

BJARNE STROUSTRUP



Programming

Principles and Practice Using C++

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.

A complete list of photo sources and credits appears on pages 1235–1236.

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 the United States, please contact:

International Sales
international@pearsoned.com

Copyright © 2009 Pearson Education, Inc.

Stroustrup, Bjarne.

Programming principles and practice using C++ / Bjarne Stroustrup.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-321-54372-1 (pbk. : alk. paper) 1. C++ (Computer program language) I. Title.

QA76.73.C153S82 2008

005.13'3—dc22

2008032595

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
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-54372-1

ISBN-10: 0-321-54372-6

Text printed in the United States on recycled paper at Courier in Kendallville, Indiana.

First printing, December 2008

Preface

“Damn the torpedoes!
Full speed ahead.”

—Admiral Farragut

Programming is the art of expressing solutions to problems so that a computer can execute those solutions. Much of the effort in programming is spent finding and refining solutions. Often, a problem is only fully understood through the process of programming a solution for it.

This book is for someone who has never programmed before but is willing to work hard to learn. It helps you understand the principles and acquire the practical skills of programming using the C++ programming language. My aim is for you to gain sufficient knowledge and experience to perform simple useful programming tasks using the best up-to-date techniques. How long will that take? As part of a first-year university course, you can work through this book in a semester (assuming that you have a workload of four courses of average difficulty). If you work by yourself, don’t expect to spend less time than that (maybe 15 hours a week for 14 weeks).

Three months may seem a long time, but there’s a lot to learn and you’ll be writing your first simple programs after about an hour. Also, all learning is gradual: each chapter introduces new useful concepts and illustrates them with examples inspired by real-world uses. Your ability to express ideas in code – getting a computer to do what you want it to do – gradually and steadily increases as you go along. I never say, “Learn a month’s worth of theory and then see if you can use it.”

Why would you want to program? Our civilization runs on software. Without understanding software you are reduced to believing in “magic” and will be locked out of many of the most interesting, profitable, and socially useful technical fields of work. When I talk about programming, I think of the whole spectrum of computer programs from personal computer applications with GUIs (graphical user interfaces), through engineering calculations and embedded systems control applications (such as digital cameras, cars, and cell phones), to text manipulation applications as found in many humanities and business applications. Like mathematics, programming – when done well – is a valuable intellectual exercise that sharpens our ability to think. However, thanks to feedback from the computer, programming is more concrete than most forms of math, and therefore accessible to more people. It is a way to reach out and change the world – ideally for the better. Finally, programming can be great fun.

Why C++? You can’t learn to program without a programming language, and C++ directly supports the key concepts and techniques used in real-world software. C++ is one of the most widely used programming languages, found in an unsurpassed range of application areas. You find C++ applications everywhere from the bottom of the oceans to the surface of Mars. C++ is precisely and comprehensively defined by a nonproprietary international standard. Quality and/or free implementations are available on every kind of computer. Most of the programming concepts that you will learn using C++ can be used directly in other languages, such as C, C#, Fortran, and Java. Finally, I simply like C++ as a language for writing elegant and efficient code.

This is not the easiest book on beginning programming; it is not meant to be. I just aim for it to be the easiest book from which you can learn the basics of real-world programming. That’s quite an ambitious goal because much modern software relies on techniques considered advanced just a few years ago.

My fundamental assumption is that you want to write programs for the use of others, and to do so responsibly, providing a decent level of system quality; that is, I assume that you want to achieve a level of professionalism. Consequently, I chose the topics for this book to cover what is needed to get started with real-world programming, not just what is easy to teach and learn. If you need a technique to get basic work done right, I describe it, demonstrate concepts and language facilities needed to support the technique, provide exercises for it, and expect you to work on those exercises. If you just want to understand toy programs, you can get along with far less than I present. On the other hand, I won’t waste your time with material of marginal practical importance. If an idea is explained here, it’s because you’ll almost certainly need it.

If your desire is to use the work of others without understanding how things are done and without adding significantly to the code yourself, this book is not for you. If so, please consider whether you would be better served by another book and another language. If that is approximately your view of programming, please also consider from where you got that view and whether it in fact is adequate for your needs. People often underestimate the complexity of program-

ming as well as its value. I would hate for you to acquire a dislike for programming because of a mismatch between what you need and the part of the software reality I describe. There are many parts of the “information technology” world that do not require knowledge of programming. This book is aimed to serve those who do want to write or understand nontrivial programs.

Because of its structure and practical aims, this book can also be used as a second book on programming for someone who already knows a bit of C++ or for someone who programs in another language and wants to learn C++. If you fit into one of those categories, I refrain from guessing how long it will take you to read this book, but I do encourage you to do many of the exercises. This will help you to counteract the common problem of writing programs in older, familiar styles rather than adopting newer techniques where these are more appropriate. If you have learned C++ in one of the more traditional ways, you’ll find something surprising and useful before you reach Chapter 7. Unless your name is Stroustrup, what I discuss here is not “your father’s C++.”

Programming is learned by writing programs. In this, programming is similar to other endeavors with a practical component. You cannot learn to swim, to play a musical instrument, or to drive a car just from reading a book – you must practice. Nor can you learn to program without reading and writing lots of code. This book focuses on code examples closely tied to explanatory text and diagrams. You need those to understand the ideals, concepts, and principles of programming and to master the language constructs used to express them. That’s essential, but by itself, it will not give you the practical skills of programming. For that, you need to do the exercises and get used to the tools for writing, compiling, and running programs. You need to make your own mistakes and learn to correct them. There is no substitute for writing code. Besides, that’s where the fun is!

On the other hand, there is more to programming – much more – than following a few rules and reading the manual. This book is emphatically not focused on “the syntax of C++.” Understanding the fundamental ideals, principles, and techniques is the essence of a good programmer. Only well-designed code has a chance of becoming part of a correct, reliable, and maintainable system. Also, “the fundamentals” are what last: they will still be essential after today’s languages and tools have evolved or been replaced.

What about computer science, software engineering, information technology, etc.? Is that all programming? Of course not! Programming is one of the fundamental topics that underlie everything in computer-related fields, and it has a natural place in a balanced course of computer science. I provide brief introductions to key concepts and techniques of algorithms, data structures, user interfaces, data processing, and software engineering. However, this book is not a substitute for a thorough and balanced study of those topics.

Code can be beautiful as well as useful. This book is written to help you see that, to understand what it means for code to be beautiful, and to help you to master the principles and acquire the practical skills to create such code. Good luck with programming!

A note to students

Of the 1000+ first-year students we have taught so far using drafts of this book at Texas A&M University, about 60% had programmed before and about 40% had never seen a line of code in their lives. Most succeeded, so you can do it, too.

You don't have to read this book as part of a course. I assume that the book will be widely used for self-study. However, whether you work your way through as part of a course or independently, try to work with others. Programming has an – unfair – reputation as a lonely activity. Most people work better and learn faster when they are part of a group with a common aim. Learning together and discussing problems with friends is not cheating! It is the most efficient – as well as most pleasant – way of making progress. If nothing else, working with friends forces you to articulate your ideas, which is just about the most efficient way of testing your understanding and making sure you remember. You don't actually have to personally discover the answer to every obscure language and programming environment problem. However, please don't cheat yourself by not doing the drills and a fair number of exercises (even if no teacher forces you to do them). Remember: programming is (among other things) a practical skill that you need to practice to master. If you don't write code (do several exercises for each chapter), reading this book will be a pointless theoretical exercise.

Most students – especially thoughtful good students – face times when they wonder whether their hard work is worthwhile. When (not if) this happens to you, take a break, reread the preface, and look at Chapter 1 (“Computers, People, and Programming”) and Chapter 22 (“Ideals and History”). There, I try to articulate what I find exciting about programming and why I consider it a crucial tool for making a positive contribution to the world. If you wonder about my teaching philosophy and general approach, have a look at Chapter 0 (“Notes to the Reader”).

You might find the weight of this book worrying, but it should reassure you that part of the reason for the heft is that I prefer to repeat an explanation or add an example rather than have you search for the one and only explanation. The other major part of the reason is that the second half of the book is reference material and “additional material” presented for you to explore only if you are interested in more information about a specific area of programming, such as embedded systems programming, text analysis, or numerical computation.

And please don't be too impatient. Learning any major new and valuable skill takes time and is worth it.

A note to teachers

No. This is not a traditional Computer Science 101 course. It is a book about how to construct working software. As such, it leaves out much of what a computer science student is traditionally exposed to (Turing completeness, state ma-

chines, discrete math, Chomsky grammars, etc.). Even hardware is ignored on the assumption that students have used computers in various ways since kindergarten. This book does not even try to mention most important CS topics. It is about programming (or more generally about how to develop software), and as such it goes into more detail about fewer topics than many traditional courses. It tries to do just one thing well, and computer science is not a one-course topic. If this book/course is used as part of a computer science, computer engineering, electrical engineering (many of our first students were EE majors), information science, or whatever program, I expect it to be taught alongside other courses as part of a well-rounded introduction.

Please read Chapter 0 (“Notes to the Reader”) for an explanation of my teaching philosophy, general approach, etc. Please try to convey those ideas to your students along the way.

Support

The book’s support website, www.stroustrup.com/Programming, contains a variety of materials supporting the teaching and learning of programming using this book. The material is likely to be improved with time, but for starters, you can find:

- Slides for lectures based on the book
- An instructor’s guide
- Header files and implementations of libraries used in the book
- Code for examples in the book
- Solutions to selected exercises
- Potentially useful links
- Errata

Suggestions for improvements are always welcome.

Acknowledgments

I’d especially like to thank my late colleague and co-teacher Lawrence “Pete” Petersen for encouraging me to tackle the task of teaching beginners long before I’d otherwise have felt comfortable doing that, and for supplying the practical teaching experience to make the course succeed. Without him, the first version of the course would have been a failure. We worked together on the first versions of the course for which this book was designed and together taught it repeatedly, learning from our experiences, improving the course and the book. My use of “we” in this book initially meant “Pete and me.”

Thanks to the students, teaching assistants, and peer teachers of ENGR 112 at Texas A&M University who directly and indirectly helped us construct this book, and to Walter Daugherty, who has also taught the course. Also thanks to Damian Dechev, Tracy Hammond, Arne Tolstrup Madsen, Gabriel Dos Reis, Nicholas Stroustrup, J. C. van Winkel, Greg Versoonder, Ronnie Ward, and Leor Zolman for constructive comments on drafts of this book. Thanks to Mogens Hansen for explaining about engine control software. Thanks to Al Aho, Stephen Edwards, Brian Kernighan, and Daisy Nguyen for helping me hide away from distractions to get writing done during the summers.

Thanks to the reviewers that Addison-Wesley found for me. Their comments, mostly based on teaching either C++ or Computer Science 101 at the college level, have been invaluable: Richard Enbody, David Gustafson, Ron McCarty, and K. Narayanaswamy. Also thanks to my editor, Peter Gordon, for many useful comments and (not least) for his patience. I'm very grateful to the production team assembled by Addison-Wesley; they added much to the quality of the book: Julie Grady (proofreader), Chris Keane (compositor), Rob Mauhar (illustrator), Julie Nahil (production editor), and Barbara Wood (copy editor).

In addition to my own unsystematic code checking, Bashar Anabtawi, Yinan Fan, and Yuriy Solodkyy checked all code fragments using Microsoft C++ 7.1 (2003) and 8.0 (2005) and GCC 3.4.4.

I would also like to thank Brian Kernighan and Doug McIlroy for setting a very high standard for writing about programming, and Dennis Ritchie and Kristen Nygaard for providing valuable lessons in practical language design.



Notes to the Reader

“When the terrain disagrees with the map,
trust the terrain.”

—Swiss army proverb

This chapter is a grab bag of information; it aims to give you an idea of what to expect from the rest of the book. Please skim through it and read what you find interesting. A teacher will find most parts immediately useful. If you are reading this book without the benefit of a good teacher, please don't try to read and understand everything in this chapter; just look at “The structure of this book” and the first part of the “A philosophy of teaching and learning” sections. You may want to return and reread this chapter once you feel comfortable writing and executing small programs.

0.1 The structure of this book

- 0.1.1 General approach
- 0.1.2 Drills, exercises, etc.
- 0.1.3 What comes after this book?

0.2 A philosophy of teaching and learning

- 0.2.1 The order of topics
- 0.2.2 Programming and programming language
- 0.2.3 Portability

0.3 Programming and computer science**0.4 Creativity and problem solving****0.5 Request for feedback****0.6 References****0.7 Biographies**

0.1 The structure of this book

This book consists of four parts and a collection of appendices:

- *Part I, “The Basics,”* presents the fundamental concepts and techniques of programming together with the C++ language and library facilities needed to get started writing code. This includes the type system, arithmetic operations, control structures, error handling, and the design, implementation, and use of functions and user-defined types.
- *Part II, “Input and Output,”* describes how to get numeric and text data from the keyboard and from files, and how to produce corresponding output to the screen and to files. Then, it shows how to present numeric data, text, and geometric shapes as graphical output, and how to get input into a program from a graphical user interface (GUI).
- *Part III, “Data and Algorithms,”* focuses on the C++ standard library’s containers and algorithms framework (the STL, standard template library). It shows how containers (such as **vector**, **list**, and **map**) are implemented (using pointers, arrays, dynamic memory, exceptions, and templates) and used. It also demonstrates the design and use of standard library algorithms (such as **sort**, **find**, and **inner_product**).
- *Part IV, “Broadening the View,”* offers a perspective on programming through a discussion of ideals and history, through examples (such as matrix computation, text manipulation, testing, and embedded systems programming), and through a brief description of the C language.
- *Appendices* provide useful information that doesn’t fit into a tutorial presentation, such as surveys of C++ language and standard library facilities, and descriptions of how to get started with an integrated development environment (IDE) and a graphical user interface (GUI) library.

Unfortunately, the world of programming doesn't really fall into four cleanly separated parts. Therefore, the "parts" of this book provide only a coarse classification of topics. We consider it a useful classification (obviously, or we wouldn't have used it), but reality has a way of escaping neat classifications. For example, we need to use input operations far sooner than we can give a thorough explanation of C++ standard I/O streams (input/output streams). Where the set of topics needed to present an idea conflicts with the overall classification, we explain the minimum needed for a good presentation, rather than just referring to the complete explanation elsewhere. Rigid classifications work much better for manuals than for tutorials.

The order of topics is determined by programming techniques, rather than programming language features; see §0.2. For a presentation organized around language features, see Appendix A.

To ease review and to help you if you miss a key point during a first reading where you have yet to discover which kind of information is crucial, we place three kinds of "alert markers" in the margin:

- Blue: concepts and techniques (this paragraph is an example of that)
- Green: advice
- Red: warning

0.1.1 General approach

In this book, we address you directly. That is simpler and clearer than the conventional "professional" indirect form of address, as found in most scientific papers. By "you" we mean "you, the reader," and by "we" we refer either to "ourselves, the author and teachers," or to you and us working together through a problem, as we might have done had we been in the same room.


This book is designed to be read chapter by chapter from the beginning to the end. Often, you'll want to go back to look at something a second or a third time. In fact, that's the only sensible approach, as you'll always dash past some details that you don't yet see the point in. In such cases, you'll eventually go back again. However, despite the index and the cross-references, this is not a book that you can open on any page and start reading with any expectation of success. Each section and each chapter assume understanding of what came before.

Each chapter is a reasonably self-contained unit, meant to be read in "one sitting" (logically, if not always feasible on a student's tight schedule). That's one major criterion for separating the text into chapters. Other criteria include that a chapter is a suitable unit for drills and exercises and that each chapter presents some specific concept, idea, or technique. This plurality of criteria has left a few chapters uncomfortably long, so please don't take "in one sitting" too literally. In particular, once you have thought about the review questions, done the drill, and

worked on a few exercises, you'll often find that you have to go back to reread a few sections and that several days have gone by. We have clustered the chapters into "parts" focused on a major topic, such as input/output. These parts make good units of review.


Common praise for a textbook is "It answered all my questions just as I thought of them!" That's an ideal for minor technical questions, and early readers have observed the phenomenon with this book. However, that cannot be the whole ideal. We raise questions that a novice would probably not think of. We aim to ask and answer questions that you need to consider to write quality software for the use of others. Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer. Asking only the easy and obvious questions would make you feel good, but it wouldn't help make you a programmer.

We try to respect your intelligence and to be considerate about your time. In our presentation, we aim for professionalism rather than cuteness, and we'd rather understate a point than hype it. We try not to exaggerate the importance of a programming technique or a language feature, but please don't underestimate a simple statement like "This is often useful." If we quietly emphasize that something is important, we mean that you'll sooner or later waste days if you don't master it. Our use of humor is more limited than we would have preferred, but experience shows that people's ideas of what is funny differ dramatically and that a failed attempt at humor can be confusing.



We do not pretend that our ideas or the tools offered are perfect. No tool, library, language, or technique is "the solution" to all of the many challenges facing a programmer. At best, it can help you to develop and express your solution. We try hard to avoid "white lies"; that is, we refrain from oversimplified explanations that are clear and easy to understand, but not true in the context of real languages and real problems. On the other hand, this book is not a reference; for more precise and complete descriptions of C++, see Bjarne Stroustrup, *The C++ Programming Language, Special Edition* (Addison-Wesley, 2000), and the ISO C++ standard.

