

What Every Professional C++ Programmer Needs to Know—Pared to Its Essentials So It Can Be Efficiently and Accurately Absorbed



# C++ COMMON KNOWLEDGE

ESSENTIAL INTERMEDIATE PROGRAMMING



STEPHEN C. DEWHURST

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales  
(800) 382-3419  
corpsales@pearsontechgroup.com

For sales outside of the U. S., please contact:

International Sales  
international@pearsoned.com

Visit us on the Web: [www.awprofessional.com](http://www.awprofessional.com)

*Library of Congress Cataloging-in-Publication Data:*

Dewhurst, Stephen C.

C++ common knowledge : essential intermediate programming / Stephen C. Dewhurst.

p. cm.

Includes bibliographical references and index.

ISBN 0-321-32192-8 (pbk. : alk. paper)

1. C++ (Computer program language) I. Title.

QA76.73.C153D48797 2005

005.13'3—dc22

2004029089

Copyright © 2005 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.  
Rights and Contracts Department  
One Lake Street  
Upper Saddle River, NJ 07458

ISBN 0-321-32192-8

Text printed in the United States at Demand Print Center in Old Tappan, New Jersey.

5th Printing July 2009

# Preface

*A successful book is not made of what is in it, but what is left out of it.*

—Mark Twain

*... as simple as possible, but no simpler.*

—Albert Einstein

*... a writer who questions the capacity of the person at the other end of the line is not a writer at all, merely a schemer.*

—E.B. White

When he took over the editorship of the late *C++ Report*, the quick Herb Sutter asked me to write a column on a topic of my choosing. I agreed, and I chose to call the column “Common Knowledge.” It was supposed to be, in Herb’s words, “a regular summary of basic lore that every working C++ programmer should know—but can’t always.” After a couple of columns in that vein, however, I became interested in template metaprogramming techniques, and the topics treated in “Common Knowledge” from that point on were far from common.

However, the problem in the C++ programming industry that motivated my original choice of column remains. I commonly encounter the following types of individuals in my training and consulting work:

- Domain experts who are expert C programmers but who have only basic knowledge of (and perhaps some animosity toward) C++
- Talented new hires direct from university who have an academic appreciation for the C++ language but little production C++ experience

- Expert Java programmers who have little C++ experience and who have a tendency to program in C++ the way one would program in Java
- C++ programmers with several years of experience maintaining existing C++ applications but who have not been challenged to learn anything beyond the basics required for maintenance

I want to be immediately productive, but many of the people with whom I'm working or who I'm training require preliminary education in various C++ language features, patterns, and coding techniques before we can get down to business. Worse, I suspect that most C++ code is written in ignorance of at least some of these basics and is therefore not what most C++ experts would consider to be production quality.

This book addresses this pervasive problem by providing essential, common knowledge that every professional C++ programmer needs to know, in a form that is pared to its essentials and that can be efficiently and accurately absorbed. Much of the information is already available from other sources or is part of that compendium of unwritten information that all expert C++ programmers know. The advantage is that this material resides in one place and was selected according to what my training and consulting experience over many years has shown are the most commonly misunderstood and most useful language features, concepts, and techniques.

Perhaps the most important aspect of the sixty-three short items that make up this book is what they leave out, rather than what they contain. Many of these topics have the potential to become complex. An author's ignorance of these complexities could result in an uninformed description that could mislead the reader, but an expert discussion of a topic in its full complexity could inundate the reader. The approach used here is to filter out *needless* complexity in the discussion of each topic. What remains, I hope, is a clear distillation of the essentials required for production C++ programming. C++ language wonks will recognize, therefore, that I've left out discussion of some issues that are interesting and even important from a theoretical perspective, but the ignorance of which does not commonly affect one's ability to read and write production C++ code.

Another motivation for this book came as I was engaged in conversation with a group of well-known C++ experts at a conference. There was a

general pall or depression among these experts that modern C++ is so complex that the “average” programmer can no longer understand it. (The specific issue was name binding in the context of templates and namespaces. Yes, getting worked up about such a topic does imply the need for more play with normal children.) On reflection, I’d have to say our attitude was pretentious and our gloom unwarranted. We “experts” have no such problems, and it’s as easy to program in C++ as it is to speak a (vastly more complex) natural language, even if you can’t diagram the deep structure of your every utterance. A recurring theme of this book is that while the full description of the minutia of a particular language feature may be daunting, day-to-day use of the feature is straightforward and natural.

Consider function overloading. A full description occupies a large chunk of the standard and whole or multiple chapters in many C++ texts. And yet, when faced with

```
void f( int );
void f( const char * );
//...
f( "Hello" );
```

not a single practicing C++ programmer will be unable to determine which `f` is called. Full knowledge of the rules by which a call to an overloaded function is resolved is useful but only rarely necessary. The same applies to many other ostensibly complex areas of C++ language and idiom.

This is not to say that all the material presented here is easy; it’s “as simple as possible, but no simpler.” In C++ programming, as in any other worthwhile intellectual activity, many important details can’t be written on an index card. Moreover, this is not a book for “dummies.” I feel a great deal of responsibility to those who grant a portion of their valuable time to reading my books. I respect these readers and try to communicate with them as I would in person to any of my colleagues. Writing at an eighth-grade level to a professional isn’t writing. It’s pandering.

Many of the book’s items treat simple misunderstandings that I’ve seen over and over again, which just need to be pointed out (for example, scope order for member function lookup and the difference between overriding and overloading). Others deal with topics that are in the process of becoming essential knowledge for C++ professionals but are

often incorrectly assumed to be difficult and are avoided (for example, class template partial specialization and template template parameters). I've received some criticism from the expert reviewers of the manuscript that I've spent too much space (approximately one third of the book) on template issues that are not really common knowledge. However, each of these experts pointed out one, two, or several of the template topics they thought did belong in the book. The telling observation is, I think, that there was little overlap among these suggestions, and every template-related item had at least one supporter.

This is the crux of the issue with the items that make up this book. I don't expect any reader to be ignorant of every item's topic, and it's likely that some readers will be familiar with all of them. Obviously, if a reader is not familiar with a particular topic, there would be (I presume) some benefit in reading about it. However, even if a reader is already familiar with a topic, I'd hope that reading about it from a new perspective might clear up a slight misunderstanding or lead to a deeper understanding. This book may also have a role in saving the more experienced C++ programmer precious time. Competent C++ programmers often find themselves (as described previously) answering the same questions over and over again to the detriment of their own work. I'd suggest that the approach of "read this first, and *then* let's talk" would save these C++ gurus countless hours and direct their expertise instead to the complex problems for which it's really needed.

