; // Gravitational constant
}
```

There is an outside chance that you might want to experiment with a universe with a different gravitational constant; in this case you may want to use a function that simply returns a value, and then change the logic behind the interface according to your crazy experimental needs.

The point is that you know WHY globals are bad, for the reasons enumerated earlier, and you can decide when it is appropriate to bend that rule, with a full understanding of the technical debt you are taking on.

Summary

In summary:

- Avoid singletons: the pattern, not the single-instance abstraction.
- Prefer a namespace to a class to model this type of abstraction.
- Use static data carefully when implementing a singleton.
- Understand the motivations for the Core Guidelines.
- Review the Core Guidelines as the C++ language grows and evolves.

Index

A

B

C

D

E

F

G

M

N

O

P

Q

R

S

T