0.1.2 Drills, exercises, etc.



Programming is not just an intellectual activity, so writing programs is necessary to master programming skills. We provide two levels of programming practice:

- *Drills:* A drill is a very simple exercise devised to develop practical, almost mechanical skills. A drill usually consists of a sequence of modifications of a single program. You should do every drill. A drill is not asking for deep understanding, cleverness, or initiative. We consider the drills part of the basic fabric of the book. If you haven't done the drills, you have not "done" the book.

- *Exercises:* Some exercises are trivial and others are very hard, but most are intended to leave some scope for initiative and imagination. If you are serious, you'll do quite a few exercises. At least do enough to know which are difficult for you. Then do a few more of those. That's how you'll learn the most. The exercises are meant to be manageable without exceptional cleverness, rather than to be tricky puzzles. However, we hope that we have provided exercises that are hard enough to challenge anybody and enough exercises to exhaust even the best student's available time. We do not expect you to do them all, but feel free to try.

In addition, we recommend that you (every student) take part in a small project (and more if time allows for it). A project is intended to produce a complete useful program. Ideally, a project is done by a small group of people (e.g., three people) working together for about a month while working through the chapters in Part III. Most people find the projects the most fun and what ties everything together.

Some people like to put the book aside and try some examples before reading to the end of a chapter; others prefer to read ahead to the end before trying to get code to run. To support readers with the former preference, we provide simple suggestions for practical work labeled “**Try this:**” at natural breaks in the text. A **Try this** is generally in the nature of a drill focused narrowly on the topic that precedes it. If you pass a **Try this** without trying – maybe because you are not near a computer or you find the text riveting – do return to it when you do the chapter drill; a **Try this** either complements the chapter drill or is a part of it.

At the end of each chapter you'll find a set of review questions. They are intended to point you to the key ideas explained in the chapter. One way to look at the review questions is as a complement to the exercises: the exercises focus on the practical aspects of programming, whereas the review questions try to help you articulate the ideas and concepts. In that, they resemble good interview questions.

The “Terms” section at the end of each chapter presents the basic vocabulary of programming and of C++. If you want to understand what people say about programming topics and to articulate your own ideas, you should know what each means.

Learning involves repetition. Our ideal is to make every important point at least twice and to reinforce it with exercises.


0.1.3 What comes after this book?

At the end of this book, will you be an expert at programming and at C++? Of course not! When done well, programming is a subtle, deep, and highly skilled art building on a variety of technical skills. You should no more expect to be an expert at programming in four months than you should expect to be an expert in biology, in math, in a natural language (such as Chinese, English, or Danish), or at playing the violin in four months – or in half a year, or a year. What you



should hope for, and what you can expect if you approach this book seriously, is to have a really good start that allows you to write relatively simple useful programs, to be able to read more complex programs, and to have a good conceptual and practical background for further work.

The best follow-up to this initial course is to work on a real project developing code to be used by someone else. After that, or (even better) in parallel with a real project, read either a professional-level general textbook (such as Stroustrup, *The C++ Programming Language*), a more specialized book relating to the needs of your project (such as Qt for GUI, or ACE for distributed programming), or a textbook focusing on a particular aspect of C++ (such as Koenig and Moo, *Accelerated C++*; Sutter's *Exceptional C++*; or Gamma et al., *Design Patterns*). For complete references, see §0.6 or the Bibliography section at the back of the book.



Eventually, you should learn another programming language. We don't consider it possible to be a professional in the realm of software – even if you are not primarily a programmer – without knowing more than one language.

0.2 A philosophy of teaching and learning

What are we trying to help you learn? And how are we approaching the process of teaching? We try to present the minimal concepts, techniques, and tools for you to do effective practical programs, including

- Program organization
- Debugging and testing
- Class design
- Computation
- Function and algorithm design
- Graphics (two-dimensional only)
- Graphical user interfaces (GUIs)
- Text manipulation
- Regular expression matching
- Files and stream input and output (I/O)
- Memory management
- Scientific/numerical/engineering calculations
- Design and programming ideals
- The C++ standard library
- Software development strategies
- C-language programming techniques


Working our way through these topics, we cover the programming techniques called procedural programming (as with the C programming language), data abstraction, object-oriented programming, and generic programming. The main topic of this book is *programming*, that is, the ideals, techniques, and tools of expressing ideas in code. The C++ programming language is our main tool, so we describe many of C++’s facilities in some detail. But please remember that C++ is just a tool, rather than the main topic of this book. This is “programming using C++,” not “C++ with a bit of programming theory.”

Each topic we address serves at least two purposes: it presents a technique, concept, or principle and also a practical language or library feature. For example, we use the interface to a two-dimensional graphics system to illustrate the use of classes and inheritance. This allows us to be economical with space (and your time) and also to emphasize that programming is more than simply slinging code together to get a result as quickly as possible. The C++ standard library is a major source of such “double duty” examples – many even do triple duty. For example, we introduce the standard library **vector**, use it to illustrate widely useful design techniques, and show many of the programming techniques used to implement it. One of our aims is to show you how major library facilities are implemented and how they map to hardware. We insist that craftsmen must understand their tools, not just consider them “magical.”

Some topics will be of greater interest to some programmers than to others. However, we encourage you not to prejudge your needs (how would you know what you’ll need in the future?) and at least look at every chapter. If you read this book as part of a course, your teacher will guide your selection.


We characterize our approach as “depth-first.” It is also “concrete-first” and “concept-based.” First, we quickly (well, relatively quickly, Chapters 1–11) assemble a set of skills needed for writing small practical programs. In doing so, we present a lot of tools and techniques in minimal detail. We focus on simple concrete code examples because people grasp the concrete faster than the abstract. That’s simply the way most humans learn. At this initial stage, you should not expect to understand every little detail. In particular, you’ll find that trying something slightly different from what just worked can have “mysterious” effects. Do try, though! And please do the drills and exercises we provide. Just remember that early on you just don’t have the concepts and skills to accurately estimate what’s simple and what’s complicated; expect surprises and learn from them.

We move fast in this initial phase – we want to get you to the point where you can write interesting programs as fast as possible. Someone will argue, “We must move slowly and carefully; we must walk before we can run!” But have you ever watched a baby learning to walk? Babies really do run by themselves before they learn the finer skills of slow, controlled walking. Similarly, you will dash ahead, occasionally stumbling, to get a feel of programming before slowing down to gain the necessary finer control and understanding. You must run before you can walk!



It is essential that you don't get stuck in an attempt to learn "everything" about some language detail or technique. For example, you could memorize all of C++'s built-in types and all the rules for their use. Of course you could, and doing so might make you feel knowledgeable. However, it would not make you a programmer. Skipping details will get you "burned" occasionally for lack of knowledge, but it is the fastest way to gain the perspective needed to write good programs. Note that our approach is essentially the one used by children learning their native language and also the most effective approach used to teach foreign languages. We encourage you to seek help from teachers, friends, colleagues, instructors, Mentors, etc. on the inevitable occasions when you are stuck. Be assured that nothing in these early chapters is fundamentally difficult. However, much will be unfamiliar and might therefore feel difficult at first.

Later, we build on the initial skills to broaden your base of knowledge and skills. We use examples and exercises to solidify your understanding, and to provide a conceptual base for programming.



We place a heavy emphasis on ideals and reasons. You need ideals to guide you when you look for practical solutions – to know when a solution is good and principled. You need to understand the reasons behind those ideals to understand why they should be your ideals, why aiming for them will help you and the users of your code. Nobody should be satisfied with "because that's the way it is" as an explanation. More importantly, an understanding of ideals and reasons allows you to generalize from what you know to new situations and to combine ideas and tools in novel ways to address new problems. Knowing "why" is an essential part of acquiring programming skills. Conversely, just memorizing lots of poorly understood rules and language facilities is limiting, a source of errors, and a massive waste of time. We consider your time precious and try not to waste it.

Many C++ language-technical details are banished to appendices and manuals, where you can look them up when needed. We assume that you have the initiative to search out information when needed. Use the index and the table of contents. Don't forget the online help facilities of your compiler, and the web. Remember, though, to consider every web resource highly suspect until you have reason to believe better of it. Many an authoritative-looking website is put up by a programming novice or someone with something to sell. Others are simply outdated. We provide a collection of links and information on our support website: www.stroustrup.com/Programming.

Please don't be too impatient for "realistic" examples. Our ideal example is the shortest and simplest code that directly illustrates a language facility, a concept, or a technique. Most real-world examples are far messier than ours, yet do not consist of more than a combination of what we demonstrate. Successful commercial programs with hundreds of thousands of lines of code are based on techniques that we illustrate in a dozen 50-line programs. The fastest way to understand real-world code is through a good understanding of the fundamentals.

On the other hand, we do not use “cute examples involving cuddly animals” to illustrate our points. We assume that you aim to write real programs to be used by real people, so every example that is not presented as language-technical is taken from a real-world use. Our basic tone is that of professionals addressing (future) professionals.

0.2.1 The order of topics

There are many ways to teach people how to program. Clearly, we don’t subscribe to the popular “the way I learned to program is the best way to learn” theories. To ease learning, we early on present topics that would have been considered advanced only a few years ago. Our ideal is for the topics we present to be driven by problems you meet as you learn to program, to flow smoothly from topic to topic as you increase your understanding and practical skills. The major flow of this book is more like a story than a dictionary or a hierarchical order.

It is impossible to learn all the principles, techniques, and language facilities needed to write a program at once. Consequently, we have to choose a subset of principles, techniques, and features to start with. More generally, a textbook or a course must lead students through a series of subsets. We consider it our responsibility to select topics and to provide emphasis. We can’t just present everything, so we must choose; what we leave out is at least as important as what we leave in – at each stage of the journey.

For contrast, it may be useful for you to see a list of (severely abbreviated) characterizations of approaches that we decided not to take:

- *“C first”*: This approach to learning C++ is wasteful of students’ time and leads to poor programming practices by forcing students to approach problems with fewer facilities, techniques, and libraries than necessary. C++ provides stronger type checking than C, a standard library with better support for novices, and exceptions for error handling.
- *Bottom-up*: This approach distracts from learning good and effective programming practices. By forcing students to solve problems with insufficient support from the language and libraries, it promotes poor and wasteful programming practices.
- *“If you present something, you must present it fully”*: This approach implies a bottom-up approach (by drilling deeper and deeper into every topic touched). It bores novices with technical details they have no interest in and quite likely will not need for years to come. Once you can program, you can look up technical details in a manual. Manuals are good at that, whereas they are awful for initial learning of concepts.

- *Top-down*: This approach, working from first principles toward details, tends to distract readers from the practical aspects of programming and force them to concentrate on high-level concepts before they have any chance of appreciating their importance. For example, you simply can't appreciate proper software development principles before you have learned how easy it is to make a mistake in a program and how hard it can be to correct it.
- *"Abstract first"*: Focusing on general principles and protecting the student from nasty real-world constraints can lead to a disdain for real-world problems, languages, tools, and hardware constraints. Often, this approach is supported by "teaching languages" that cannot be used later and (deliberately) insulate students from hardware and system concerns.
- *Software engineering principles first*: This approach and the abstract-first approach tend to share the problems of the top-down approach: without concrete examples and practical experience, you simply cannot appreciate the value of abstraction and proper software development practices.
- *"Object-oriented from day one"*: Object-oriented programming is one of the best ways of organizing code and programming efforts, but it is not the only effective way. In particular, we feel that a grounding in the basics of types and algorithmic code is a prerequisite for appreciation of the design of classes and class hierarchies. We do use user-defined types (what some people would call "objects") from day one, but we don't show how to design a class until Chapter 6 and don't show a class hierarchy until Chapter 12.
- *"Just believe in magic"*: This approach relies on demonstrations of powerful tools and techniques without introducing the novice to the underlying techniques and facilities. This leaves the student guessing – and usually guessing wrong – about why things are the way they are, what it costs to use them, and where they can be reasonably applied. This can lead to overrigid following of familiar patterns of work and become a barrier to further learning.

Naturally, we do not claim that these other approaches are never useful. In fact, we use several of these for specific subtopics where their strengths can be appreciated. However, as general approaches to learning programming aimed at real-world use, we reject them and apply our alternative: concrete-first and depth-first with an emphasis on concepts and techniques.

0.2.2 Programming and programming language



We teach programming first and treat our chosen programming language as secondary, as a tool. Our general approach can be used with any general-purpose

programming language. Our primary aim is to help you learn general concepts, principles, and techniques. However, those cannot be appreciated in isolation. For example, details of syntax, the kinds of ideas that can be directly expressed, and tool support differ from programming language to programming language. However, many of the fundamental techniques for producing bug-free code, such as writing logically simple code (Chapters 5 and 6), establishing invariants (§9.4.3), and separating interfaces from implementation details (§9.7 and §14.1–2), vary little from programming language to programming language.

Programming and design techniques must be learned using a programming language. Design, code organization, and debugging are not skills you can acquire in the abstract. You need to write code in some programming language and gain practical experience with that. This implies that you must learn the basics of a programming language. We say “the basics” because the days when you could learn all of a major industrial language in a few weeks are gone for good. The parts of C++ we present were chosen as the subset that most directly supports the production of good code. Also, we present C++ features that you can’t avoid encountering either because they are necessary for logical completeness or are common in the C++ community.

0.2.3 Portability

It is common to write C++ to run on a variety of machines. Major C++ applications run on machines we haven’t ever heard of! We consider portability and the use of a variety of machine architectures and operating systems most important. Essentially every example in this book is not only ISO Standard C++, but also portable. Unless specifically stated, the code we present should work on every C++ implementation and has been tested on several machines and operating systems.

The details of how to compile, link, and run a C++ program differ from system to system. It would be tedious to mention the details of every system and every compiler each time we need to refer to an implementation issue. In Appendix E, we give the most basic information about getting started using Visual Studio and Microsoft C++ on a Windows machine.

If you have trouble with one of the popular, but rather elaborate, IDEs (integrated development environments), we suggest you try working from the command line; it’s surprisingly simple. For example, here is the full set of commands needed to compile, link, and execute a simple program consisting of two source files, `my_file1.cpp` and `my_file2.cpp`, using the GNU C++ compiler, `g++`, on a Unix or Linux system:

```
g++ -o my_program my_file1.cpp my_file2.cpp  
my_program
```

Yes, that really is all it takes.



0.3 Programming and computer science

Is programming all that there is to computer science? Of course not! The only reason we raise this question is that people have been known to be confused about this. We touch upon major topics from computer science, such as algorithms and data structures, but our aim is to teach programming: the design and implementation of programs. That is both more and less than most accepted notions of computer science:

- *More*, because programming involves many technical skills that are not usually considered part of any science
- *Less*, because we do not systematically present the foundation for the parts of computer science we use

The aim of this book is to be part of a course in computer science (if becoming a computer scientist is your aim), to be the foundation for the first of many courses in software construction and maintenance (if your aim is to become a programmer or a software engineer), and in general to be part of a greater whole.

We rely on computer science throughout and we emphasize principles, but we teach programming as a practical skill based on theory and experience, rather than as a science.

0.4 Creativity and problem solving

The primary aim of this book is to help you to express your ideas in code, not to teach you how to get those ideas. Along the way, we give many examples of how we can address a problem, usually through analysis of a problem followed by gradual refinement of a solution. We consider programming itself a form of problem solving: only through complete understanding of a problem and its solution can you express a correct program for it, and only through constructing and testing a program can you be certain that your understanding is complete. Thus, programming is inherently part of an effort to gain understanding. However, we aim to demonstrate this through examples, rather than through “preaching” or presentation of detailed prescriptions for problem solving.

0.5 Request for feedback

We don’t think that the perfect textbook can exist; the needs of individuals differ too much for that. However, we’d like to make this book and its supporting materials as good as we can make them. For that, we need feedback; a good textbook cannot be written in isolation from its readers. Please send us reports on

errors, typos, unclear text, missing explanations, etc. We'd also appreciate suggestions for better exercises, better examples, and topics to add, topics to delete, etc. Constructive comments will help future readers and we'll post errata on our support website: www.stroustrup.com/Programming.

0.6 References

Along with listing the publications mentioned in this chapter, this section also includes publications you might find helpful.

- Austern, Matthew H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999. ISBN 0201309564.
- Austern, Matthew H. (editor). "Technical Report on C++ Standard Library Extensions." ISO/IEC PDTR 19768.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872493.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.
- Goldthwaite, Lois (editor). "Technical Report on C++ Performance." ISO/IEC PDTR 18015.
- Koenig, Andrew (editor). *The C++ Standard*. ISO/IEC 14882:2002. Wiley, 2003. ISBN 0470846747.
- Koenig, Andrew, and Barbara Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Langer, Angelika, and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley, 2000. ISBN 0201183951.
- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0201749625.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley, 2005. ISBN 0321334876.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003. ISBN 0201795256.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.

Stroustrup, Bjarne. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000. ISBN 0201700735.

Stroustrup, Bjarne. “C and C++: Siblings”; “C and C++: A Case for Compatibility”; and “C and C++: Case Studies in Compatibility.” *C/C++ Users Journal*, July, Aug., Sept. 2002.

Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.

A more comprehensive list of references can be found in the Bibliography section at the back of the book.