I initially tried to group these sixty-three items into neat chapters, but the items had other ideas. They instead tended to clump themselves together in ways that ranged from the obvious to the unexpected. For example, the items related to exceptions and resource management form a rather natural group. Less obviously, the items *Capability Queries*, *Meaning of Pointer Comparison*, *Virtual Constructors and Prototype*, *Factory Method*, and *Covariant Return Types* are strongly and somewhat surprisingly interrelated and are best grouped in close proximity to each other. *Pointer Arithmetic* decided to hang with *Smart Pointers* rather than with the pointer and array material earlier in the book. Rather than attempt to impose an arbitrary chapter structure on these natural groupings, I decided to grant the individual items freedom of association. Of course, many other interrelationships exist among the topics treated by the items than can be represented in a simple linear ordering, so the items make frequent internal references among themselves. It's a clumped but connected community.

While the main idea is to be brief, discussion of a topic sometimes includes ancillary details that are not directly related to the subject at hand. These details are never necessary to follow the discussion, but the reader is put on notice that a particular facility or technique exists. For instance, the `Heap` template example that appears in several items informs the reader in passing about the existence of the useful but rarely discussed STL heap algorithms, and the discussion of placement new outlines the technical basis of the sophisticated buffer management techniques employed by much of the standard library. I also try to take the opportunity, whenever it seems natural to do so, to fold the discussion of subsidiary topics into the discussion of a particular, named item. Therefore, *RAII* contains a short discussion of the order of constructor and destructor activation, *Template Argument Deduction* discusses the use of helper functions for specializing class templates, and *Assignment and Initialization Are Different* folds in a discussion of computational constructors. This book could easily have twice the number of items, but, like the clumping of the items themselves, correlation of a subsidiary topic with a specific item puts the topic in context and helps the reader to absorb the material efficiently and accurately.

I've reluctantly included several topics that cannot reasonably be treated in this book's format of short items. In particular, the items on design patterns and the design of the standard template library are laughably short and incomplete. Yet they make an appearance simply to put some common misconceptions to rest, emphasize the importance of the topics, and encourage the reader to learn more.

Stock examples are part of our programming culture, like the stories that families swap when they gather for holidays. Therefore, `Shape`, `String`, `Stack`, and many of the other usual suspects put in an appearance. The common appreciation of these baseline examples confers the same efficiencies as design patterns in communication, as in "Suppose I want to rotate a `Shape`, except..." or "When you concatenate two `Strings`..." Simply mentioning a common example orients the conversation and avoids the need for time-consuming background discussion. "You know how your brother acts whenever he's arrested? Well, the other day..."

Unlike my previous books, this one tries to avoid passing judgment on certain poor programming practices and misuses of C++ language features; that's a goal for other books, the best of which I list in the bibliography. (I was, however, not entirely successful in avoiding the tendency to

preach; some bad programming practices just have to be mentioned, even if only in passing.) The goal of this book is to inform the reader of the technical essentials of production-level C++ programming in as efficient a manner as possible.

—Stephen C. Dewhurst  
Carver, Massachusetts  
January 2005

## Item 3 | Design Patterns

Anyone who is not already familiar with design patterns may, after a brief survey of the field, come away with the impression that design patterns are a lot of marketing hype, are just some simple coding techniques, or are the playthings of computer scientists who really should get out more. While each of these impressions carries a grain of truth, design patterns are an essential component of the professional C++ programmer's toolkit.

A “design pattern” is a recurring architectural theme that provides a solution to a common design problem within a particular context and describes the consequences of this solution. A design pattern is more than a simple description of a technique; it's a named capsule of design wisdom gleaned from successful existing practice, written in such a way that it can be easily communicated and reused. Patterns are about programmer to programmer communication.

From a practical perspective, design patterns have two important properties. First, they describe proven, successful design techniques that can be customized in a context-dependent way to new design situations. Second, and perhaps more important, mentioning the application of a particular pattern serves to document not only the technique that is applied but also the reasons for its application and the effect of having applied it.

This sort of thing is nothing new. Consider an analogy from the field of algorithms. (Algorithms are not design patterns, and they're not “code patterns.” They're algorithms, and this is an analogy.) Consider the following statement that I might make to a colleague: “I have an unsorted sequence that I have to search a number of times. Therefore, I'm going to quick sort it and use binary search to perform each lookup.” The ability to use the terms “quick sort” and “binary search” is of inestimable value not only in design but also in communicating that design to an educated colleague. When I say “quick sort,” my colleague knows that the sequence I'm sorting is in a random access structure, that it will probably be sorted within  $O(n \lg_2 n)$  time, and that the elements in the sequence may be compared with a less-than-like operator. When I say “binary search,” my

colleague knows (even if I hadn't earlier mentioned "quick sort") that the sequence is sorted, that I will locate the item of interest within  $O(\lg_2 n)$  comparisons, and that an appropriate operation is available to compare sequence elements. Shared knowledge of, and a standard vocabulary for, standard algorithms permits not only efficient documentation but also efficient criticism. For example, if I planned to perform this search and sort procedure on a singly linked list structure, my colleague would immediately smirk and point out that I couldn't use quick sort and probably wouldn't want to use binary search.

Until the advent of design patterns, we missed these advantages in documentation, communication, and efficient smirking with our object-oriented designs. We were forced into low-level descriptions of our designs, with all the inefficiency and imprecision that entails. It's not that techniques for sophisticated object-oriented design didn't exist; it's that the techniques were not readily available to the entire programming community under a shared, common terminology. Design patterns address that problem, and we can now describe object-oriented designs as efficiently and unambiguously as algorithmic designs.

For example, when we see that the Bridge pattern has been applied to a design, we know that at a simple mechanical level an abstract data type implementation has been separated into an interface class and an implementation class. Additionally, we know that the reason this was done was to separate strongly the interface from the implementation so that changes to the implementation would not affect users of the interface. We also know a runtime cost exists for this separation, how the source code for the abstract data type should be arranged, and many other details. A pattern name is an efficient, unambiguous handle to a wealth of information and experience about a technique, and careful, accurate use of patterns and pattern terminology in design and documentation clarifies code and designs.

Patterns works sometimes describe design patterns as a form of literature (they really do) that follows a certain formal structure. Several common variants are in use, but all forms contain four essential parts.

First, a design pattern must have an unambiguous name. For example, the term "wrapper" is useless for a design pattern, because it is already in common use and has dozens of meanings. Using a term like "Wrapper" as a pattern name would lead only to confusion and misunderstanding.

Instead, the different design techniques that formerly went under the name “wrapper” are now designated by the pattern names “Bridge,” “Strategy,” “Façade,” “Object Adapter,” and probably several others. Use of a precise pattern name has a clear advantage over using a less precise term, in the same way that “binary search” is a more precise and useful term than “lookup.”

Second, the pattern description must define the problem addressed by the pattern. This description may be relatively broad or narrow.

Third, the pattern description describes the problem’s solution. Depending on the statement of the problem, the solution may be rather high level or relatively low level, but it should still be general enough to customize according to the various contexts in which the problem may occur.

Fourth, the pattern description describes the consequences of applying the pattern to the context. How has the context changed for better or worse after application of the pattern?

Will knowledge of patterns make a bad designer a good designer? Time for another analogy: Consider one of those painful mathematics courses you may have been forced to undergo, in which the final examination is to prove a number of theorems in a certain area of mathematics. How do you get out of such a course alive? One obvious way is to be a genius. Starting from first principles, you develop the underpinnings of an entire branch of mathematics and eventually prove the theorems. A more practical approach would be to memorize and internalize a large number of theorems in that area of mathematics and use whatever native mathematical ability, inspiration, or good luck you have at your disposal to select the appropriate subsidiary theorems and combine them with some logical glue to prove the new theorems. This approach is advantageous even for our fictitious genius, because a proof built upon established theorems is more efficient to construct and easier to communicate to mere mortals. Familiarity with subsidiary theorems does not, of course, guarantee that a poor mathematician will be able to pass the test, but such knowledge will at least enable that person to understand the proof once it has been produced.

In a similar vein, developing a complex object-oriented design from first principles is probably going to be tedious, and communication of the eventual design difficult. Composition of design patterns to produce an object-oriented design is similar to use of subsidiary theorems in mathematics to

prove a new theorem. Design patterns are often described as “micro-architectures” that can be composed with other patterns to produce a new architecture. Of course, selecting appropriate patterns and composing them effectively requires design expertise and native ability. However, even your manager will be able to understand the completed design if he or she has the requisite knowledge of patterns.

# Index

See *also* Index of Code Examples, page 245

->\* (dash angle bracket asterisk), pointer-to-member operator, 58  
-> operator, overloading, 145–146, 147–148  
( ) (parentheses)  
  function call operator, 58  
  grouping in declarations, 61  
  grouping with pointer-to-member operators, 58  
\* (asterisk) operator, overloading, 145–146, 147–148  
. \* (dot asterisk), pointer-to-member operator, 58  
1984 (veiled reference), 224

## A

ABC (abstract base class), 113  
abort, 117, 140  
Abstract data type. *See* data abstraction.  
Access protection, 88, 111, 113–115  
Accessor functions. *See* get/set interfaces.  
Address arithmetic. *See* pointer arithmetic.  
ADL (argument dependent lookup), 89–90  
Alexandrescu, Andrei, xviii  
Aliases, 84. *See also* references.  
Allison, Chuck, xvii  
Allocating arrays, 127–129

Allocators, rebind convention, 180  
Anonymous namespaces, 84–85  
Argument dependent lookup (ADL), 89–90  
Arguments, templates, 153–154  
Array declarators, pointers to, 61–62  
Array formal arguments, 17–19  
Array index calculation. *See* pointer arithmetic.  
Arrays  
  allocation/deallocation, 127–129  
  auto\_ptr references, 148  
  of class objects, 120–121  
  decay, 17  
  as function arguments. *See* array formal arguments.  
  memory management, 127–129  
  of pointers, 25  
  references to, 62  
  sorting, 221–222  
Assignment. *See also* copying; initialization.  
  computational constructor, 43  
  construction, 42  
  copying, 45–47  
  destruction, 42–43  
  exception safe copy, 46  
  *versus* initialization, 41–43  
  user-defined types, 41–43  
  virtual copy, 47

  and virtual function table pointers, 38  
Asterisk (\*) operator, overloading, 145–146, 147–148  
Audit trails, resource control. *See* RAII.  
auto keyword, 233  
auto\_ptr  
  array references, 148  
  as container elements, 148  
  conversions, 147–148  
  description, 147–148  
  operator overloading, 147  
  *versus* smart pointers, 148

## B

Base classes  
  abstract bases, manufacturing, 113–115  
  forcing to abstract base, 113–115  
  implementation guidelines, 77–79  
  member function, determining, 77–79  
  polymorphic, 3–5  
  Template Method, 77–79  
Body objects, 117  
Boedigheimer, Kim, xvii  
Bradbury, Ray (veiled reference), 50  
The Brother, xv, 81, 147

## C

- Callbacks. *See also* Command pattern.
  - definition, 67
  - “don’t call us, we’ll call you,” 68
  - framework code, 68
  - function objects as, 67–70.
    - See also* Command pattern.
  - function pointers, 50–51
  - Hollywood Principle, 68
- Capability queries, 93–95
- Cast operators
  - casting down
    - an inheritance hierarchy, 30–31
    - from a pointer to a base class, 31
    - to a reference type, 32
  - const qualifiers,
    - adding/removing, 29–30
  - const\_cast, 29–30
  - conversion across
    - a hierarchy, 94–95
  - cross-cast, 94–95
  - dynamic\_cast, 31–32
  - function style, 29
  - new style
    - description, 29–32
    - versus* old-style, 29
  - old-style *versus* new style, 29
  - reinterpret\_cast, 31
  - static\_cast, 30–32
  - type qualifiers, changing, 29–30, 30–31
  - volatile qualifiers,
    - adding/removing, 29–30
- Class layout
  - assignment, and virtual function table pointers, 38
  - contravariance, 54–55
  - covariant return types, 109
  - member offsets, 39
  - pointer comparison, 97–98
  - virtual functions, 37
  - virtual inheritance, 37–38
    - what you see is what you get, 37
- Class members, pointers
  - to *versus* pointers, 53–56
- Class objects. *See* objects.
- Classes
  - handle, exception safe swaps, 46
  - interface, 93–94
- Cloning objects, 99–101. *See also* assignment; copying; initialization; Prototype.
- Command pattern, 69. *See also* callbacks.
- Communication, with other programmers
  - data abstraction, 2
  - design patterns, 7–8
  - identifiers in template declarations, 201–202
  - versus* ignorance, 224
  - overloading, 214–215
  - typedefs, 62
- Comparator operations, 71
- Comparators, STL function objects as, 72–73
- Complete specialization. *See* explicit specialization.
- Computational
  - constructor, 41, 43
- Const member functions
  - lazy evaluation, 34
  - logically const, 35
  - meaning of, 33–36
  - modifying objects, 34
  - overloaded index operator, 35–36
- Const pointers, *versus* pointers
  - to const, 21–23
- const qualifiers,
  - adding/removing, 29–30
- const\_cast operator, 29–30
- Construction
  - assignment, 42
  - copying, 45–47
- Construction order, 141–142
- Constructors
  - calling, 119–121
  - description, 143–144
  - exception safety, 143–144
  - operator overloading, 143–144
  - placement new, 119–121
  - protected, 114–115
  - virtual, 99–101
- Container elements,
  - auto\_ptr as, 148
- Contract, base class as, 4–5
- Contravariance
  - class layout, 54–55
  - member templates, 174
  - pointers to data members, 54–55
  - pointers to member functions, 54–55, 58
- Conventions. *See also* idioms.
  - anonymous temporary function object, 73
  - class vs. typename, 232–233
  - copy operations, 45
  - exception safety axioms, 131–133
  - generic programming, 191, 193, 207, 223
  - in generic programming, 170
  - multilevel pointers, 26
  - naming conventions, 88, 229
  - placement of const qualifier, 21–22
  - rebind convention for allocators, 180
  - and the STL, 12
  - STL (standard template library), 11
  - traits and promulgation, 193, 197
  - unnecessary static, 232
  - unnecessary virtual, 232
- Conversions, auto\_ptr, 147–148