0.7 Biographies

You might reasonably ask, “Who are these guys who want to teach me how to program?” So here is some biographical information. I, Bjarne Stroustrup, wrote this book, and together with Lawrence “Pete” Petersen, I designed and taught the university-level beginner’s (first-year) course that was developed concurrently with the book, using drafts of the book.

Bjarne Stroustrup



I’m the designer and original implementer of the C++ programming language. I have used the language, and many other programming languages, for a wide variety of programming tasks over the last 30 years or so. I just love elegant and efficient code used in challenging applications, such as robot control, graphics, games, text analysis, and networking. I have taught design, programming, and C++ to people of essentially all abilities and interests. I’m a founding member of the ISO standards committee for C++ where I serve as the

chair of the working group for language evolution.

This is my first introductory book. My other books, such as *The C++ Programming Language* and *The Design and Evolution of C++*, were written for experienced programmers.

I was born into a blue-collar (working-class) family in Århus, Denmark, and got my master’s degree in mathematics with computer science in my hometown university. My Ph.D. in computer science is from Cambridge University, England. I worked for AT&T for about 25 years, first in the famous Computer Science Research Center of Bell Labs – where Unix, C, C++, and so much else were invented – and later in AT&T Labs–Research.

I’m a member of the U.S. National Academy of Engineering, a Fellow of the ACM, an IEEE Fellow, a Bell Laboratories Fellow, and an AT&T Fellow. As the

first computer scientist ever, I received the 2005 William Procter Prize for Scientific Achievement from Sigma Xi (the scientific research society).

I do have a life outside work. I'm married and have two children, one a medical doctor and one a Ph.D. student. I read a lot (including history, science fiction, crime, and current affairs) and like most kinds of music (including classical, rock, blues, and country). Good food with friends is an essential part of life, and I enjoy visiting interesting places and people, all over the world. To be able to enjoy the good food, I run.

For more information, see my home pages: www.research.att.com/~bs and www.cs.tamu.edu/people/faculty/bs. In particular, there you can find out how to pronounce my name.

Lawrence “Pete” Petersen



In late 2006, Pete introduced himself as follows: “I am a teacher. For almost 20 years, I have taught programming languages at Texas A&M. I have been selected by students for Teaching Excellence Awards five times and in 1996 received the Distinguished Teaching Award from the Alumni Association for the College of Engineering. I am a Fellow of the Wakonse Program for Teaching Excellence and a Fellow of the Academy for Educator Development.

As the son of an army officer, I was raised on the move. After completing a degree in philosophy at the University of Washington, I served in the army for 22 years as a Field Artillery Officer and as a Research Analyst for Operational Testing. I taught at the Field Artillery Officer’s Advanced Course at Fort Sill, Oklahoma, from 1971 to 1973. In 1979 I helped organize a Test Officer’s Training Course and taught it as lead instructor at nine different locations across the United States from 1978 to 1981 and from 1985 to 1989.

In 1991 I formed a small software company that produced management software for university departments until 1999. My interests are in teaching, designing, and programming software that real people can use. I completed master’s degrees in industrial engineering at Georgia Tech and in education curriculum and instruction at Texas A&M. I also completed a master’s program in microcomputers from NTS. My Ph.D. is in information and operations management from Texas A&M.

My wife, Barbara, and I live in Bryan, Texas. We like to travel, garden, and entertain; and we spend as much time as we can with our sons and their families, and especially with our grandchildren, Angelina, Carlos, Tess, Avery, Nicholas, and Jordan.”

Sadly, Pete died of lung cancer in 2007. Without him, the course would never have succeeded.

Postscript

Most chapters provide a short “postscript” trying to give some perspective on the information presented in the chapter. We do that in the realization that the information can be – and often is – daunting and will only be fully comprehended after doing exercises, reading further chapters (which apply the ideas of the chapter), and a later review. Don’t panic. Relax; this is natural and expected. You won’t become an expert in a day, but you can become a reasonably competent programmer as you work your way through the book. On the way, you’ll encounter much information, many examples, and many techniques that lots of programmers have found stimulating and fun.



A Display Model

“The world was black and white then.
[It] didn’t turn color
until sometime in the 1930s.”

— Calvin’s dad

This chapter presents a display model (the output part of GUI), giving examples of use and fundamental notions such as screen coordinates, lines, and color. **Line**, **Lines**, **Polygons**, **Axis**, and **Text** are examples of **Shapes**. A **Shape** is an object in memory that we can display and manipulate on a screen. The next two chapters will explore these classes further, with Chapter 13 focusing on their implementation and Chapter 14 on design issues.

12.1 Why graphics?	12.7 Using Shape primitives
12.2 A display model	12.7.1 Graphics headers and main
12.3 A first example	12.7.2 An almost blank window
12.4 Using a GUI library	12.7.3 Axis
12.5 Coordinates	12.7.4 Graphing a function
12.6 Shapes	12.7.5 Polygons
	12.7.6 Rectangles
	12.7.7 Fill
	12.7.8 Text
	12.7.9 Images
	12.7.10 And much more
	12.8 Getting this to run
	12.8.1 Source files

12.1 Why graphics?

Why do we spend four chapters on graphics and one on GUIs (graphical user interfaces)? After all, this is a book about programming, not a graphics book. There is a huge number of interesting software topics that we don't discuss, and we can at best scratch the surface on the topic of graphics. So, "Why graphics?" Basically, graphics is a subject that allows us to explore several important areas of software design, programming, and programming language facilities:

- *Graphics are useful.* There is much more to programming than graphics and much more to software than code manipulated through a GUI. However, in many areas good graphics are either essential or very important. For example, we wouldn't dream of studying scientific computing, data analysis, or just about any quantitative subject without the ability to graph data. Chapter 15 gives simple (but general) facilities for graphing data.
- *Graphics are fun.* There are few areas of computing where the effect of a piece of code is as immediately obvious and – when finally free of bugs – as pleasing. We'd be tempted to play with graphics even if it wasn't useful!
- *Graphics provide lots of interesting code to read.* Part of learning to program is to read lots of code to get a feel for what good code is like. Similarly, the way to become a good writer of English involves reading a lot of books, articles, and quality newspapers. Because of the direct correspondence between what we see on the screen and what we write in our programs, simple graphics code is more readable than most kinds of code of similar complexity. This chapter will prove that you can read graphics code after a few minutes of introduction; Chapter 13 will demonstrate how you can write it after another couple of hours.

- *Graphics are a fertile source of design examples.* It is actually hard to design and implement a good graphics and GUI library. Graphics are a very rich source of concrete and practical examples of design decisions and design techniques. Some of the most useful techniques for designing classes, designing functions, separating software into layers (of abstraction), and constructing libraries can be illustrated with a relatively small amount of graphics and GUI code.
- *Graphics provide a good introduction to what is commonly called object-oriented programming and the language features that support it.* Despite rumors to the contrary, object-oriented programming wasn't invented to be able to do graphics (see Chapter 22), but it was soon applied to that, and graphics provide some of the most accessible examples of object-oriented designs.
- *Some of the key graphics concepts are nontrivial.* So they are worth teaching, rather than leaving it to your own initiative (and patience) to seek out information. If we did not show how graphics and GUI were done, you might consider them “magic,” thus violating one of the fundamental aims of this book.

12.2 A display model

The iostream library is oriented toward reading and writing streams of characters as they might appear in a list of numeric values or a book. The only direct supports for the notion of graphical position are the newline and tab characters. You can embed notions of color and two-dimensional positions, etc., in a one-dimensional stream of characters. That's what layout (typesetting, “markup”) languages such as Troff, Tex, Word, HTTP, and XML (and their associated graphical packages) do. For example:

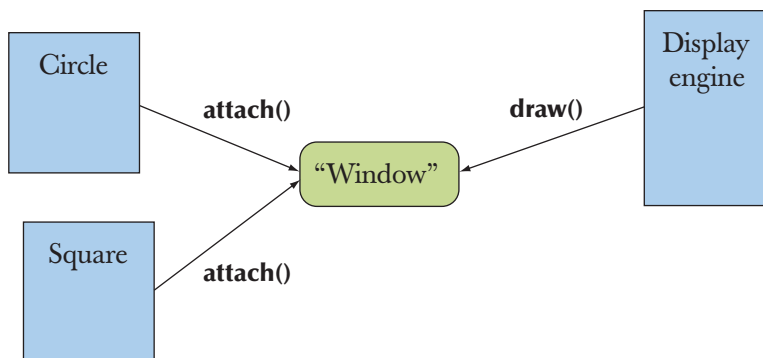
```
<hr>
<h2>
Organization
</h2>
This list is organized in three parts:
<ul>
    <li><b>Proposals</b>, numbered EPddd, . . .</li>
    <li><b>Issues</b>, numbered EIddd, . . .</li>
    <li><b>Suggestions</b>, numbered ESddd, . . .</li>
</ul>
<p>We try to . . .
<p>
```

This is a piece of HTML specifying a header (<h2> . . . </h2>) a list (. . .) with list items (. . .) and a paragraph (<p>). We left out most of

the actual text because it is irrelevant here. The point is that you can express layout notions in plain text, but the connection between the characters written and what appears on the screen is indirect, governed by a program that interprets those “markup” commands. Such techniques are fundamentally simple and immensely useful (just about everything you read has been produced using them), but they also have their limitations.

In this chapter and the next four, we present an alternative: a notion of graphics and of graphical user interfaces that is directly aimed at a computer screen. The fundamental concepts are inherently graphical (and two-dimensional, adapted to the rectangular area of a computer screen), such as coordinates, lines, rectangles, and circles. The aim from a programming point of view is a direct correspondence between the objects in memory and the images on the screen.

The basic model is as follows: We compose objects with basic objects provided by a graphics system, such as lines. We “attach” these graphics objects to a window object, representing our physical screen. A program that we can think of as the display itself, as “a display engine,” as “our graphics library,” as “the GUI library,” or even (humorously) as “the small gnome writing on the back of the screen” then takes the objects we have added to our window and draws them on the screen:



The “display engine” draws lines on the screen, places strings of text on the screen, colors areas of the screen, etc. For simplicity, we’ll use the phrase “our GUI library” or even “the system” for the display engine even though our GUI library does much more than just drawing the objects. In the same way that our code lets the GUI library do most of the work for us, the GUI library delegates much of its work to the operating system.

12.3 A first example

Our job is to define classes from which we can make objects that we want to see on the screen. For example, we might want to draw a graph as a series of connected lines. Here is a small program presenting a very simple version of that:

```

#include "Simple_window.h"    // get access to our window library
#include "Graph.h"           // get access to our graphics library facilities

int main()
{
    using namespace Graph_lib; // our graphics facilities are in Graph_lib

    Point tl(100,100);        // to become top left corner of window

    Simple_window win(tl,600,400,"Canvas"); // make a simple window

    Polygon poly;             // make a shape (a polygon)

    poly.add(Point(300,200)); // add a point
    poly.add(Point(350,100)); // add another point
    poly.add(Point(400,200)); // add a third point

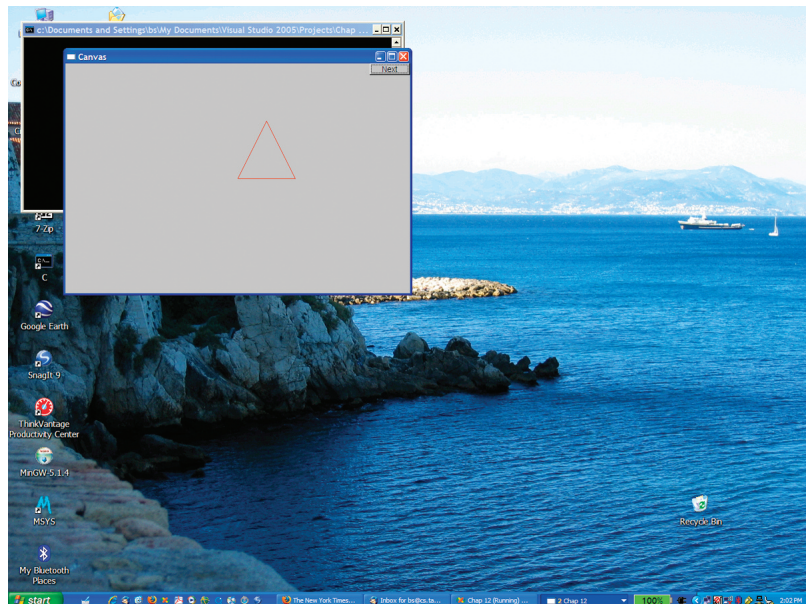
    poly.set_color(Color::red); // adjust properties of poly

    win.attach (poly);        // connect poly to the window

    win.wait_for_button();    // give control to the display engine
}

```

When we run this program, the screen looks something like this:



Let's go through the program line by line to see what was done. First we include the headers for our graphics interface libraries:

```
#include "Simple_window.h"    // get access to our window library
#include "Graph.h"           // get access to our graphics library facilities
```

Then, in `main()`, we start by telling the compiler that our graphics facilities are to be found in `Graph_lib`:

```
using namespace Graph_lib;    // our graphics facilities are in Graph_lib
```

Then, we define a point that we will use as the top left corner of our window:

```
Point tl(100,100);    // to become top left corner of window
```

Next, we create a window on the screen:

```
Simple_window win(tl,600,400,"Canvas");    // make a simple window
```

We use a class representing a window in our `Graph_lib` interface library called `Simple_window`. The name of this particular `Simple_window` is `win`; that is, `win` is a variable of class `Simple_window`. The initializer list for `win` starts with the point to be used as the top left corner, `tl`, followed by `600` and `400`. Those are the width and height, respectively, of the window, as displayed on the screen, measured in pixels. We'll explain in more detail later, but the main point here is that we specify a rectangle by giving its width and height. The string `Canvas` is used to label the window. If you look, you can see the word `Canvas` in the top left corner of the window's frame.

On our screen, the window appeared in a position chosen by the GUI library. In §13.7.2, we'll show how to choose a particular position, but for now, we'll just take what our library picks; that's often just right anyway.

Next, we put an object in the window:

```
Polygon poly;                // make a shape (a polygon)

poly.add(Point(300,200));    // add a point
poly.add(Point(350,100));    // add another point
poly.add(Point(400,200));    // add a third point
```

We define a polygon, `poly`, and then add points to it. In our graphics library, a `Polygon` starts empty and we can add as many points to it as we like. Since we added three points, we get a triangle. A point is simply a pair of values giving the x and y (horizontal and vertical) coordinates within a window.

Just to show off, we then color the lines of our polygon red:

```
poly.set_color(Color::red);    // adjust properties of poly
```

Finally, we attach **poly** to our window, **win**:

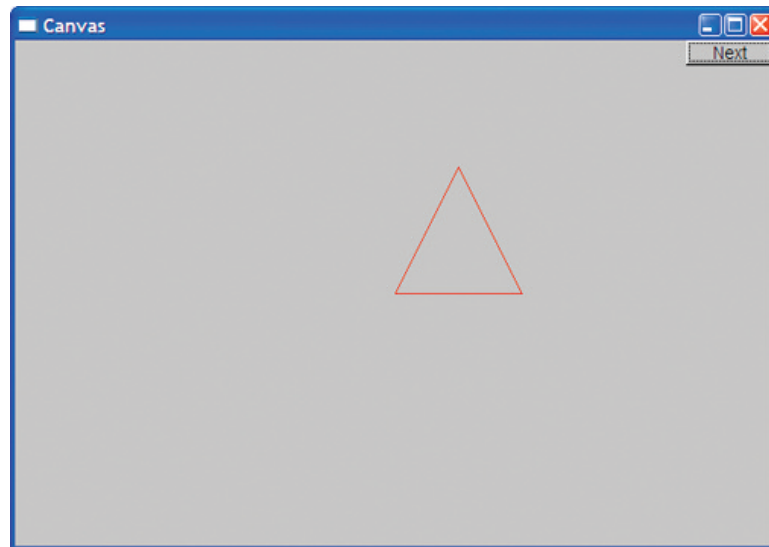
```
win.attach(poly);              // connect poly to the window
```

If the program wasn't so fast, you would notice that so far nothing had happened to the screen: nothing at all. We created a window (an object of class **Simple_window**, to be precise), created a polygon (called **poly**), painted that polygon red (**Color::red**), and attached it to the window (called **win**), but we have not yet asked for that window to be displayed on the screen. That's done by the final line of the program:

```
win.wait_for_button();         // give control to the display engine
```

To get a GUI system to display objects on the screen, you have to give control to “the system.” Our **wait_for_button()** does that, and it also waits for you to “press” (“click”) the “Next” button of our **Simple_window** before proceeding. This gives you a chance to look at the window before the program finishes and the window disappears. When you press the button, the program terminates, closing the window.


In isolation, our window looks like this:



You'll notice that we “cheated” a bit. Where did that button labeled “Next” come from? We built it into our **Simple_window** class. In Chapter 16, we'll move from

Simple_window to “plain” **Window**, which has no potentially spurious facilities built in, and show how we can write our own code to control interaction with a window.

For the next three chapters, we’ll simply use that “Next” button to move from one “display” to the next when we want to display information in stages (“frame by frame”).