Copying. *See also* assignment;  
 cloning; initialization.  
 address of non-const to  
 pointer to const, 22  
 assignment, 45–47  
 class objects, 38  
 construction, 45–47  
 objects, 38  
 Covariance, 174  
 Cranberries, xviii, 183  
 Cross-cast, 94–95

## D

Dash angle bracket asterisk  
 (->\*), pointer-to-member  
 operator, 58  
 Data abstraction, 1–2  
 Data hiding. *See* access protec-  
 tion; data abstraction.  
 Deallocating arrays, 127–129  
 Decay  
 arrays, 17, 25  
 functions, 17, 72  
 Declaring function pointers, 49  
 Delta adjustment of class  
 pointer, 58, 98, 108  
 Design patterns  
 description, 7–10  
 essential parts, 8–10  
 Factory Method, 103–106,  
 108–109  
 microarchitectures, 10  
 Template Method  
*versus* C++ templates, 77  
 description, 77–79  
 wrappers, 8  
 Destruction  
 assignment, 42–43  
 order, 141–142  
 RAII, 140  
 restricting heap allocation,  
 117  
 Dewhurst, David R., xviii

Disambiguation  
 auto keyword, 233  
 register keyword, 233  
 with template, 179–181  
 templates, 179–181  
 with typename, 169–172  
 “Don’t call us, we’ll call  
 you,” 68, 79  
 Dot asterisk (\*), pointer-to-  
 member operator, 58  
 Down casting. *See also* cast  
 operators.  
 runtime type information, 31  
 safe, 31  
 dynamic\_cast operator, 31–32

## E

e, 65, 66  
 Element type, determining,  
 189–191  
 Embedded type information,  
 189–191  
 Exception safety  
 axioms, 131–133  
 catching, 135–137  
 constructors, 143–144  
 copy assignment, 46  
 exceptions, 143–144  
 functions, 135–137  
 new operator, 143–144  
 safe destruction, 132  
 swap throwing, 133  
 synchronous exceptions,  
 131–132  
 throwing, 135–137  
 Exceptions, memory allocation,  
 143–144  
 exit and destruction, 117, 140  
 Explicit specialization, 155–159,  
 183–187  
 Expressions  
 const qualifiers,  
 adding/removing, 29–30

volatile qualifiers,  
 adding/removing, 29–30

## F

*Fahrenheit* 451 (veiled  
 reference), 50  
 Fehér, Attila, xvii  
 File handles, resource control.  
*See* RAII.  
 Fire safety advice, 50  
 Framework code, callbacks, 68  
 Function declarators, pointers  
 to, 61–62  
 Function objects. *See also*  
 function pointers;  
 smart pointers.  
 as callbacks, 67–70. *See also*  
 Command pattern.  
 description, 63–66  
 integrating with member  
 functions, 66  
 Function overloading. *See also*  
 operator overloading.  
 function templates, 213  
 generic algorithms, 223  
 infix calls, 90  
 overloaded index operator,  
 35–36  
 pointers to overloaded  
 functions, 51  
 required knowledge, xiii  
 scope, 88  
 SFINAE, 217  
 taking address of, 51  
 Function pointers. *See also*  
 function objects.  
 callbacks, 50–51  
 declaring, 49  
 description, 49–51  
 generic, 49–50  
 to inline functions, 50  
 to non-static member  
 functions, 50

Function pointers, *continued*  
 to overloaded functions, 51  
 virtual, 64–66

Function style, 29

Function templates  
 generic algorithms, 221–224  
*versus* nontemplate functions,  
 213  
 overloading, 213–215  
 template argument  
 deduction, 209–212

Functionoid. *See* function  
 objects.

Functions  
 arguments from arrays. *See*  
 array formal arguments.  
 decay, 17, 72  
 multilevel pointers, 25, 27  
 references to, 62  
 selecting right version, 214  
 static linkages, avoiding, 85

Functor. *See* function objects.

## G

Generic function pointers,  
 49–50

Gentleman Cambrioleur, 196

get/set interfaces, 1

Global scope, namespaces,  
 81–85

Goldfedder, Brandon, xvii

Gordon, Peter, xvii

Graphical shapes, resource  
 control. *See* RAII.

Guarding against multiple  
 inclusions, 229–230

## H

Handle/body idiom, 117

Handle classes  
 exception safe swaps, 46  
 RAII, 139

restricting heap allocation,  
 117

Header files, multiple inclusion,  
 229–230

Heap  
 algorithms, 155  
 allocation, restricting,  
 117–118  
 class template explicit  
 specialization, 155–159

Heinlein, Robert (veiled  
 reference), 76

Henney, Kevlin, xvii

Hewins, Sarah G., xviii

Hitchhiker’s Guide to the  
 Galaxy, (veiled reference),  
 211, 213

Hollywood principle, 68, 79,  
 224

## I

Idioms  
 assumption of non-throwing  
 deletion, 136  
 checking for assignment to  
 self, 47  
 computational constructor,  
 43  
 to create abstract base class,  
 113–115  
 exception safety axioms,  
 131–133  
 function object, 63  
 getting current `new_handler`,  
 51  
 intent of `dynamic_cast` to ref-  
 erence, 32  
 meaning of assignment, 46  
 ordering code for exception  
 safety, 136  
 partitioning code to avoid  
 RTTI, 94  
 to prohibit copying, 111

RAII, 139–142  
 reference to pointer formal  
 argument, 26  
 resource acquisition is  
 initialization, 139–142  
 to restrict heap allocation,  
 117–118  
 result of assignment *vs.*  
 initialization, 46  
 robust computation of  
 array size, 17  
 smart pointer, 145–147  
 STL function object, 72–73  
 violation of copy operation  
 idioms, 148  
 virtual constructor, 100

Ids, templates, 154

Ignorance  
 callbacks, 67  
 healthful aspects, 5, 12, 224  
 Java programmers, 37  
 of object type, 100–101  
 pitfalls of, xii  
 and templates, 180

Initialization. *See also*  
 assignment; copying.  
 argument passing, 41  
*versus* assignment, 41–43  
 catching exceptions, 41  
 declaration, 41  
 function return, 41

Initialization order. *See*  
 construction order.

Inline functions, pointers to, 50

Instantiation  
 template member functions,  
 225–227  
 templates, 154

Integers as pointers  
 new cast operators, 29, 31  
 placement new, 119  
 pointer arithmetic, 151

Interface class, 65, 93

**J**

Java  
*versus* C++, xii  
 function and array  
   declarators, 62  
 good natured poke at, 37  
 interface classes, 93  
 member function lookup, 88  
 Johnson, Matthew, xvii  
 Josuttis, Nicolai, 217–220

**K**

Kamel, Moataz, xvii  
 Keywords, optional, 231–233  
 Koenig lookup. *See* ADL (argument dependent lookup).  
 Kyu Sakamoto, reference to, 42, 68, 70

**L**

Lazy evaluation, 34  
 Lippman, Stan, 141  
 Logically const, 35  
 Login sessions, resource control.  
   *See* RAII.  
 Lupin, Arsene, 196

**M**

Managers, gratuitous swipe at, 10  
 Martin, Steve (veiled reference), 46  
 McFerrin, Bobby (veiled reference), 179  
 Member functions  
   function matching errors, 87–88  
   integrating function objects, 66  
   lookup, 87–88  
   pointers to  
     contravariance, 54–55  
     declaration syntax, 57–58

integrating with function objects, 66  
 operator precedence, 58, 61  
*versus* pointers, 57–59  
 simple pointer to function, 58  
 virtualness, 58  
 roles, 77–78  
 templates, 173–177  
 Member object. *See* class layout.  
 Member offset. *See* class layout.  
 Member operator functions,  
   overloading non-member operators, 91–92  
 Member specialization, 165  
 Member templates, 173–177  
 Memory  
   class-specific management, 123–126  
   heap allocation, restricting, 117–118  
   resource control. *See* RAII.  
 Memory management, arrays, 127–129  
 Meyers, Scott, xvii, 1  
 Multidimensional arrays  
   array formal arguments, 18–19  
   function and array declarators, 61  
   pointer arithmetic, 150

**N**

Names, templates, 154  
 Namespaces  
   aliases, 84  
   anonymous, 84–85  
   continual explicit qualification, 82–83  
   description, 81–85  
   names  
     declaring, 82  
     importing, 83  
   using declarations, 84  
   using directives, 82–83

Nested names, templates, 179  
 Network connections, resource control. *See* RAII.  
 New cast operations. *See* cast operators, new style.  
 new operator  
   description, 143–144  
   exception safety, 143–144  
   *versus* operator new, 119, 123, 143  
   operator overloading, 143–144  
 New style cast operators, *versus* old-style, 29  
 1984 (veiled reference), 224  
 Non-static member functions, pointers to, 50  
 Nontemplate functions *versus* function templates, 213

**O**

Objects  
   alternate names for. *See* aliases; references.  
   arrays of, 120–121  
   capability queries, 93–95  
   changing logical state, 33–36  
   cloning, 99–101  
   copying, 38  
   copying, prohibiting, 111  
   creating, based on existing objects, 103–106  
   heap allocation, restricting, 117–118  
   integrating member functions with function objects, 66  
   lazy evaluation of values, 34  
   managing with RAII, 139–142  
   modifying, 34  
   with multiple addresses. *See* pointer comparison.  
   with multiple types. *See* polymorphism.

- Objects, *continued*
    - polymorphic, 3–5
    - structure and layout, 37–39
    - traits, 193–197
    - type constraints, 111
    - virtual constructors, 99–101
  - Offset. *See* delta adjustment of
    - class pointer; pointers to, data members.
  - Old-style cast operators, *versus* new style, 29
  - operator delete
    - class-specific memory management, 123–126
    - usual version, 125
  - Operator function lookup, 91–92
  - operator new
    - class-specific memory management, 123–126
    - versus* new operator, 119, 123, 143
    - placement new, 119–121
    - usual version, 125
  - Operator overloading. *See also* function overloading.
    - `auto_ptr`, 147
    - constructors, 143–144
    - exception safety, 131
    - exceptions, 143–144
    - function calls, 91–92
    - function objects, 63
    - infix calls, 90–92
    - new operator, 143–144
    - operator function lookup, 91–92
    - versus* overriding, 75
    - placement new, 119
    - pointer arithmetic, 149–151
    - policies, 206
    - smart pointers, 145–146
    - STL function objects, 72
    - STL (standard template library), 11
  - Operator precedence, pointers
    - to member functions, 58, 61
  - Orwell, George (veiled reference), 224
  - Ottosen, Thorsten, xvii
  - Overloading
    - > operator, 145–146, 147–148
    - \* (asterisk) operator, 145–146, 147–148
    - as communication, with other programmers, 214–215
    - function call operators, 63
    - function templates, 213–215, 217–220
    - functions. *See* function overloading.
    - index operator, 35–36
    - operators. *See* operator overloading.
    - operators, policies, 206
    - versus* overriding, 75–76
  - Overriding
    - functions, covariant return types, 107–109
    - versus* overloading, 75–76
- P**
- Parameters, templates, 153–154
  - Parentheses (())
    - function call operator, 58
    - grouping in declarations, 61
    - grouping with pointer-to-member operators, 58
  - Partial specialization, 161, 183–187, 197
  - Patterns. *See* design patterns.
  - Paul Revere and the Raiders, 143
  - Pointer arithmetic, 19, 149–151
  - Pointer comparison, 97–98
  - Pointers. *See also* smart pointers.
    - arrays of, 25
    - dereferencing, 53–54
    - integers as, 151
    - list iterators, 151
    - losing type information, 98
    - managing buffers of, 25
    - multilevel. *See* pointers to, pointers.
    - versus* references, 13
    - stacks of, 185–187
    - subtracting, 151
  - Pointers to
    - array declarators, 61–62
    - characters, 156
    - class members, *versus* pointers, 53–56
    - const
      - versus* const pointers, 21–23
      - converting to pointer to non-const, 23
    - data members, 54–55
    - function declarators, 61–62
    - functions. *See* callbacks; Command pattern; function pointers.
    - member functions
      - contravariance, 54–55
      - declaration syntax, 57–58
      - integrating with function objects, 66
      - operator precedence, 58, 61
      - versus* pointers, 57–59
      - simple pointer to function, 58
      - virtualness, 58
    - non-const, converting to pointer to const, 22–23
    - pointers
      - changing pointer values, 26
      - conversions, 26–27
      - converting to pointer to const, 27

converting to pointer to non-const, 27  
 description, 25–27  
 managing buffers of pointers, 25–26  
 void, 98  
 Policies, 205–208  
 Polymorphic base classes, 3–5  
 Polymorphism, 3–5  
 Predicates, STL function objects as, 73–74  
 Primary templates. *See also* STL (standard template library); templates.  
 explicit specialization, 155–158, 183–187  
 instantiation, 154  
 member specialization, 165  
 partial specialization, 161, 183–187  
 SFINAE, 218  
 specialization, 154  
 specializing for type information, 183  
 Promulgation, conventions, 193, 197  
 Prototype, 99–101