You are so used to the operating system putting a frame around each window that you might not have noticed it specifically. However, the pictures in this and the following chapters were produced on a Microsoft Windows system, so you get the usual three buttons on the top right “for free.” This can be useful: if your program gets in a real mess (as it surely will sometimes during debugging), you can kill it by hitting the  button. When you run your program on another system, a different frame will be added to fit that system’s conventions. Our only contribution to the frame is the label (here, **Canvas**).

12.4 Using a GUI library

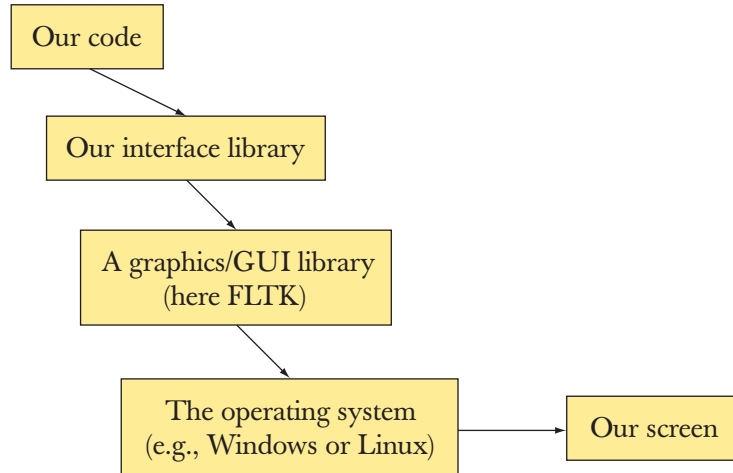


In this book, we will not use the operating system’s graphical and GUI (graphical user interface) facilities directly. Doing so would limit our programs to run on a single operating system and would also force us to deal directly with a lot of messy details. As with text I/O, we’ll use a library to smooth over operating system differences, I/O device variations, etc. and to simplify our code. Unfortunately, C++ does not provide a standard GUI library the way it provides the standard stream I/O library, so we use one of the many available C++ GUI libraries. So as not to tie you directly into one of those GUI libraries, and to save you from hitting the full complexity of a GUI library all at once, we use a set of simple interface classes that can be implemented in a couple of hundred lines of code for just about any GUI library.

The GUI toolkit that we are using (indirectly for now) is called FLTK (Fast Light Tool Kit, pronounced “full tick”) from www.fltk.org. Our code is portable wherever FLTK is used (Windows, Unix, Mac, Linux, etc.). Our interface classes can also be re-implemented using other toolkits, so code using them is potentially even more portable.

The programming model presented by our interface classes is far simpler than what common toolkits offer. For example, our complete graphics and GUI interface library is about 600 lines of C++ code, whereas the extremely terse FLTK documentation is 370 pages. You can download that from www.fltk.org, but we don’t recommend you do that just yet. You can do without that level of detail for a while. The general ideas presented in Chapters 12–16 can be used with any popular GUI toolkit. We will of course explain how our interface classes map to FLTK so that you will (eventually) see how you can use that (and similar toolkits) directly, if necessary.

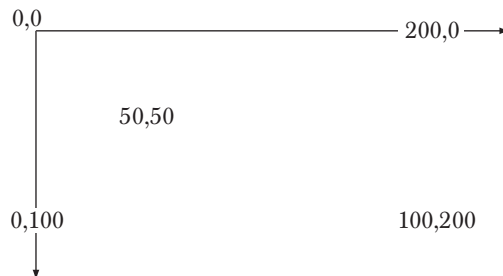
We can illustrate the parts of our “graphics world” like this:



Our interface classes provide a simple and user-extensible basic notion of two-dimensional shapes with limited support for the use of color. To drive that, we present a simple notion of GUI based on “callback” functions triggered by the use of user-defined buttons, etc. on the screen (Chapter 16).

12.5 Coordinates

A computer screen is a rectangular area composed of pixels. A pixel is a tiny spot that can be given some color. The most common way of modeling a screen in a program is as a rectangle of pixels. Each pixel is identified by an x (horizontal) coordinate and a y (vertical) coordinate. The x coordinates start with 0, indicating the leftmost pixel, and increase (toward the right) to the rightmost pixel. The y coordinates start with 0, indicating the topmost pixel, and increase (toward the bottom) to the lowest pixel:





Please note that y coordinates “grow downward.” Mathematicians, in particular, find this odd, but screens (and windows) come in many sizes, and the top left point is about all that they have in common.

The number of pixels available depends on the screen: 1024-by-768, 1280-by-1024, 1450-by-1050, and 1600-by-1200 are common screen sizes.

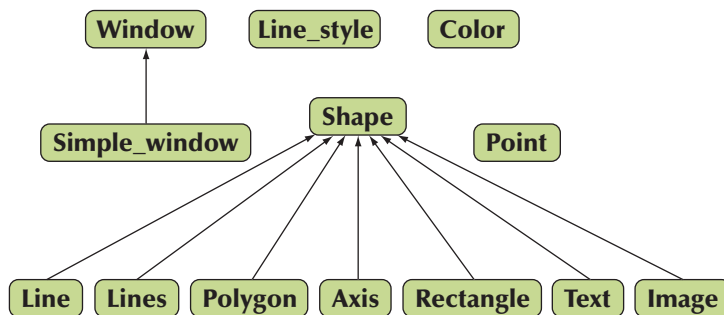
In the context of interacting with a computer using a screen, a window is a rectangular region of the screen devoted to some specific purpose and controlled by a program. A window is addressed exactly as a screen. Basically, we see a window as a small screen. For example, when we said

```
Simple_window win(1,600,400,"Canvas");
```

we requested a rectangular area 600 pixels wide and 400 pixels high that we can address 0–599 (left to right) and 0–399 (top to bottom). The area of a window that you can draw on is commonly referred to as a *canvas*. The 600-by-400 area refers to “the inside” of the window, that is, the area inside the system-provided frame; it does not include the space the system uses for the title bar, quit button, etc.

12.6 Shapes

Our basic toolbox for drawing on the screen consists of about a dozen classes:



An arrow indicates that the class pointing to can be used where the class pointed to is required. For example, a **Polygon** can be used where a **Shape** is required; that is, a **Polygon** is a kind of **Shape**.

We will start out presenting and using

- **Simple_window**, **Window**
- **Shape**, **Text**, **Polygon**, **Line**, **Lines**, **Rectangle**, **Function**, etc.
- **Color**, **Line_style**, **Point**
- **Axis**

Later (Chapter 16), we'll add GUI (user interaction) classes:

- **Button**, **In_box**, **Menu**, etc.

We could easily add many more classes (for some definition of “easy”), such as

- **Spline**, **Grid**, **Block_chart**, **Pie_chart**, etc.

However, defining or describing a complete GUI framework with all its facilities is beyond the scope of this book.

12.7 Using Shape primitives

In this section, we will walk you through some of the primitive facilities of our graphics library: **Simple_window**, **Window**, **Shape**, **Text**, **Polygon**, **Line**, **Lines**, **Rectangle**, **Color**, **Line_style**, **Point**, **Axis**. The aim is to give you a broad view of what you can do with those facilities, but not yet a detailed understanding of any of those classes. In the next chapters, we explore the design of each.

We will now walk through a simple program, explaining the code line by line and showing the effect of each on the screen. When you run the program you'll see how the image changes as we add shapes to the window and modify existing shapes. Basically, we are “animating” the progress through the code by looking at the program as it is executed.

12.7.1 Graphics headers and main

First, we include the header files defining our interface to the graphics and GUI facilities:

```
#include "Window.h"    // a plain window
#include "Graph.h"
```

or

```
#include "Simple_window.h"    // if we want that “Next” button
#include "Graph.h"
```

As you probably guessed, **Window.h** contains the facilities related to windows and **Graph.h** the facilities related to drawing shapes (including text) into windows. These facilities are defined in the **Graph_lib** namespace. To simplify notation we use a namespace directive to make the names from **Graph_lib** directly available in our program:

```
using namespace Graph_lib;
```

As usual, **main()** contains the code we want to execute (directly or indirectly) and deals with exceptions:

```
int main ()
try
{
    // . . . here is our code . . .

}
catch(exception& e) {
    // some error reporting
    return 1;
}
catch(...) {
    // some more error reporting
    return 2;
}
```

12.7.2 An almost blank window

We will not discuss error handling here (see Chapter 5, in particular, §5.6.3), but go straight to the graphics within **main()**:

```
Point tl(100,100); // top left corner of our window

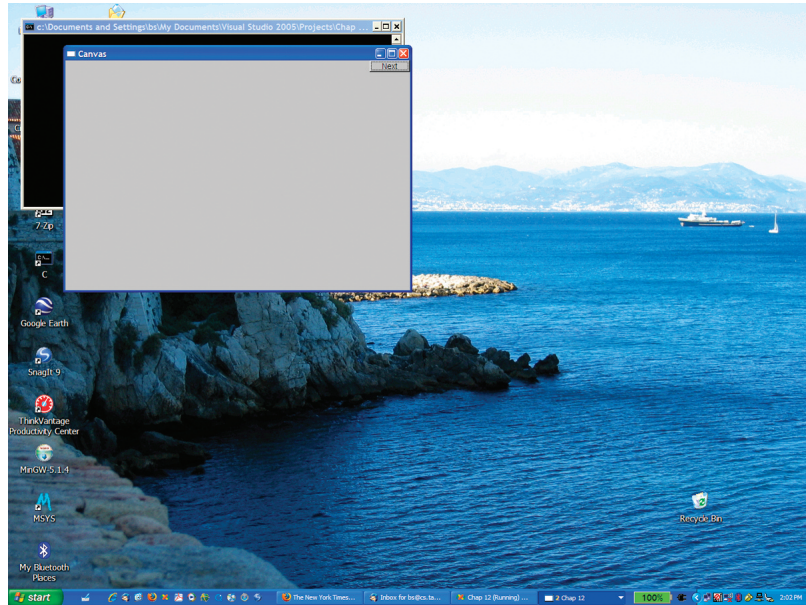
Simple_window win(tl,600,400,"Canvas");
    // screen coordinate tl for top left corner
    // window size(600*400)
    // title: Canvas
win.wait_for_button(); // display!
```

This creates a **Simple_window**, that is, a window with a “Next” button, and displays it on the screen. Obviously, we need to have **#included** the header **Simple_window.h** rather than **Window.h** to get **Simple_window**. Here we are specific about where on the screen the window should go: its top left corner goes at **Point(100,100)**. That’s near, but not too near, the top left corner of the screen. Obviously, **Point** is a class with a constructor that takes a pair of integers and interprets them as an (x,y) coordinate pair. We could have written

```
Simple_window win(Point(100,100),600,400,"Canvas");
```

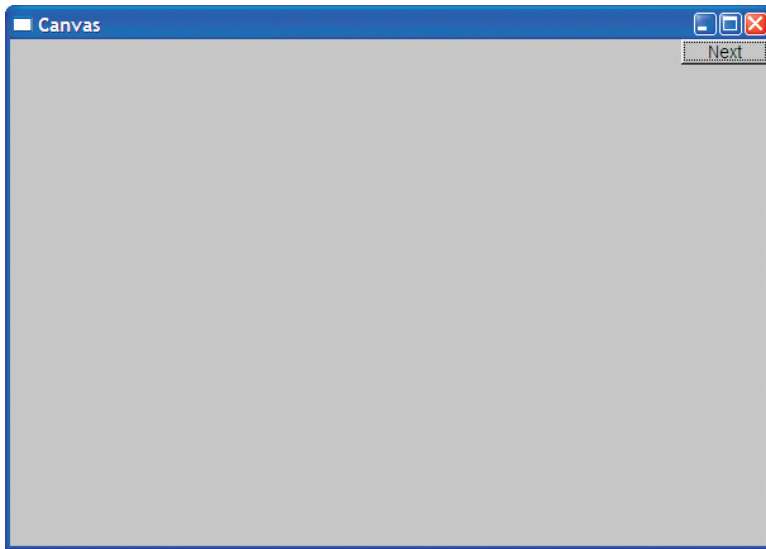
However, we want to use the point (100,100) several times so it is more convenient to give it a symbolic name. The 600 is the width and 400 is the height of the window, and **Canvas** is the label we want put on the frame of the window.

To actually get the window drawn on the screen, we have to give control to the GUI system. We do this by calling `win.wait_for_button()` and the result is:



In the background of our window, we see a laptop screen (somewhat cleaned up for the occasion). For people who are curious about irrelevant details, we can tell you that I took the photo standing near the Picasso library in Antibes looking across the bay to Nice. The black console window partially hidden behind is the one running our program. Having a console window is somewhat ugly and unnecessary, but it has the advantage of giving us an effective way of killing our window if a partially debugged program gets into an infinite loop and refuses to go away. If you look carefully, you'll notice that we have the Microsoft C++ compiler running, but you could just as well have used some other compiler (such as Borland or GNU).

For the rest of the presentation we will eliminate the distractions around our window and just show that window by itself:



The actual size of the window (in inches) depends on the resolution of your screen. Some screens have bigger pixels than other screens.

12.7.3 Axis

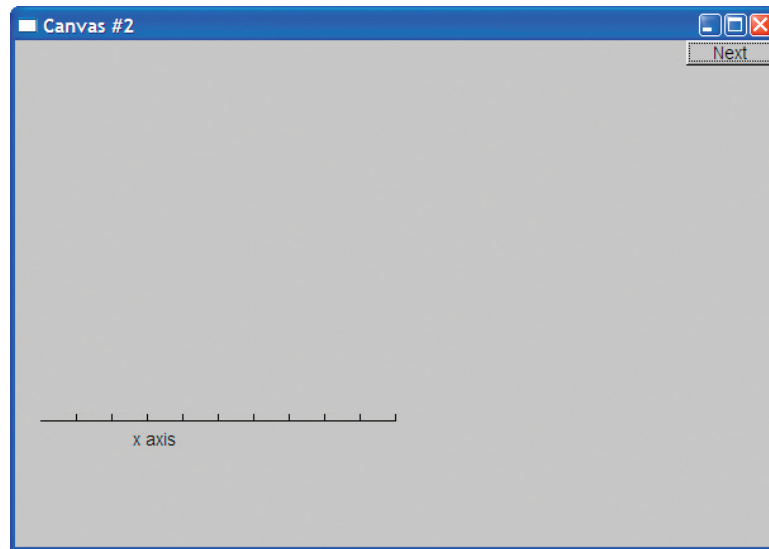
An almost blank window isn't very interesting, so we'd better add some information. What would we like to display? Just to remind you that graphics is not all fun and games, we will start with something serious and somewhat complicated: an axis. A graph without axes is usually a disgrace. You just don't know what the data represents without axes. Maybe you explained it all in some accompanying text, but it is far safer to add axes; people often don't read the explanation and often a nice graphical representation gets separated from its original context. So, a graph needs axes:

```

Axis xa(Axis::x, Point(20,300), 280, 10, "x axis");    // make an Axis
    // an Axis is a kind of Shape
    // Axis::x means horizontal
    // starting at (20,300)
    // 280 pixels long
    // 10 "notches"
    // label the axis "x axis"
win.attach(xa);    // attach xa to the window, win
win.set_label("Canvas #2");    // relabel the window
win.wait_for_button();    // display!

```

The sequence of actions is: make the axis object, add it to the window, and finally display it:



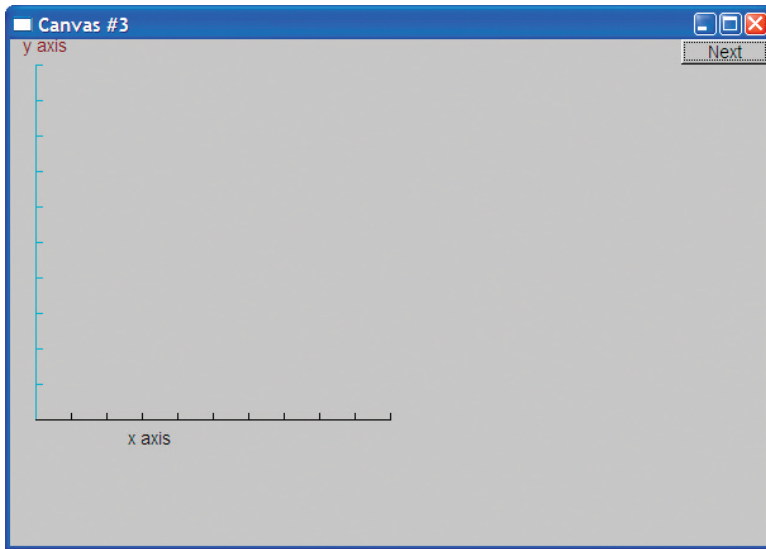
We can see that an `Axis::x` is a horizontal line. We see the required number of “notches” (10) and the label “x axis.” Usually, the label will explain what the axis and the notches represent. Naturally, we chose to place the x axis somewhere near the bottom of the window. In real life, we’d represent the height and width by symbolic constants so that we could refer to “just above the bottom” as something like `y_max-bottom_margin` rather than by a “magic constant,” such as `300` (§4.3.1, §15.6.2).

To help identify our output we relabeled the screen to `Canvas #2` using `Window`’s member function `set_label()`.

Now, let’s add a y axis:

```
Axis ya(Axis::y, Point(20,300), 280, 10, "y axis");
ya.set_color(Color::cyan);           // choose a color
ya.label.set_color(Color::dark_red); // choose a color for the text
win.attach(ya);
win.set_label("Canvas #3");
win.wait_for_button();               // display!
```

Just to show off some facilities, we colored our y axis cyan and our label dark red.