## Q

QWAN (Quality Without A Name), 90

## R

RAI, 139–142. *See also* `auto_ptr`.  
 Rebind convention for allocators, 180  
 Reeves, Jack, xviii  
 References. *See also* aliases.  
 to arrays, 62  
 to const, 15  
 description, 13–15  
 to functions, 62

initialization, 14–15  
 to non-const, 15  
 null, 13–14  
*versus* pointers, 13  
`register` keyword, 233  
`reinterpret_cast` operator, 31  
 Resource acquisition is initialization. *See* RAI.  
 Resource control. *See* RAI.  
 RTTI (runtime type information)  
 for capability query, 95  
 incorrect use of, 103  
 runtime cost of, 31  
 for safe down cast, 31

## S

Sakamoto, Kyu (veiled reference), 42, 68, 70  
 Saks, Dan, xvii  
 Semaphores, resource control. *See* RAI.  
 SFINAE (substitution failure is not an error), 217–220  
 Single-dimensional arrays, array formal arguments, 17–18  
 Slettebø, Terje, xvii  
 Smart pointers. *See also* function objects; pointers.  
*versus* `auto_ptr`, 148  
 list iterators, 11, 151  
 operator overloading, 145–146  
 templates, 145–146  
 Social commentary, xiii, 7, 10, 71, 190, 195  
 Specialization  
 explicit, 183–187  
 partial, 183–187  
 SFINAE, 218  
 templates. *See* templates, specialization.  
 for type information, 183

Specializing for type information, 183  
 Standard template library (STL), 11–12. *See also* primary templates; templates.  
 Static linkages, avoiding, 85  
`static_cast` operator, 30–32  
 STL function objects  
 as comparators, 72–73  
 description, 71–74  
 as predicates, 73–74  
 true/false questions, 73–74  
 STL (standard template library), 11–12. *See also* primary templates; templates.  
 Subcontractor, derived class as, 4–5  
 Substitution failure is not an error (SFINAE), 217–220  
 Sutter, Herb, xi, xvii, xviii, 136  
 Swedish, and technical communication, 99

## T

Template argument deduction, 18, 209–212  
 Template Method  
*versus* C++ templates, 77  
 description, 77–79  
 Template template parameters, 199–204  
 Templates. *See also* primary templates; STL (standard template library).  
 arguments  
 customizing. *See* templates, specialization.  
 description, 153–154  
 array formal arguments, 18–19  
 C++ *versus* Template Method, 77

Templates, *continued*  
 customizing. *See* templates, specialization.  
 disambiguation, 169–172, 179–181  
 ids, 154  
 ignorance and, 180  
 instantiation, 154  
 member functions, 173–177  
 names, 154  
 nested names, 179  
 parameters, 153–154  
 smart pointers, 145–146  
 specialization  
   explicit, 155–159, 183–187, 209–212  
   partial, 161–164, 183–187  
   terminology, 153–154  
   traits, 197  
 traits, 195–197

Terminology  
 assignment *versus*  
   initialization, 41–43  
 const pointers *versus* pointers  
   to const, 21–23  
 member templates, 174  
 new operator *versus* opera-  
   tor new, 119, 123, 143  
 new style cast operators  
   *versus* old-style, 29  
 overloading *versus*  
   overriding, 75–76  
 pointers to class members  
   *versus* pointers, 53–56  
 pointers to const *versus* const  
   pointers, 21–23

pointers to member functions  
   *versus* pointers, 57–59  
 references *versus* pointers, 13  
 template argument  
   deduction, 209  
 Template Method *versus* C++  
   templates, 77  
 templates, 153–154  
 wrappers, 8

Tondo, Clovis, xvii

Traits  
 conventions, 193, 197  
 description, 193–197  
 specialization, 197  
 templates, 195–197

Traits class, 193–197

True/false questions, 73–74

Type  
 container elements,  
   determining, 189–191  
 information, embedded,  
   189–191  
 information about, 193–197  
 qualifiers, changing, 29–30,  
   30–31  
 traits, 193–197

Typename, disambiguation,  
 169–172

**U**

Unnecessary static conventions,  
 232

Unnecessary virtual  
 conventions, 232

User-defined types, assignment,  
 41–43

Using declarations, 84  
 Using directives, 82–83  
 Usual operator new and  
   operator delete, 125

## V

Vandevoorde, David, 217–220

Variables, avoiding static  
 linkage, 85

Virtual constructors, 99–101

Virtual copy, assignment, 47

Virtual function pointers, 64–66

Virtual functions, class layout,  
 37

Virtual inheritance, class layout,  
 37–38

virtual keyword, 231–233

volatile qualifiers,  
 adding/removing, 29–30

## W

Ward, Dick and Judy, xviii

What you see is what you get, 37

Wilson, Flip (veiled reference),  
 37

Wilson, Matthew, xvii

## Z

Zolman, Leor, xvii

Zoo animals, resource control.  
*See* RAII.

# Index of Code Examples

See also Index, page 237

## A

ABC class, 113–114  
Abstract base class  
  ABC class, 113–114  
  Action class, 69  
  Func class, 65  
  Rollable class, 93  
Access games  
  aFunc function, 118  
  NoCopy class, 111  
  NoHeap class, 117–118  
  OnHeap class, 118  
Action class, 69  
aFunc function, 82–84, 89, 118  
Allocator  
  AnAlloc class template, 179  
  AnAlloc::rebind member  
  template, 179  
  SList class template, 180  
AnAlloc class template, 179  
AnAlloc::rebind member  
  template, 179  
App class, 78  
append function, 120–121  
App::startup member  
  function, 78  
Argument dependent lookup,  
  89  
Array class template, 225  
Array<Circle, 7> explicit  
  instantiation, 227  
ArrayDeletePolicy class  
  template, 207

Array<T, n>::operator ==  
  template member function,  
  226  
Assignment  
  SList<T>::operator =  
  member template, 175  
  String::operator =  
  member function, 42, 135  
aTemplateContext function  
  template, 132

## B

B class, 59, 75, 87  
begForgiveness function, 51  
Blob class, 231  
Button class, 67, 69  
Button::setAction member  
  function, 136

## C

C class, 54  
Callback  
  Action class, 69  
  begForgiveness function,  
  51  
CanConvert class template,  
  220  
Capability class, 93  
cast function template, 209  
CheckedPtr class template,  
  145