We don't actually think that it is a good idea to use different colors for x and y axes. We just wanted to show you how you can set the color of a shape and of individual elements of a shape. Using lots of color is not necessarily a good idea. In particular, novices tend to use color with more enthusiasm than taste.

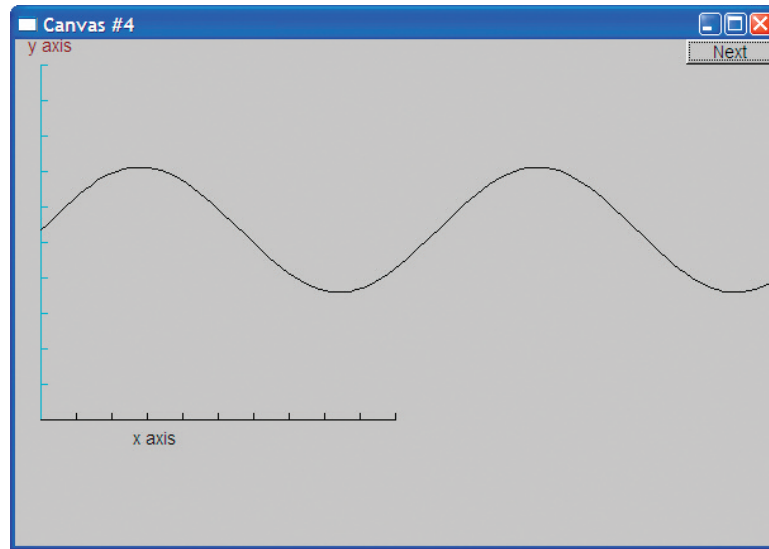
12.7.4 Graphing a function

What next? We now have a window with axes, so it seems a good idea to graph a function. We make a shape representing a sine function and attach it:

```
Function sine(sin,0,100,Point(20,150),1000,50,50);    // sine curve
    // plot sin() in the range [0:100) with (0,0) at (20,150)
    // using 1000 points; scale x values *50, scale y values *50

win.attach(sine);
win.set_label("Canvas #4");
win.wait_for_button();
```

Here, the **Function** named **sine** will draw a sine curve using the standard library function **sin()** to generate values. We explain details about how to graph functions in §15.3. For now, just note that to graph a function we have to say where it starts (a **Point**) and for what set of input values we want to see it (a range), and we need to give some information about how to squeeze that information into our window (scaling):



Note how the curve simply stops when it hits the edge of the window. Points drawn outside our window rectangle are simply ignored by the GUI system and never seen.

12.7.5 Polygons

A graphed function is an example of data presentation. We'll see much more of that in Chapter 15. However, we can also draw different kinds of objects in a window: geometric shapes. We use geometric shapes for graphical illustrations, to indicate user interaction elements (such as buttons), and generally to make our presentations more interesting. A **Polygon** is characterized by a sequence of points, which the **Polygon** class connects by lines. The first line connects the first point to the second, the second line connects the second point to the third, and the last line connects the last point to the first:

```
sine.set_color(Color::blue);    // we changed our mind about sine's color

Polygon poly;                   // a polygon; a Polygon is a kind of Shape
poly.add(Point(300,200));        // three points make a triangle
poly.add(Point(350,100));
poly.add(Point(400,200));

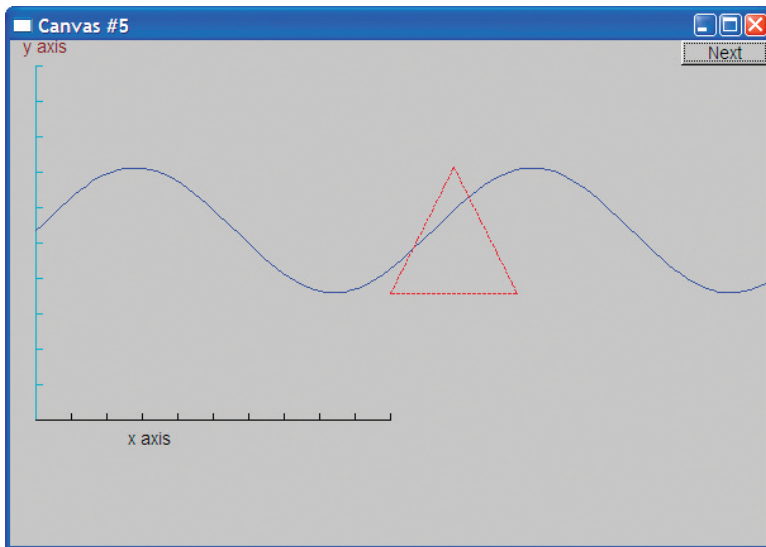
poly.set_color(Color::red);
```

```

poly.set_style(Line_style::dash);
win.attach(poly);
win.set_label("Canvas #5");
win.wait_for_button();

```

This time we change the color of the sine curve (**sine**) just to show how. Then, we add a triangle, just as in our first example from §12.3, as an example of a polygon. Again, we set a color, and finally, we set a style. The lines of a **Polygon** have a “style.” By default that is solid, but we can also make those lines dashed, dotted, etc. as needed (see §13.5). We get



12.7.6 Rectangles



A screen is a rectangle, a window is a rectangle, and a piece of paper is a rectangle. In fact, an awful lot of the shapes in our modern world are rectangles (or at least rectangles with rounded corners). There is a reason for this: a rectangle is the simplest shape to deal with. For example, it's easy to describe (top left corner plus width plus height, or top left corner plus bottom right corner, or whatever), it's easy to tell whether a point is inside a rectangle or outside it, and it's easy to get hardware to draw a rectangle of pixels fast.

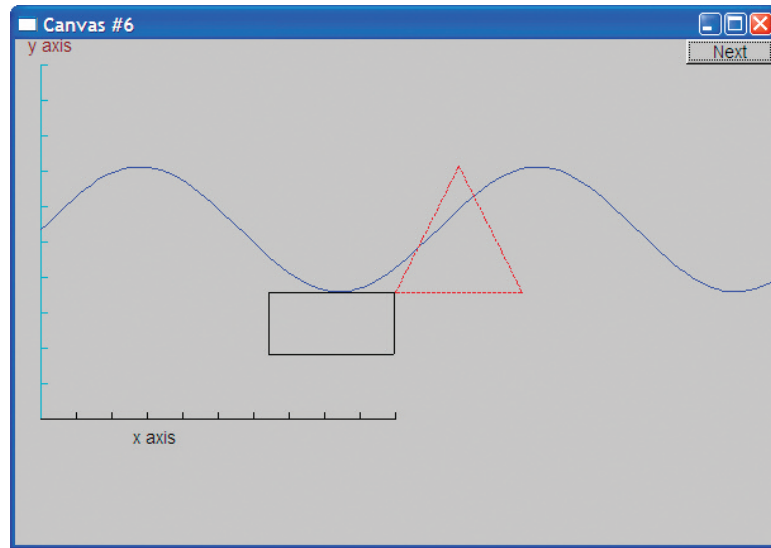
So, most higher-level graphics libraries deal better with rectangles than with other closed shapes. Consequently, we provide **Rectangle** as a class separate from the **Polygon** class. A **Rectangle** is characterized by its top left corner plus a width and height:

```

Rectangle r(Point(200,200), 100, 50); // top left corner, width, height
win.attach(r);
win.set_label("Canvas #6");
win.wait_for_button();

```

From that, we get

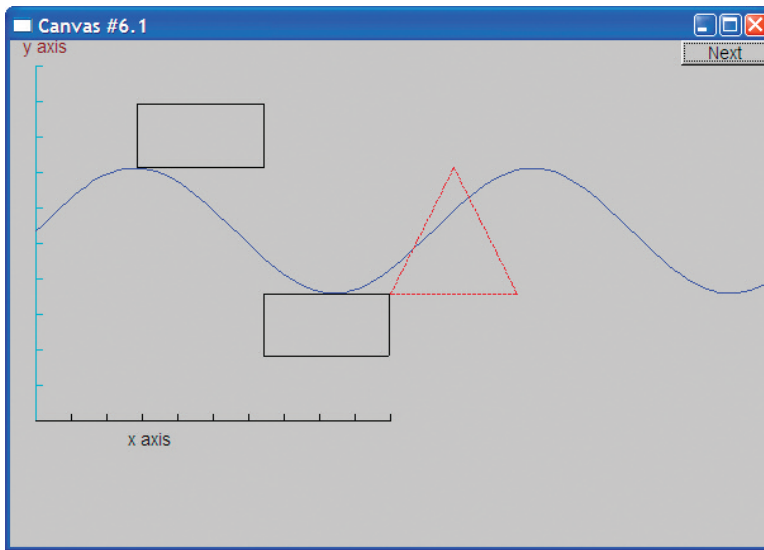


Please note that making a polyline with four points in the right places is not enough to make a **Rectangle**. It is easy to make a **Closed_polyline** that looks like a **Rectangle** on the screen (you can even make an **Open_polyline** that looks just like a **Rectangle**); for example:

```

Closed_polyline poly_rect;
poly_rect.add(Point(100,50));
poly_rect.add(Point(100,50));
poly_rect.add(Point(200,100));
poly_rect.add(Point(100,100));

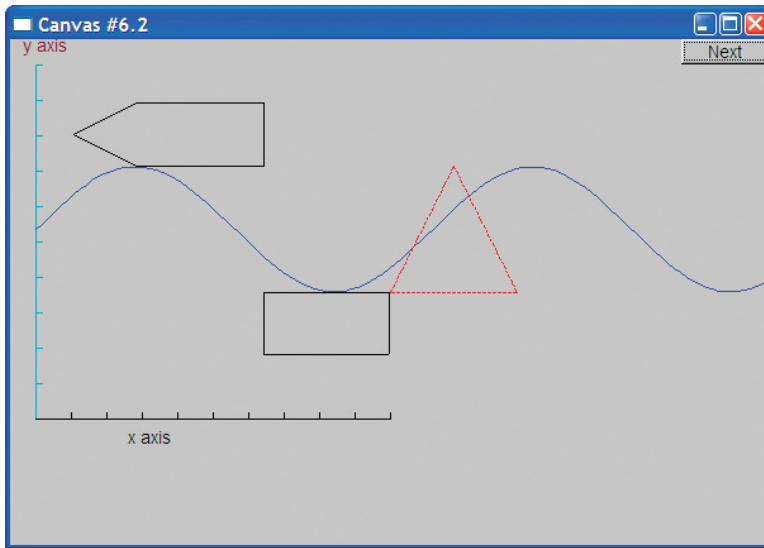
```



In fact, the *image* on the screen of such a **poly_rect** is a rectangle. However, the **poly_rect** object in memory is not a **Rectangle** and it does not “know” anything about rectangles. The simplest way to prove that is to add another point:

```
poly_rect.add(Point(50,75));
```

No rectangle has five points:



It is important for our reasoning about our code that a **Rectangle** doesn't just happen to look like a rectangle on the screen; it maintains the fundamental guar-

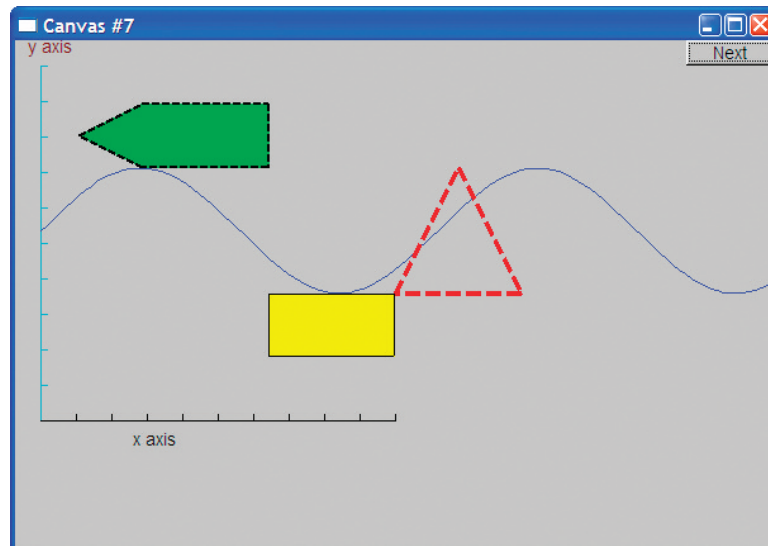
antees of a rectangle (as we know them from geometry). We write code that depends on a **Rectangle** really being a rectangle on the screen and staying that way.

12.7.7 Fill

We have been drawing our shapes as outlines. We can also “fill” a rectangle with color:

```
r.set_fill_color(Color::yellow);    // color the inside of the rectangle
poly.set_style(Line_style(Line_style::dash,4));
poly_rect.set_style(Line_style(Line_style::dash,2));
win.set_label("Canvas #7");
win.wait_for_button();
```

We also decided that we didn’t like the line style of our triangle (**poly**), so we set its line style to “fat (thickness four times normal) dashed.” Similarly, we changed the style of **poly_rect** (now no longer looking like a rectangle):



If you look carefully at **poly_rect**, you’ll see that the outline is printed on top of the fill.

It is possible to fill any closed shape (see §13.9). Rectangles are just special in how easy (and fast) they are to fill.

12.7.8 Text

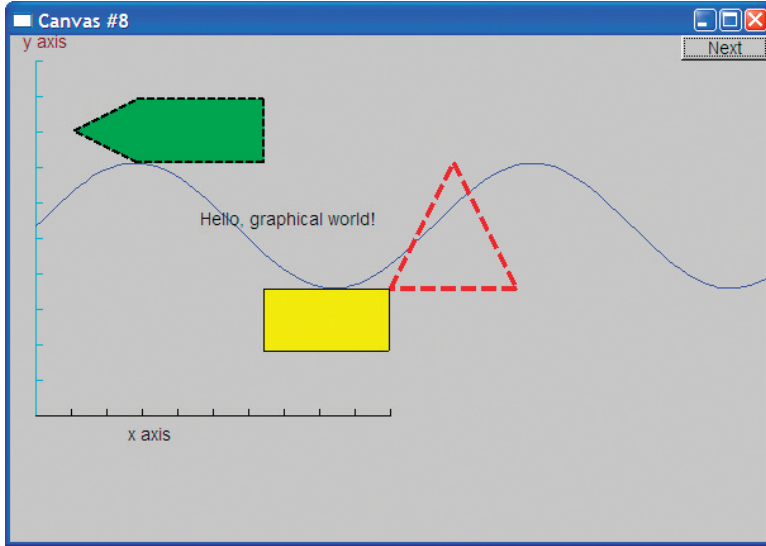
Finally, no system for drawing is complete without a simple way of writing text — drawing each character as a set of lines just doesn’t cut it. We label the window itself, and axes can have labels, but we can also place text anywhere using a **Text** object:




```

Text t(Point(150,150), "Hello, graphical world! ");
win.attach(t);
win.set_label("Canvas #8");
win.wait_for_button();

```



From the primitive graphics elements you see in this window, you can build displays of just about any complexity and subtlety. For now, just note a peculiarity of the code in this chapter: there are no loops, no selection statements, and all data was “hardwired” in. The output was just composed of primitives in the simplest possible way. Once we start composing these primitives using data and algorithms, things will start to get interesting.