Circle class  
  capability queries, 94  
  covariant return types,  
  107–108  
  pointers to class members, 55  
  pointers to member  
  functions, 57  
CircleEditor class, 108  
Class templates  
  AnAlloc, 179  
  Array, 225  
  ArrayDeletePolicy, 207  
  CanConvert, 220  
  CheckedPtr, 145  
  ContainerTraits, 194  
  ContainerTraits<  
  vector<T> >, 197  
  ContainerTraits<const  
  T \*>, 197  
  ContainerTraits<T \*>,  
  196  
  Heap, 155, 165  
  Heap<T \*>, 161  
  IsArray, 187  
  IsClass, 219  
  IsInt, 183  
  IsPCM, 187  
  IsPtr, 184  
  MFunc, 66  
  NoDeletePolicy, 207  
  PFunc1, 215  
  PFunc2, 212, 214  
  PtrCmp, 162

Class templates, *continued*

PtrDeletePolicy, 206–207  
 PtrList, 169  
 PtrVector, 25  
 ReadonlySeq, 191  
 SCollection, 170  
 Seq, 189–190  
 SList, 173, 180  
 Stack, 185–186, 200–202,  
 205–206  
 Wrapper1, 203  
 Wrapper2, 203  
 Wrapper3, 203–204

## Classes

ABC, 113–114  
 Action, 69  
 App, 78  
 B, 59, 75, 87  
 Blob, 231  
 Button, 67, 69  
 C, 54  
 Circle  
   capability queries, 94  
   covariant return types,  
   107–108  
   pointers to class members,  
   55  
   pointers to member  
   functions, 57  
 CircleEditor, 108  
 ContainerTraits<const  
   char \*>, 196  
 ContainerTraits  
   <ForeignContainer>,  
   195  
 D, 59, 76, 87  
 E, 88  
 Employee, 104–105  
 Fib, 63  
 ForeignContainer, 195  
 Func, 65  
 Handle  
   array allocation, 127–128  
   class-specific memory  
   management, 123–124

  copy operations, 45  
   optional keywords, 232  
   restricting heap allocation,  
   118  
 Heap<char \*>, 157–158  
 Heap<const char \*>, 156  
 IsWarm, 73  
 Meal, 100  
 MyApp, 78–79  
 MyContainer, 170  
 MyHandle, 124  
 NMFunc, 66  
 NoCopy, 111  
 NoHeap, 117–118  
 ObservedBlob, 97  
 OnHeap, 118  
 PlayMusic, 69  
 PopLess, 72  
 rep, 125  
 ResourceHandle, 139  
 Rollable, 93  
 S, 38  
 Shape  
   capability queries, 93  
   covariant return types,  
   107–108  
   optional kwds, 231  
   pointer comparison, 97  
   pointers to class members,  
   55  
   pointers to member  
   functions, 57  
 ShapeEditor, 108  
 SharpBlob, 231  
 Spaghetti, 100  
 Square, 94  
 State, 71, 222–223  
 String, 41  
 Subject, 97  
 T, 39  
 Temp, 105  
 Trace, 141  
 Wheel, 94  
 X, 33–36, 91  
 cleanupBuf function, 121

Command pattern, 69

## Comparator

  PopLess class, 72  
   popLess function, 71  
   PtrCmp class template, 162  
   strLess function, 157

Computational constructor, 43

ContainerTraits class  
   template, 194

ContainerTraits  
   < vector<T> > class  
   template, 197

ContainerTraits<const  
   char \*> class, 196

ContainerTraits<const T  
   \*> class template, 197

ContainerTraits  
   <ForeignContainer>  
   class, 195

ContainerTraits<T \*> class  
   template, 196

## Copy assignment

  Handle::operator =  
   member function, 46–47  
 Handle::swap member  
   function, 46

Covariant return, 107–108

**D**

D class, 59, 76, 87

**E**

E class, 88

Employee class, 104–105

## Exceptions

  aTemplateContext  
   function template, 132  
 Button::setAction  
   member function, 136  
 f function, 140–141  
 ResourceHandle class, 139  
 String::operator =  
   member function, 135

Trace class, 141  
 X: :X member function, 133  
 Explicit instantiation  
   Array<Circle, 7>, 227  
   Heap<double>, 167  
 Explicit specialization  
   ContainerTraits<const char \*> class, 196  
   ContainerTraits<ForeignContainer> class, 195  
   Heap class template, 155  
   Heap<char \*> class, 157–158  
   Heap<const char \*> class, 156  
   IsInt class template, 183  
 extractHeap function template, 158

## F

f function, 140–141  
 Factory Method  
   Circle class, 108  
   Employee class, 104–105  
   Shape class, 108  
   Temp class, 105  
 Fib class, 63  
 fibonacci function, 64  
 Fib::operator () member function, 63  
 fill function template, 170, 172  
 ForeignContainer class, 195  
 friend function, 43  
 Func class, 65  
 Function object  
   Action class, 69  
   Fib class, 63  
   Func class, 65  
   IsWarm class, 73  
   MFunc class template, 66  
   NFunc class, 66  
   PFunc1 class template, 215

PFunc2 class template, 212, 214  
 PlayMusic class, 69  
 PopLess class, 72  
 PtrCmp class template, 162  
 Function template overloading  
   g function, 213  
   g function template, 213  
   makePFunc function template, 215  
 Function templates  
   aTemplateContext, 132  
   cast, 209  
   extractHeap, 158  
   fill, 170, 172  
   g, 213  
   makePFunc, 212, 214–215  
   min, 209  
   process, 19, 189, 191, 194  
   process\_2d, 19  
   repeat, 211  
   set\_2d, 15  
   slowSort, 221–224  
   swap, 14, 45  
   zeroOut, 211  
 Functions  
   aFunc, 82–84, 89, 118  
   append, 120–121  
   begForgiveness, 51  
   cleanupBuf, 121  
   f, 140–141  
   fibonacci, 64  
   g, 213  
   genInfo, 104  
   integrate, 65  
   operator new, 119  
   org\_semantics::operator +, 82  
   popLess, 71  
   scanTo, 26  
   someFunc, 115  
   String::operator +, 43  
   strLess, 157  
   swap, 222

## G

g function, 213  
 g function template, 213  
 Generic algorithms, 221–224  
 genInfo function, 104

## H

Handle class  
   array allocation, 127–128  
   class-specific memory management, 123–124  
   copy operations, 45  
   optional keywords, 232  
   restricting heap allocation, 118  
 Handle::operator = member function, 46–47  
 Handle::operator delete member function, 126  
 Handle::operator new member function, 125  
 Handle::swap member function, 46  
 hasIterator preprocessor macro, 219  
 Heap class template, 155, 165  
 Heap<char \*> class, 157–158  
 Heap<const char \*> class, 156  
 Heap<const char \*>::pop member function, 166  
 Heap<const char \*>::push member function, 157, 166–167  
 Heap<double> explicit instantiation, 167  
 Heap<T \*> class template, 161  
 Heap<T \*>::push template member function, 162  
 Heap<T>::pop template member function, 156  
 Heap<T>::push template member function, 155  
 Helper function, 212, 214