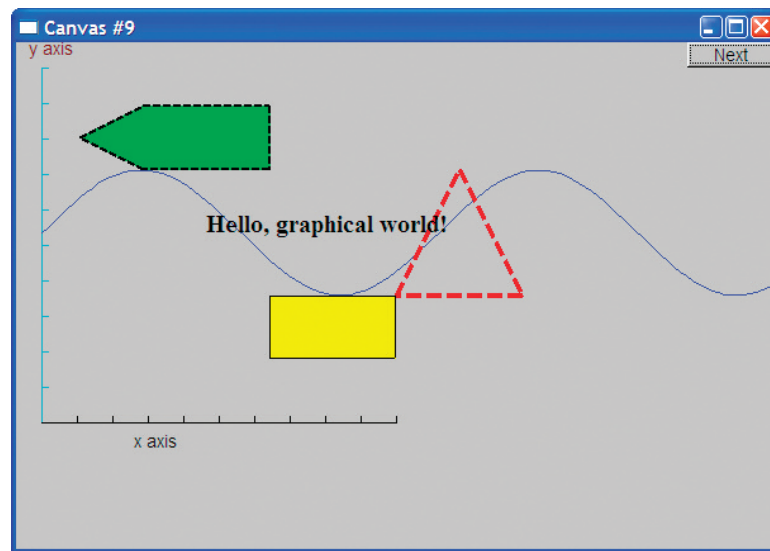
We have seen how we can control the color of text: the label of an **Axis** (§12.7.3) is simply a **Text** object. In addition, we can choose a font and set the size of the characters:

```

t.set_font(Font::times_bold);
t.set_font_size(20);
win.set_label("Canvas #9");
win.wait_for_button();

```

We enlarged the characters of the **Text** string **Hello, graphical world!** to point size 20 and chose the Times font in bold:

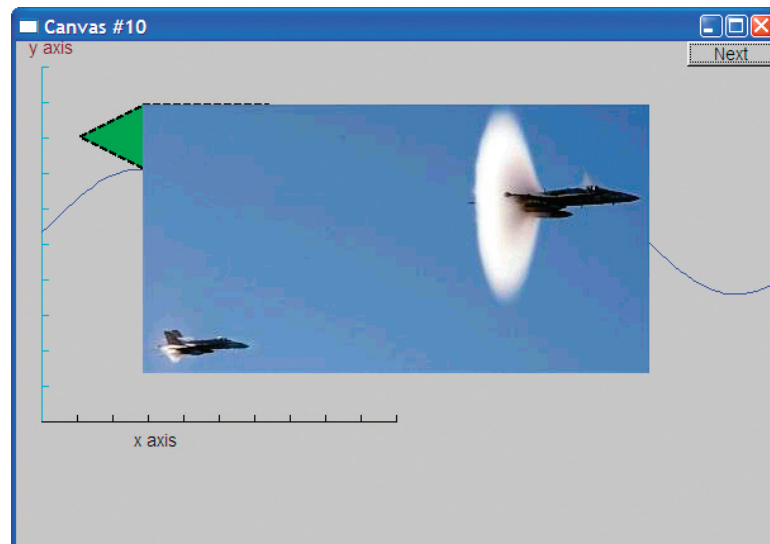


12.7.9 Images

We can also load images from files:

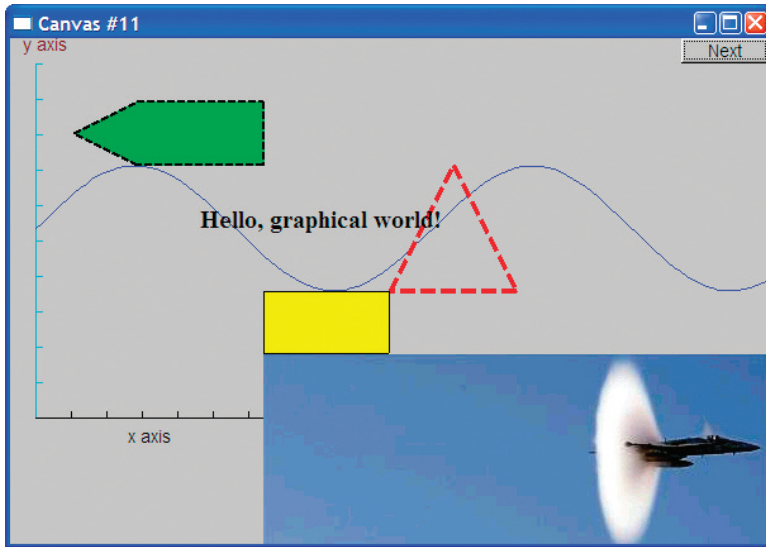
```
Image ii(Point(100,50),"image.jpg");    // 400*212-pixel jpg
win.attach(ii);
win.set_label("Canvas #10");
win.wait_for_button();
```

As it happens, the file called **image.jpg** is a photo of two planes breaking the sound barrier:



That photo is relatively large and we placed it right on top of our text and shapes. So, to clean up our window a bit, let us move it a bit out of the way:

```
ii.move(100,200);
win.set_label("Canvas #11");
win.wait_for_button();
```



Note how the parts of the photo that didn't fit in the window are simply not represented. What would have appeared outside the window is “clipped” away.

12.7.10 And much more

And here, without further comment, is some more code:

```
Circle c(Point(100,200),50);
Ellipse e(Point(100,200), 75,25);
e.set_color(Color::dark_red);
Mark m(Point(100,200),'x');

ostringstream oss;
oss << "screen size: " << x_max() << "*" << y_max()
    << "; window size: " << win.x_max() << "*" << win.y_max();
Text sizes(Point(100,20),oss.str());

Image cal(Point(225,225),"snow_cpp.gif"); // 320*240-pixel gif
cal.set_mask(Point(40,40),200,150);      // display center part of image
```

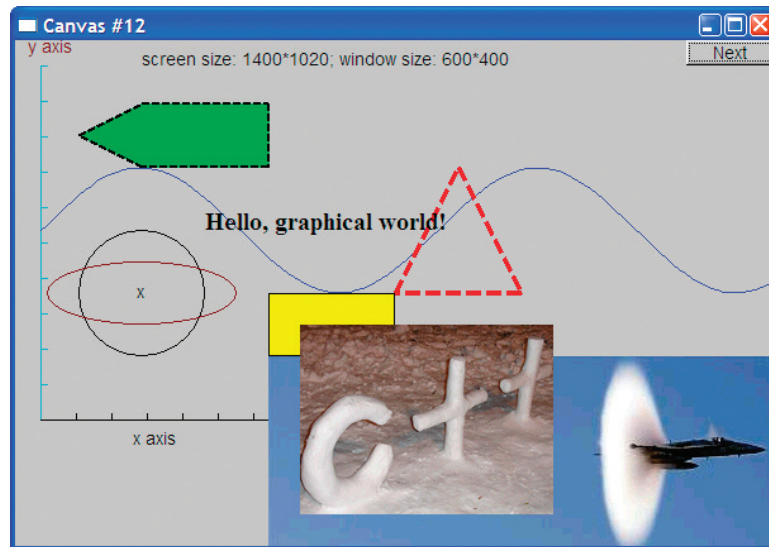
```

win.attach(c);
win.attach(m);
win.attach(e);

win.attach(sizes);
win.attach(cal);
win.set_label("Canvas #12");
win.wait_for_button();

```

Can you guess what this code does? Is it obvious?



The connection between the code and what appears on the screen is direct. If you don't yet see how that code caused that output, it soon will become clear. Note the way we used a **stringstream** (§11.4) to format the text object displaying sizes.



12.8 Getting this to run

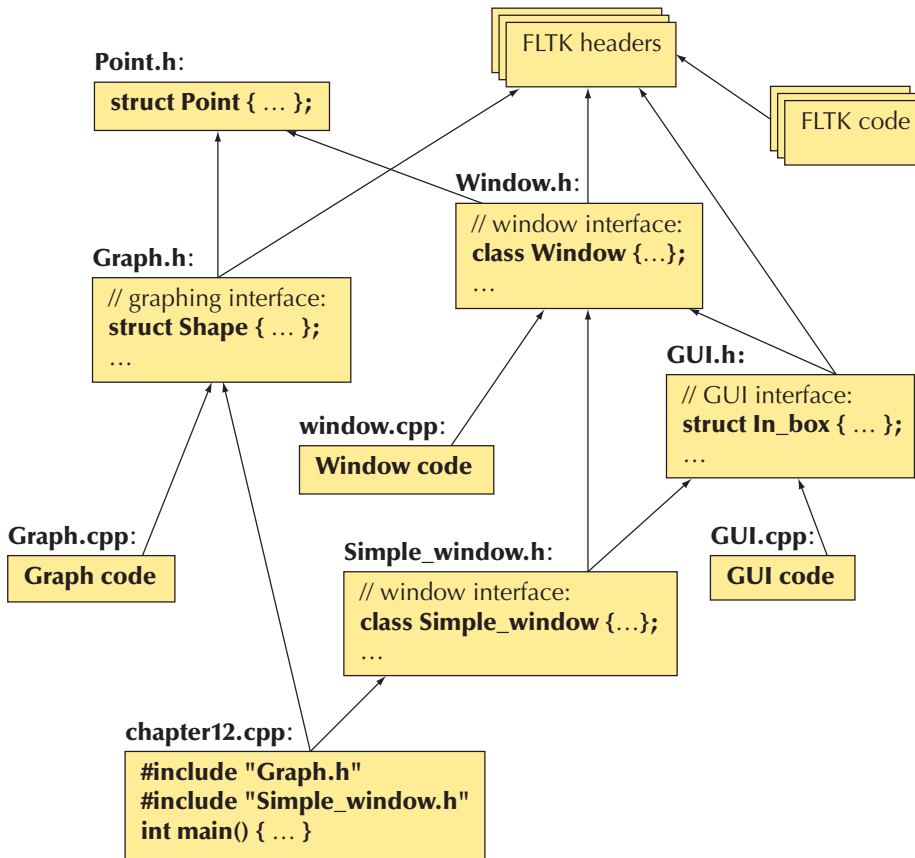
We have seen how to make a window and how to draw various shapes in it. In the following chapters, we'll see how those **Shape** classes are defined and show more ways of using them.

Getting this program to run requires more than the programs we have presented so far. In addition to our code in **main()**, we need to get the interface library code compiled and linked to our code, and finally, nothing will run unless the FLTK library (or whatever GUI system we use) is installed and correctly linked to ours.

One way of looking at the program is that it has four distinct parts:

- Our program code (`main()`, etc.)
- Our interface library (`Window`, `Shape`, `Polygon`, etc.)
- The FLTK library
- The C++ standard library

Indirectly, we also use the operating system. Leaving out the OS and the standard library, we can illustrate the organization of our graphics code like this:



Appendix D explains how to get all of this to work together.

12.8.1 Source files

Our graphics and GUI interface library consists of just five header files and three code files:

- Headers:
 - **Point.h**
 - **Window.h**
 - **Simple_window.h**
 - **Graph.h**
 - **GUI.h**
- Code files:
 - **Window.cpp**
 - **Graph.cpp**
 - **GUI.cpp**

Until Chapter 16, you can ignore the GUI files.



Drill

The drill is the graphical equivalent to the “Hello, World!” program. Its purpose is to get you acquainted with the simplest graphical output tools.

1. Get an empty **Simple_window** with the size 600 by 400 and a label **My window** compiled, linked, and run. Note that you have to link the FLTK library as described in Appendix D; **#include Graph.h**, **Window.h**, and **GUI.h** in your code; and include **Graph.cpp** and **Window.cpp** in your project.
2. Now add the examples from §12.7 one by one, testing between each added subsection example.
3. Go through and make one minor change (e.g., in color, in location, or in number of points) to each of the subsection examples.

Review

1. Why do we use graphics?
2. When do we try not to use graphics?
3. Why is graphics interesting for a programmer?
4. What is a window?
5. In which namespace do we keep our graphics interface classes (our graphics library)?
6. What header files do you need to do basic graphics using our graphics library?

7. What is the simplest window to use?
8. What is the minimal window?
9. What's a window label?
10. How do you label a window?
11. How do screen coordinates work? Window coordinates? Mathematical coordinates?
12. What are examples of simple “shapes” that we can display?
13. What command attaches a shape to a window?
14. Which basic shape would you use to draw a hexagon?
15. How do you write text somewhere in a window?
16. How would you put a photo of your best friend in a window (using a program you wrote yourself)?
17. You made a **Window** object, but nothing appears on your screen. What are some possible reasons for that?
18. You have made a shape, but it doesn't appear in the window. What are some possible reasons for that?

Terms

color	graphics	JPEG
coordinates	GUI	line style
display	GUI library	software layer
fill color	HTTP	window
FLTK	image	XML

Exercises

We recommend that you use **Simple_window** for these exercises.

1. Draw a rectangle as a **Rectangle** and as a **Polygon**. Make the lines of the **Polygon** red and the lines of the **Rectangle** blue.
2. Draw a 100-by-30 **Rectangle** and place the text “Howdy!” inside it.
3. Draw your initials 150 pixels high. Use a thick line. Draw each initial in a different color.
4. Draw a checkers board: 8-by-8 alternating white and red squares.
5. Draw a red $\frac{1}{4}$ -inch frame around a rectangle that is three-quarters the height of your screen and two-thirds the width.
6. What happens when you draw a **Shape** that doesn't fit inside its window? What happens when you draw a **Window** that doesn't fit on your screen? Write two programs that illustrate these two phenomena.
7. Draw a two-dimensional house seen from the front, the way a child would: with a door, two windows, and a roof with a chimney. Feel free to add details; maybe have “smoke” come out of the chimney.

8. Draw the Olympic five rings. If you can't remember the colors, look them up.
9. Display an image on the screen, e.g., a photo of a friend. Label the image both with a title on the window and with a caption in the window.
10. Draw the file diagram from §12.8.
11. Draw a series of regular polygons, one inside the other. The innermost should be an equilateral triangle, enclosed by a square, enclosed by a pentagon, etc. For the mathematically adept only: let all the points of each **N**-polygon touch sides of the **(N+1)**-polygon.
12. A superellipse is a two-dimensional shape defined by the equation

$$\left| \frac{x}{a} \right|^m + \left| \frac{y}{b} \right|^n = 1; \quad m, n > 0.$$

Look up *superellipse* on the web to get a better idea of what such shapes look like. Write a program that draws “starlike” patterns by connecting points on a superellipse. Take **a**, **b**, **m**, **n**, and **N** as arguments. Select **N** points on the superellipse defined by **a**, **b**, **m**, and **n**. Make the points equally spaced for some definition of “equal.” Connect each of those **N** points to one or more other points (if you like you can make the number of points connect to another argument or just use **N-1**, i.e., all the other points).

13. Find a way to add color to the superellipse shapes from the previous exercise. Make some lines one color and other lines another color or other colors.

Postscript

The ideal for program design is to have our concepts directly represented as entities in our program. So, we often represent ideas by classes, real-world entities by objects of classes, and actions and computations by functions. Graphics is a domain where this idea has an obvious application. We have concepts, such as circles and polygons, and we represent them in our program as class **Circle** and class **Polygon**. Where graphics is unusual is that when writing a graphics program, we also have the opportunity to see objects of those classes on the screen; that is, the state of our program is directly represented for us to observe – in most applications we are not that lucky. This direct correspondence between ideas, code, and output is what makes graphics programming so attractive. Please do remember, though, that graphics are just illustrations of the general idea of using classes to directly represent concepts in code. That idea is far more general and useful: just about anything we can think of can be represented in code as a class, an object of a class, or a set of classes.



Index

- `!. See` Not, 1050
- `!=. See` Not equal (inequality), 67, 1052, 1064
- `". . ". See` String literal, 62
- `#. See` Preprocessor directives, 1090–1091
- `$. See` End of line, 837, 1134
- `%. See`
 - Output format specifier, 1141
 - Remainder (modulo); 68
- `%=. See` Remainder and assign, 1053
- `&. See`
 - Address of, 574, 1050
 - Bitwise logical operations (and), 917, 1052, 1057
 - Reference to (in declarations), 273–277, 1062
- `&&. See` Logical and, 1053, 1057
- `&=. See` Bitwise logical operations (and and assign), 1053
- `' . . '. See` Character literals, 159, 1043–1044
- `() . See`
 - Expression (grouping), 95, 831, 837, 840
 - Function call, 282, 735–736
 - Function of (in declarations), 112–114, 1062
 - Regular expression (grouping), 1133
- `*. See`
 - Contents of (dereference), 579–580
 - Multiply, 1051
 - Pointer to (in declarations), 573, 1062
 - Repetition (in **regex**), 832, 837–838, 1133–1134
- `*/ end of block comment, 237`
- `*=. See` Multiply and assign (scale), 67
- `+. See`
 - Add, 66, 1051
 - Concatenation (of **strings**), 68–69, 815, 1132
 - Repetition in **regex**, 837–839, 1133–1134
- `++. See` Increment, 66, 695
- `+=. See`
 - Add and assign, 1053
 - Move forward, 1064
 - string** (add at end), 815, 1132
- `, (comma). See`
 - Comma operator, 1054
 - List separator, 1066, 1084
- `-. See`
 - Minus (subtraction), 66, 1051
 - Regular expression (range), 841
- `--. See` Decrement, 66, 1102, 1050
- `-> (arrow). See` Member access, 593, 1050–1051, 1072, 1102

- `--`. *See*
 - Move backward, 1064
 - Subtract and assign, 67, 1053, 1103
- `.` (dot). *See*
 - Member access, 302, 592–593, 1050–1051
 - Regular expression, 837, 1133
- `...` (ellipsis). *See*
 - Arguments (unchecked), 1068–1069
 - Catch all exceptions, 150
- `/`. *See* Divide, 66, 1051
- `//`. *See* Line comment, 45
- `/* . . */`. *See* Block comment, 237
- `/=`. *See* Divide and assign, 67, 1053
- `:` (colon). *See*
 - Base and member initializers, 310, 471, 543
 - Conditional expression, 266
 - Label, 104–107, 302, 502, 1059
- `::`. *See* Scope (resolution), 291, 310, 1049
- `;` (semicolon). *See* Statement (terminator), 50, 99
- `<`. *See* Less than, 67, 1052
- `<<`. *See*
 - Bitwise logical operations (left shift), 917, 1051
 - Output, 357–359, 1129
- `<=`. *See* Less than or equal, 67, 1052
- `<<=`. *See* Bitwise logical operations (shift left and assign), 1053
- `< . . >`. *See* Template (arguments and parameters), 151, 656–657
- `=`. *See*
 - Assignment, 66, 1053
 - Initialization, 69–73, 1173
- `==`. *See* Equal, 67, 1052
- `>`. *See*
 - Greater than, 67, 1052
 - Input prompt, 221
 - Template (argument-list terminator), 656–657
- `>=`. *See* Greater than or equal, 67, 1052
- `>>`. *See*
 - Bitwise logical operations (right shift), 917, 1051
 - Input, 61, 359
- `>>=`. *See* Bitwise logical operations (shift right and assign), 1053
- `?.` *See*
 - Conditional expression `?:`, 266, 1053
 - Regular expression, 831–832, 837, 838–839, 1134
- `[]`. *See*
 - Array of (in declaration), 627, 1062
 - Regular expression (character class), 837, 1133
 - Subscripting, 579–590, 628, 1064
- `\` (backslash). *See*
 - Character literal, 1043
 - Escape character, 1133
 - Regular expression (escape character), 830–831, 837, 841
- `^`. *See*
 - Bitwise logical operations (exclusive or), 917–918, 1052, 1057
 - Regular expression (not), 837, 1134
- `^=`. *See* Bitwise logical operations (xor and assign), 1053
- `_`. *See* Underscore, 75, 76, 1045
- `{}`. *See*
 - Block delimiter, 47, 110
 - Regular expression (range), 831, 837–839, 1133–1134
- `|`. *See*
 - Bitwise logical operations (bitwise or), 917, 1052, 1057
 - Regular expression (or), 831–832, 837, 840–841, 1134
- `|=`. *See* Bitwise logical operations (or and assign), 1053
- `||`. *See* Logical or, 1053, 1057
- `~`. *See*
 - Bitwise logical operations (complement), 917, 1050
 - Destructors, 586–588
- `0` (zero). *See*
 - Null pointer, 583–584
 - Prefix, 378, 380
 - `printf()` format specifier, 1142
- `0x`. *See* Prefix, 378, 380

A

- a**, append file mode, 1140
- `\a` alert, character literal, 1043
- `abort()`, 1149
- `abs()`, absolute value, 879, 1137
 - complex**, 881, 1139
- Abstract classes, 487, 1171
 - class hierarchies, 503
 - creating, 487, 503–504, 1080–1081
 - Shape** example, 487–488
- Abstract-first approach to programming, 10
- Abstraction, 92–93, 1171
 - level, ideals, 778–779
- Access control, 302, 496, 501–502
 - base classes, 501–502
 - encapsulation, 496
 - members, 484–485
 - private, 496, 501–502
 - private by default, 302–303
 - private *vs.* public, 302–304
 - private**: label, 302
 - protected, 496, 502
 - protected**: label, 502
 - public, 302, 496, 501–502
 - public by default, 303–304. *See also* **struct**.
 - public**: label, 302
 - Shape** example, 488–491
- `accumulate()`, 729, 739–740, 1139
 - accumulator, 739
 - generalizing, 740–742
- `acos()`, arccosine, 879, 1137
- Action, 47
- Activation record, 284. *See also* Stacks.
- Ad hoc polymorphism, 659–661
- Ada language, 796–798
- Adaptors
 - `bind1st()`, 1123
 - `bind2nd()`, 1123
 - container, 1106
 - function objects, 1123
 - `mem_fun()`, 1123
 - `mem_fun_ref()`, 1123
 - `not1()`, 1123
 - `not2()`, 1123
 - `priority_queue`, 1106
 - `queue`, 1106
 - `stack`, 1106
- `add()`, 445, 483–484, 600–602
- Add (plus) **+**, 66, 1051
- Add and assign **+=**, 66, 73, 1053
- Additive operators, 1051
- Address, 574, 1171
 - unchecked conversions, 905
- Address of (unary) **&**, 574, 1050
- `adjacent_difference()`, 739, 1139
- `adjacent_find()`, 1113
- `advance()`, 600–602, 708–710, 1103
- Affordability, software, 34
- Age distribution example, 527–528
- Alert markers, 3
- Algol family of languages, 791–798
- Algol60 language, 792–794
- <algorithm>**, 729, 1095
- Algorithms, 1171
 - and containers, 696
 - header files, 1095–1096
 - numerical, 1139
 - passing arguments to. *See* Function objects.
- Algorithms, numerical, 739, 1139
 - `accumulate()`, 729, 739–742, 1139
 - `adjacent_difference()`, 739, 1139
 - `inner_product()`, 729, 739, 742–744, 1139
 - `partial_sum()`, 739, 1139
- Algorithms, STL, 1112–1113
 - <algorithm>**, 729
 - `binary_search()`, 764
 - comparing elements, 729
 - `copy()`, 728, 757–758
 - `copy_if()`, 757
 - copying elements, 728
 - `count()`, 728
 - `count_if()`, 728
 - `equal()`, 729
 - `equal_range()`, 728, 763–764
 - `find()`, 728, 729–732
 - `find_if()`, 728, 732–734
 - heap, 1119–1120
 - `lower_bound()`, 764
 - `max`, 1121

Algorithms, STL (*continued*)

- merge()**, 728
- merging sorted sequences, 728
- min**, 1121
- modifying sequence, 1114–1116
- mutating sequence, 1114–1116
- nonmodifying sequence, 1113–1114
- numerical. *See* Algorithms, numerical.
- permutations, 1120
- search()**, 763–764
- searching, 1117–1118. *See also* **find()**; **find_if()**.
- set, 1118–1119
- shuffle, 1115–1116
- sort()**, 728, 762–763
- sorting, 728, 762–763, 1117–1118
- summing elements, 729
- testing, 961–968
- unique_copy()**, 728, 757, 760–761
- upper_bound()**, 764
- utility, 1116–1117
- value comparisons, 1120–1121

Aliases, 1089, 1171. *See also* References.

Allocating memory

See also Deallocating memory;
Memory.

- allocator_type**, 1108
- bad_alloc** exception, 1058
- C++ and C, 1009–1010
- calloc()**, 1147
- embedded systems, 897–898, 902–904
- free store, 578–579
- malloc()**, 1009, 1147
- new**, 1057–1058
- pools, 902–903
- realloc()**, 1010
- stacks, 903–904

allocator_type, 1108

Almost containers, 721–722, 1106

alnum, **regex** character class, 842, 1134

alpha, **regex** character class, 842, 1134

Alternation

- patterns, 192–193
- regular expressions, 840–841

Ambiguous function call, 1067–1068

Analysis, 35, 174, 177

and, synonym for **&**, 1003, 1004

and_eq, synonym for **&=**, 1003, 1004

app mode, 385, 1126

append(), 815, 1132

Append

- files, 385, 1140
- string +=**, 815

Application

- collection of programs, 1172
- operator **()**, 735–736

Approximation, 521–526, 1172

Arccosine, **acos()**, 879

Arcsine, **asin()**, 879

Arctangent, **atan()**, 879

arg(), of complex number, theta, 881, 1139

Argument deduction, 664–665

Argument errors

- callee responsibility, 141–143
- caller responsibility, 140–141
- reasons for, 142–143

Arguments, 270, 1172

- formal. *See* Parameters.

- functions, 1068–1069

- passing. *See* Passing arguments.

- program input, 91

- source of exceptions, 145–146

- templates, 1083–1084

- types, class interfaces, 319–321

- unchecked, 995–996, 1068–1069

- unexpected, 134

Arithmetic if **?:**, 266. *See also* Conditional expression.

Arithmetic operations. *See* Numerics.

array standard library class, 718–719, 1105

<array>, 1095

Arrays, 627–628, 1172

See also Containers; **vector**.

- []** declaration, 627

- []** dereferencing, 628

- accessing elements, 628, 863–865
 - assignment, 633
 - associative. *See* Associative containers.
 - built-in, 718–719
 - C-style strings, 633–634
 - copying, 632
 - dereferencing, 628
 - element numbering, 627
 - initializing, 582–583, 633–634
 - multidimensional, 859–861, 1065
 - palindrome example, 638–640
 - passing pointers to arrays, 905–912
 - pointers to elements, 628–631
 - range checking, 628
 - subscripting [], 628
 - terminating zero, 633
 - vector** alternative, 909–912
 - Arrays and pointers, 630–636
 - debugging, 634–637
 - asin()**, arcsine, 879, 1137
 - asm**, assembler insert, 1003
 - Assemblers, 785
 - Assertions
 - assert()**, 1026–1027
 - <cassert>**, 1097
 - debugging, 161
 - definition, 1172
 - assign()**, 1109
 - Assignment =, 69–73
 - arrays, 633
 - assignment and initialization, 69–73
 - composite assignment operators, 73–74
 - containers, 1108–1109
 - Date** example, 305–306
 - enumerators, 314
 - expressions, 1053
 - string**, 815
 - vector**, resizing, 653–655
 - Assignment operators (composite), 66
 - %=**, 73, 1053
 - &=**, 1053
 - *=**, 73, 1053
 - +=**, 73, 1053, 1103
 - =**, 73, 1053, 1103
 - /=**, 73, 1053
 - <<=**, 1053
 - >>=**, 1053
 - ^=**, 1053
 - |=**, 1053
 - Associative arrays. *See* Associative containers.
 - Associative containers, 744, 1105
 - email example, 820–824
 - header files, 744
 - map, 744
 - multimap**, 744, 824–825
 - multiset**, 744
 - operations, 1111–1112
 - set**, 744
 - unordered_map**, 744
 - unordered_multimap**, 744
 - unordered_multiset**, 744
 - unordered_set**, 744
 - Assumptions, testing, 976–978
 - at()**, range-checked subscripting, 668–669, 1109
 - atan()**, arctangent, 879, 1137
 - ate** mode, 385, 1126
 - atof()**, string to **double**, 1146
 - atoi()**, string to **int**, 1146
 - atol()**, string to **long**, 1146
 - AT&T Bell Labs, 803
 - AT&T Labs, 803
 - attach()** *vs.* **add()** example, 483–484
 - Automatic storage, 577
 - auto_ptr**, 678
 - Axis** example, 420–422, 439, 518–521, 532–534
- ## B
- b**, binary file mode, 1140
 - Babbage, Charles, 797
 - back()**, last element, 708, 1109
 - back_inserter()**, 1122
 - Backus, John, 788
 - Backus-Naur (BNF) Form, 788, 793
 - bad()** stream state, 349, 1127

- bad_alloc** exception, 1058
- Balanced trees, 748–750
- Base-2 number system (binary), 1042
- Base-8 number system (octal), 1041–1042
- Base-10
 - logarithms, 879
 - number system (decimal), 1041–1042
- Base-16 number system (hexadecimal), 1041–1042
- Base and member initializers, 310, 471, 543
- Base classes, 485–488, 496–499, 1172
 - abstract classes, 487, 503–504, 1080–1081
 - access control, 501–502
 - derived classes, 1078–1079
 - description, 496–497
 - initialization of, 417, 543
 - interface, 503–505
 - object layout, 497–499
 - overriding, 500–501
 - Shape** example, 487–488
 - virtual function calls, 493, 498–499
 - vptr**, 498
 - vtbl**, 498
- Base-e exponentials, 879
- Basic guarantee, 677
- basic_string**, 816
- BCPL language, 803
- begin()**
 - iterator, 1109
 - numeric example, 121–122
 - string**, 815, 1132
 - vector**, 695
- Bell Telephone Laboratories (Bell Labs), 801, 803–806, 988–989
- Bentley, John, 895, 926
- Bidirectional iterator, 1104
- bidirectional** iterators, 722–723
- Big-O notation, complexity, 573
- Binary I/O, 386–389
- binary** mode, 385, 1126
- Binary number system, 1042
- Binary search, 728, 747, 763–764
- binary_search()**, 764, 1117
- bind1st()** adaptor, 1123
- bind2nd()** adaptor, 1123
- bitand**, synonym for **&**, 1003, 1004
- Bitfields, 917, 928–930, 1082
- bitor**, synonym for **|**, 1003, 1004
- Bits, 78, 916, 1172
 - bitfields, 917
 - bool**, 917
 - char**, 917
 - enumerations, 917
 - integer types, 917
 - manipulating, 926–928
 - signed, 922–926
 - size, 916–917
 - two's complement, 922
 - unsigned**, 922–926
- <bitset>**, 1095
- bitset**, 920–922
 - bitwise logical operations, 922
 - construction, 921
 - exceptions, 1099
 - I/O, 922
- Bitwise logical operations, 917–920, 1057
 - and **&**, 917–918, 1052, 1057
 - and and assign **&=**, 1053
 - complement **~**, 917
 - exclusive or **^**, 917–918, 1052, 1057
 - exclusive or and assign **^=**, 1053
 - left shift **<<**, 917
 - left shift and assign **<<=**, 1053
 - or **|**, 917–918, 1052, 1057
 - or and assign, **|=**, 927
 - right shift **>>**, 917
 - right shift and assign **>>=**, 1053
- Black-box testing, 952–953
- Blackboard, 36
- blank**, character class, **regex**, 842, 1134
- Block, 110
 - debugging, 159
 - delimiter **{}**, 47, 110
 - nesting within functions, 268–269
 - try** block, 144–145

Block comment `/* . . */`, 237
 Blue marginal alerts, 3
 BNF (Backus-Naur) Form, 788, 793
 Body, functions, 113
bool, 63, 66–67, 1062
 bit space, 917
 bits in memory, 78
 C++ and C, 992, 1003, 1004
 size, 78
boolalpha, manipulator, 1129
 Boolean conversions, 1055
 Borland, 796
 Bottom-up approach, 9, 776–777
 Bounds error, 147
 Branching, testing, 966–968. *See also*
 Conditional statements.
break, **case** label termination, 104–107
 Broadcast functions, 867
bsearch(), 1149
 Buffer, 342
 flushing, 239–240
 iostream, 402
 overflow, 639, 759, 966. *See also*
 gets(), **scanf()**.
 Bugs, 156, 1172
 See also Debugging; Testing.
 finding the last, 164–165
 first documented, 790
 regression tests, 953
 Built-in types, 300, 1062
 arrays, 718–719, 1064–1065
 bool, 77, 1063
 characters, 77, 855, 1063
 default constructors, 323
 exceptions, 1087
 floating-point, 77, 855–858, 1063
 integers, 77, 855–858, 922–926, 1063
 pointers, 574–586, 1063–1064
 references, 277–278, 1065–1066
Button example, 439, 548–550
 attaching to menus, 558
 detecting a click, 544–546
 “Next,” 418–420, 541–542
 Byte, 78, 1172
 operations, C-style strings, 1014–1015

C

.c suffix, 995
.cpp, suffix, 48, 1154
 C# language, 796
 C++ language, 804–806
 See also Programming; Programs;
 Software.
 coding standards, list of, 943
 portability, 11
 use for teaching, xxiv, 6–9
 C++ and C, 988–990
 C functions, 994–998
 C linkage convention, 999
 C missing features, 991–993
 calling one from the other, 998–1000
 casts, 1006–1007
 compatibility, 990–991
 const, 1020–1021
 constants, 1020–1021
 container example, 1025–1031
 definitions, 1004–1006
 enum, 1008
 extern "C", 999
 family tree, 989
 free-store, 1009–1011
 input/output, 1016–1020
 keywords, 1003–1004
 layout rules, 1000
 macros, 1020–1025
 malloc(), 1009
 namespaces, 1008
 nesting **structs**, 1003
 old-style casts, 1006
 opaque types, 1026
 performance, 990
 realloc(), 1010
 structure tags, 1002–1003
 type checking, 998–999
 void, 996
 void*, 1007–1008
 “C first” approach to programming, 9
 C language, 800–804
 See also C standard library.
 C++ compatibility, 988–990. *See also*
 C++ and C.