**I**

integrate function, 65  
 Interface class  
   Action class, 69  
   Func class, 65  
   Rollable class, 93  
 IsArray class template, 187  
 IsClass class template, 219  
 IsInt class template, 183  
 IsPCM class template, 187  
 IsPtr class template, 184  
 is\_ptr preprocessor macro, 217  
 IsWarm class, 73

**M**

makePFun function template, 212, 214–215  
 Meal class, 100  
 Member array new, 118, 128  
 Member delete, 126  
 Member functions  
   App::startup, 78  
   Button::setAction, 136  
   Fib::operator (), 63  
   Handle::operator =, 46–47  
   Handle::operator delete, 126  
   Handle::operator new, 125  
   Handle::swap, 46  
   Heap<const char \*>::pop, 166  
   Heap<const char \*>::push, 157, 166–167  
   String::operator =, 42, 135  
   String::String, 42  
   String::String, 42–43  
   X::getValue, 34–35  
   X::memFunc2, 91  
   X::X, 133

Member new  
   Handle class, 123–124  
   Handle::operator new member function, 125  
   MyHandle class, 124  
 Member specialization  
   Heap<const char \*>::pop member function, 166  
   Heap<const char \*>::push member function, 157, 166–167  
 Member templates  
   AnAlloc::rebind, 179  
   SList<T>::operator =, 175  
   SList<T>::SList, 174–175  
   SList<T>::sort, 176  
 MFunc class template, 66  
 minimum function template, 209  
 Multiple inheritance, 97  
 MyApp class, 78–79  
 MyContainer class, 170  
 MyHandle class, 124

**N**

Namespaces  
   aFunc function, 82–84  
   org\_semantics, 81, 89  
   org\_semantics namespace, 81  
   org\_semantics::operator + function, 82  
 NMFun class, 66  
 NoCopy class, 111  
 NoDeletePolicy class template, 207  
 NoHeap class, 117–118

**O**

ObservedBlob class, 97  
 OnHeap class, 118

operator new function, 119  
 org\_semantics namespace, 81, 89  
 org\_semantics::operator + function, 82

**P**

Partial specialization  
   ContainerTraits  
     < vector<T> > class template, 197  
   ContainerTraits<const T \*> class template, 197  
   ContainerTraits<T \*> class template, 196  
   Heap<T \*> class template, 161  
   Heap<T \*>::push template member function, 162  
   IsArray class template, 187  
   IsPCM class template, 187  
   IsPtr class template, 184  
 PFun1 class template, 215  
 PFun2 class template, 212, 214  
 Placement new  
   append function, 120–121  
   operator new function, 119  
 PlayMusic class, 69  
 POD, 38  
 Pointer arithmetic  
   process\_2d function template, 19  
   set\_2d function template, 15  
 Policy  
   ArrayDeletePolicy class template, 207  
   NoDeletePolicy class template, 207  
   PtrDeletePolicy class template, 206–207  
   Stack class template, 205–206  
 PopLess class, 72  
 popLess function, 71

Predicate, 73  
 Preprocessor macro  
   hasIterator, 219  
   is\_ptr, 217  
 process function template, 19,  
   189, 191, 194  
 process\_2d function  
   template, 19  
 Prototype  
   Action class, 69  
   Circle class, 107  
   Meal class, 100  
   PlayMusic class, 69  
   Shape class, 107  
   Spaghetti class, 100  
 PtrCmp class template, 162  
 PtrDeletePolicy class  
   template, 206–207  
 PtrList class template, 169  
 PtrVector class template, 25

## R

ReadOnlySeq class template,  
   191  
 rep class, 125  
 repeat function template, 211  
 ResourceHandle class, 139  
 Rollable class, 93

## S

S class, 38  
 scanTo function, 26  
 SCollection class template,  
   170  
 Seq class template, 189–190  
 set\_2d function template, 15  
 SFINAE  
   CanConvert class template,  
     220  
   hasIterator preprocessor  
     macro, 219  
   IsClass class template, 219  
   is\_ptr preprocessor macro,  
     217

Shape class  
   capability queries, 93  
   covariant return types,  
     107–108  
   optional kwds, 231  
   pointer comparison, 97  
   pointers to class members, 55  
   pointers to member  
     functions, 57  
 ShapeEditor class, 108  
 SharpBlob class, 231  
 SList class template, 173, 180  
 SList<T>: :empty template  
   member function, 173  
 SList<T>: :Node template  
   member class, 174  
 SList<T>: :operator =  
   member template, 175  
 SList<T>: :SList member  
   template, 174–175  
 SList<T>: :sort member  
   template, 176  
 slowSort function template,  
   221–224  
 Smart pointer, 145  
 someFunc function, 115  
 Spaghetti class, 100  
 Square class, 94  
 Stack class template, 185–186,  
   200–202, 205–206  
 Stack<T>: :push template  
   member function, 185  
 State class, 71, 222–223  
 String class, 41  
 String: :operator +  
   function, 43  
 String: :operator =  
   member function, 42, 135  
 String: :String member  
   function, 42  
 String: :String member  
   function, 42–43  
 strLess function, 157  
 Subject class, 97  
 swap function, 222  
 swap function template, 14, 45

## T

T class, 39  
 Temp class, 105  
 Template argument deduction  
   cast function template, 209  
   makePFun function template,  
     212, 214  
   minimum function template,  
     209  
   repeat function template,  
     211  
   zeroOut function template,  
     211  
 Template member class, 174  
 Template member functions  
   Array<T, n>: :operator  
     ==, 226  
   Heap<T \*>: :push, 162  
   Heap<T>: :pop, 156  
   Heap<T>: :push, 155  
   SList<T>: :empty, 173  
   Stack<T>: :push, 185  
 Template Method  
   App class, 78  
   App: :startup member  
     function, 78  
   MyApp class, 78–79  
 Trace class, 141  
 Traits  
   ContainerTraits class  
     template, 194  
   ContainerTraits  
     < vector<T> > class  
     template, 197  
   ContainerTraits<const  
     char \*> class, 196  
   ContainerTraits<const  
     T \*> class template, 197  
   ContainerTraits  
     <ForeignContainer>  
     class, 195  
   ContainerTraits<T \*>  
     class template, 196

## **W**

Wheel class, 94  
Wrapper1 class template, 203  
Wrapper2 class template, 203  
Wrapper3 class template,  
203–204

## **X**

X class, 33–36, 91  
X::getValue member func-  
tion, 34–35  
X::memFunc2 member  
function, 91  
X::X member function, 133

## **Z**

zeroOut function template,  
211