- C language (*continued*)
 - K&R, 802, 988–989
 - linkage convention, 999
 - missing features, 991–993
- C standard library
 - C-style strings, 1145–1146
 - header files, 1097
 - input/output. *See* C-style I/O (stdio).
 - memory, 1146–1147
- C-style casts, 1006–1007, 1051, 1058
- C-style I/O (stdio)
 - `%`, conversion specification, 1141
 - conversion specifications, 1141–1143
 - file modes, 1140–1141
 - files, opening and closing, 1140–1141
 - `fprintf()`, 1017, 1141
 - `getch()`, 1018, 1145
 - `getchar()`, 1010, 1017–1019, 1145
 - `gets()`, 1018, 1144–1145
 - output formats, user-defined types, 1144
 - padding, 1143
 - `printf()`, 1016–1017, 1141
 - `scanf()`, 1017–1019, 1144–1145
 - `stderr`, 1144
 - `stdin`, 1144
 - `stdout`, 1144
 - truncation, 1143
- C-style strings, 633–634, 1011–1013, 1145
 - byte operations, 1014–1015
 - from `string`, `c_str()`, 344, 815
 - `const`, 1013–1014
 - copying, 1012–1013, 1015
 - executing as a command, `system()`, 1149
 - lexicographical comparison, 1012
 - operations, 1146
 - pointer declaration, 1015–1016
 - `strcat()`, concatenate, 1012–1013
 - `strchr()`, find character, 1014
 - `strcmp()`, compare, 1011–1013
 - `strcpy()`, copy, 1012–1013, 1015
 - `strlen()`, length of, 1012
 - `strncat()`, 1012–1013
 - `strncmp()`, 1012–1013
 - `strncpy()`, 1012–1013
 - three-way comparison, 1012
- CAD/CAM, 27, 33
- Calculator example, 172, 185–186
 - analysis and design, 174–177
 - `expression()`, 194–198
 - `get_token()`, 194
 - grammars and programming, 186–193
 - parsing, 188–191
 - `primary()`, 194, 206
 - symbol table, 246
 - `term()`, 194, 195–200, 204–205
 - Token, 182–183
 - `Token_stream`, 204–212, 239–240
- Call stack, 287
- Callback functions, 544–546
- Callback implementation, 1162–1163
- Calling functions. *See* Function calls.
- `calloc()`, 1147
- Cambridge University, 803
- `capacity()`, 651–652, 1111
- Capital letters. *See* Case.
- Case (of characters)
 - formatting, 393–394
 - identifying, 393
 - `islower()`, 393, 1131
 - `map` container, 750
 - in names, 74–77
 - sensitivity, 393–394
 - `tolower()`, changing case, 394, 1131
 - `toupper()`, changing case, 394, 1131
- `case` labels, 104–107
- `<cassert>`, 1097
- Casting away `const`, 594–595
- Casts
 - See also* Type conversion.
 - C++ and C, 992, 1003
 - C-style casts, 1006–1007
 - casting away `const`, 594
 - `const_cast`, 1058
 - `dynamic_cast`, 894, 1058

- lexical_cast** example, 819
- narrow_cast** example, 151
- reinterpret_cast**, 594
- static_cast**, 594, 905, 1058
 - unrelated types, 594
- CAT scans, 30
- catch**, 145, 1003
- Catch all exceptions ..., 150
- Catching exceptions, 144–150, 238–240, 1087
- cb_next()** example, 544–546
- <cctype>**, 1097, 1131
- ceil()**, 879, 1137
- cerr**, 149, 1125, 1144
- <cerrno>**, 1097
- <cmath>**, 1097
- Chaining operations, 178–179
- char** type, 63, 66–67, 78
 - bits, 917
 - built-in, 1062
 - properties, 712–713
 - signed** *vs.* **unsigned**, 858, 925
- Character classes
 - list of, 1134–1135
 - in regular expressions, 837–838, 842
- Character classification, 393–394, 1131
- Character literals, 159, 1043–1044
- CHAR_BIT** limit macro, 1136
- CHAR_MAX** limit macro, 1136
- CHAR_MIN** limit macro, 1136
- cin**, 61
 - C equivalent. *See* **stdin**.
 - standard character input, 61, 341, 1125
- Circle** example, 464–467, 489
 - vs.* **Ellipse**, 467
- Circular reference. *See* Reference (circular).
- class**, 181, 1002–1003
- Class
 - abstract, 487, 503–504, 1080–1081. *See also* Abstract class.
 - base, 496–497
 - coding standards, 941–942
 - concrete, 487–488, 1172
 - const** member functions, 1073
 - constructors, 1075–1077, 1081
 - copying, 1077–1078, 1081
 - creating objects. *See* Concrete classes.
 - default constructors, 322–325
 - defining, 210, 301, 1071, 1172
 - derived, 496
 - destructors, 1077, 1081
 - encapsulation, 496
 - friend** declaration, 1073–1074
 - generated operations, 1081
 - grouping related, 503–504
 - hierarchies, 503
 - history of, 799
 - implementation, 302–304
 - inheritance, 496–497, 504–505
 - interface, 504–505
 - member access. *See* Access control.
 - naming. *See* Namespaces.
 - nesting, 268
 - object layout, 497–499
 - organizing. *See* Namespaces.
 - parameterized, 659–661. *See also* Templates.
 - private**, 302–304, 496, 501–502, 1071–1072
 - protected**, 487, 496, 501–502
 - public**, 302–304, 496, 501–502, 1071–1072
 - run-time polymorphism, 496
 - subclasses, 496–497. *See also* Derived class.
 - superclasses, 496–497. *See also* Base class.
 - templates, 658–661
 - testing, 973–976
 - this** pointer, 1073
 - types as parameters. *See* Templates.
 - union**, 1082–1083
 - unqualified name, 1072
 - uses for, 301
- Class interfaces, 318, 1071
 - argument types, 319–321

- const** member functions, 325–326
- constants, 325–326. *See also* **const**.
- copying, 321–322
- helper functions, 326–328
- immutable values, 325, 326
- initializing objects, 322–325
- members, 326–328
- mutable values, 326–328
- public *vs.* private, 302–304
- symbolic constants, defining, 321
- uninitialized variables, 322–325
- Class members, 301, 1071
 - (arrow), 1072
 - .
 - .. (scope resolution), 1072
 - accessing, 302. *See also* Access control.
 - allocated at same address, 1082–1083
 - bitfields, 1082
 - class interfaces, 326–328
 - data, 301
 - definitions, 1074–1075
 - function, 309–313
 - in-class definition, 1074–1075
 - static const int** members, 1075
 - Token** example, 181–182
 - Token_stream** example, 210
 - out-of-class definition, 1074–1075
- Class scope, 264, 1046
- Class template
 - parameterized class, 659–661
 - parameterized type, 659–661
 - specialization, 658–659
 - type generators, 658–659
- classic_elimination()** example, 874–875
- Cleaning up code
 - comments, 236–237
 - functions, 233–234
 - layout, 234–236
 - logical separations, 233–234
 - revision history, 236–237
 - scaffolding, 233–234
 - symbolic constants, 231–233
- clear()**, 349–352, 1110
- <climits>**, 1097
- <locale>**, 1097
- clock()**, 981–983
- clock_t**, 1147
- clone()** example, 496
- close()** file, 346
- Closed_polyline** example, 451–453
 - vs.* **Polygon**, 453
- <cmath>**, 879, 1097, 1137
- cntrl**, 842, 1134
- COBOL language, 788–790
- Code
 - definition, 1172
 - layout, cleaning up, 234–236
 - libraries, uses for, 175
 - storage, 577
 - structure, ideals, 776
 - test coverage, 968
- Coding standards, 935–936
 - C++, list of, 943
 - complexity, sources of, 935–936
 - ideals, 936–937
 - sample rules, 938–943
- Color** example, 421–422, 445–447
 - color chat example, 459–461
 - fill, 427–428, 456–458, 492
 - transparency, 447
- Columns, matrices, 864–865, 870
- Comments, 45–46
 - block **/*...*/**, 237, 1040
 - C++ and C, 992
 - cleaning up, 236–237
 - vs.* code, 237
 - line **//**, 45–46, 1040
 - role in debugging, 157–158
- Common Lisp language, 790
- Communication skills, programmers, 22
- Compacting garbage collection, 900–901
- Comparison, 67
 - See also* **==**; **<**.
 - C-style strings, 1011–1012
 - characters, 711
 - containers, 1111
 - key_compare**, 1108
 - lexicographical, C-style strings, 1012

- `lexicographical_compare()`, 1121
- `min/max` algorithms, 1120–1121
- `string`, 815
- three-way, 1012
- Compatibility. *See* C++ and C.
- Compile-time errors. *See* Errors, compile-time.
- Compiled languages, 47–48
- Compilers, 48, 1172
 - compile-time errors, 51
 - conditional compilation, 1024–1025
 - syntax checking, 48–50
- compl**, synonym for `~`, 1003, 1045
- complex**
 - `!=`, not equal (inequality), 881, 1138
 - `*`, multiply, 881, 1138
 - `+`, add (plus), 881, 1138
 - `-`, subtract (minus), 881, 1138
 - `<<`, output, 881, 1139
 - `==`, equal, 881, 1138
 - `>>`, input, 881, 1139
 - `/`, divide, 881, 1138
 - `abs()`, absolute value, 881, 1139
 - `conj()`, conjugate, 881
 - Fortran language, 882
 - `imag()`, imaginary part, 881
 - `norm()`, square of `abs()`, 881
 - number types, 1138–1139
 - `polar()`, polar coordinate, 881
 - `real()`, real part, 881
 - rho, 881
 - square of `abs()`, 881
 - theta, 881
- <complex>** 1096
 - complex** operators, 881, 1138–1139
 - standard math functions, 1137
- Complex numbers, 880–882
- Complexity, 1172
 - sources of, 935–936
- Composite assignment operators, 73–74
- Compound statements, 110
- Computation, 91
 - See also* Programs; Software.
 - correctness, 92–94
 - data structures, 90
 - efficiency, 92–94
 - input/output, 91
 - objectives, 92–94
 - organizing programs, 92–94
 - programmer ideals, 92–94
 - simplicity, 92–94
 - state, definition, 90
- Computation *vs.* data, 691–693
- Computer-assisted surgery, 30
- Computer science, 12, 24–25
- Computers
 - CAT scans, 30
 - computer-assisted surgery, 30
 - in daily life, 19–21
 - information processing, 31–32
 - Mars Rover, 32–33
 - medicine, 30
 - pervasiveness of, 19–21
 - server farms, 31
 - shipping, 26–28
 - space exploration, 32–33
 - telecommunications, 28–29
 - timekeeping, 26
 - world total, 19
- Concatenation of **strings**, 66
 - `+`, 68–69, 815, 1132
 - `+=`, 68–69, 815, 1132
- Concept-based approach to programming, 6
- Concrete classes, 487–488, 1172
- Concrete-first approach to programming, 6
- Concurrency, 894
- Conditional compilation, 1024–1025
- Conditional expression **?:**, 266, 1053
- Conditional statements
 - See also* Branching, testing.
 - for**, 110–112
 - if**, 101–103
 - switch**, 104–107
 - while**, 108–109
- Conforming programs, 1039
- Confusing variable names, 77
- conj()**, complex conjugate, 881, 1138
- Conjugate, 881

- Consistency, ideals, 780
- Console, as user interface, 540
- Console input/output, 540
- Console window, displaying, 160
- const**, 95–96
 - See also* Constant; Static storage, **static const**.
 - C++ and C, 992, 1020–1021
 - C-style strings, 1013–1014
 - class interfaces, 325–326
 - declarations, 260–261
 - initializing, 260
 - member functions, 325–326, 1073
 - overloading on, 626–627
 - passing arguments by, 273–276, 279–281
 - type, 1062
- *const**, immutable pointer, 1062
- Constant
 - See also* **const**.
 - expressions, 1056–1057
- const_cast**, casting away **const**, 594, 1058
- const_iterator**, 1108
- Constraints, **vector** range checking, 670
- Constructors, 306–309, 1075–1077
 - See also* Destructors; Initializers.
 - containers, 1108–1109
 - copy, 614–616, 620–624
 - Date** example 307, 319–321
 - debugging, 622–624
 - default, 322–325, 1081
 - error handling 309, 675–677
 - essential operations, 620–624
 - exceptions, 675–677
 - explicit**, 621–622
 - implicit conversions, 621–622
 - initialization of bases and members, 310, 471, 543
 - invariant, 309, 676–677
 - need for default, 620–621
 - Token** example, 182–183
 - Token_stream** example, 210
- Container adaptors, 1106
- Containers, 146, 720–721, 1172
 - See also* Arrays; **list**; **map**; **vector**.
 - and algorithms, 696
 - almost containers, 721–722, 1106
 - assignments, 1108–1109
 - associative, 1105, 1111–1112
 - capacity()**, 1110–1111
 - of characters. *See* **string**.
 - comparing, 1111
 - constructors, 1108–1109
 - contiguous storage, 712
 - copying, 1111
 - destructors, 1108–1109
 - element access, 1109
 - embedded systems, 912–916
 - header files, 1095–1096
 - information sources about, 720–721
 - iterator categories, 722–723
 - iterators, 1109
 - list operations, 1110
 - member types, 1108
 - operations overview, 1107
 - queue operations, 1110
 - sequence, 1105
 - size()**, 1110–1111
 - stack operations, 1110
 - standard library, 1105–1111
 - swapping, 1111
 - templates, 661–662
- Contents of ***** (dereference, indirection), 579–580
- Contiguous storage, 712
- Control characters, **isctrl()**, 393
- Control inversion, GUIs, 556–557
- Control variables, 109
- Controls. *See* Widgets.
- Conversion specifications, **printf()**, 1141–1143
- Conversion
 - See also* Type conversion.
 - character case, 394
 - representation, 368–370
 - unchecked, 905
- Coordinates
 - See also* **Point**.
 - computer screens, 415–416
 - graphs, 422–423
- copy()**, 757–758, 1114
- Copy assignments, 616–618, 620–624

Copy constructors, 614–616, 620–624
copy_backward(), 1114
copy_if(), 757
 Copying, 613–619
 arrays, 632
 C-style strings, 1012–1013, 1015
 class interfaces, 321–322
 containers, 1111
 I/O streams, 758–761
 objects, 494–496
 sequences, 728, 757–762
 vector, 613–618, 1108–1109
 Correctness
 definition, 1172
 ideals, 92–94, 775
 importance of, 891–892
 software, 34
cos(), cosine, 517–518, 879, 1137
cosh(), hyperbolic cosine, 1137
 Cost, definition, 1172
count(), 728, 1113
count_if(), 728, 1113
cout, 45
 C equivalent. *See* **stdout**.
 “Hello, World!” example, 45–46
 printing error messages, 149. *See also* **cerr**.
 standard output, 341, 1125
 Critical systems, coding standards,
 942–943
<cstdlib>, 1097
<cstdio>, 1097
<cstdlib>, 1097, 1147, 1149
c_str(), 1132
<cstring>, 1097, 1131, 1147
<ctime>, 1097, 1147
 Current object, 312–313. *See also* **this**
 pointer.
 Cursor, definition, 45
<wchar>, 1097
<wctype>, 1097

D

d, any decimal digit, **regex**, 842, 1134
\d, decimal digit, **regex**, 837, 1135

\D, not a decimal digit, **regex**, 838, 1135
d suffix, 1042
 Dahl, Ole-Johan, 798–800
 Data
 See also Containers; Sequences;
 vector; **map**; **list**.
 abstraction, 781
 collections. *See* Containers.
 vs. computation, 691–693
 generalizing code, 688–690
 in memory. *See* Free store.
 processing, overview, 686–690
 separating from algorithms, 696
 storing. *See* Containers.
 structure. *See* Containers; **struct**;
 class.
 traversing. *See* Iteration; Iterators.
 uniform access and manipulation,
 688–690. *See also* STL.
 Data member, 301, 484–485
 Data structure. *See* Data; **struct**.
 Data type. *See* Type.
 Date and time, 1147–1149
Date example. *See* Chapters 6–7.
DBL_EPSILON limit macro, 1137
DBL_MAX limit macro, 1137
DBL_MIN limit macro, 1137
 Deallocating memory, 584–586, 1057–
 1058. *See also* **delete**; **delete[]**.
 Debugging, 52, 156, 1172
 See also Errors; Testing.
 arrays and pointers, 634–637
 assertions, 161
 block termination, 159
 bugs, 156
 character literal termination, 159
 commenting code, 157–158
 compile-time errors, 159
 consistent code layout, 158
 constructors, 622–624
 declaring names, 159
 displaying the console window,
 160
 expression termination, 159
 finding the last bug, 164–165
 function size, 158

- Debugging (*continued*)
 - GUIs, 562–564
 - input data, 164
 - invariants, 160–161
 - keeping it simple, 158
 - logic errors, 152–154
 - matching parentheses, 159
 - naming conventions, 158
 - post-conditions, 163–164
 - pre-conditions, 161–163
 - process description, 156–157
 - reporting errors, 157
 - stepping through code, 160
 - string literal termination, 159
 - systematic approach, 164–165
 - test cases, 164, 225
 - testing, 979
 - tracing code execution, 160–161
 - transient bugs, 581
 - using library facilities, 158
 - widgets, 563–564
- dec** manipulator, 378–379, 1130
- Decimal digits, **isdigit()**, 393
- Decimal integer literals, 1041
- Decimal number system, 377–379, 1041–1042
- Deciphering (decryption), example, 930–935
- Declaration operators, 1062
 - &** reference to, 273–277, 1062
 - ()** function of, 112–114, 1062
 - *** pointer to, 573, 1062
 - []** array of, 627, 1062
- Declarations, 51, 1061–1062
 - C++ and C, 992
 - classes, 302
 - collections of. *See* Header files.
 - constants, 260–261
 - definition, 51, 77, 255, 1173, 1061–1062
 - vs.* definitions, 257–258
 - entities used for, 259
 - extern** keyword, 257
 - forward, 259
 - function, 255–256, 1066
 - function arguments, 270–271
 - function return type, 270–271
 - grouping. *See* Namespaces.
 - managing. *See* Header files.
 - need for, 259
 - order of, 213–214
 - parts of, 1061
 - subdividing programs, 258–259
 - “undeclared identifier” errors, 256
 - uses for, 1061
 - variables, 258, 260–261
- Decrementing **--**, 97–98
 - iterator, 1101–1104
 - pointer, 630
- Deep copy, 619
- Default constructors, 323–324
 - alternatives for, 324–325
 - for built-in types, 323
 - initializing objects, 322–323
 - need for, identifying, 620–621
 - uses for, 323–324
- #define**, 1090–1091
- Definitions. 77, 256–257, 1173
 - See also* Declarations.
 - C++ and C, 1004–1006
 - vs.* declarations, 257–258
 - function, 112–114, 270–271
- delete**
 - C++ and C, 992, 1003
 - deallocating free store, 1057–1058
 - destructors, 586–590
 - embedded systems, 894, 898–901, 901–902
 - free-store deallocation, 584–586
 - in unary expressions, 1051
- delete[]**, 585, 1051, 1057–1058
- Delphi language, 796
- Dependencies, testing, 962–963
- Depth-first approach to programming, 6
- deque**, double ended queue, 1105
- <deque>**, 1095
- Dereference/indirection
 - ***, 579–580. *See also* Contents of.
 - >**, 593. *See also* Member access.
 - []**, 116–117. *See also* Subscripting.
- Derivation, classes, 496

- Derived classes, 496, 1173
 - access control, 501–502
 - base classes, 1078–1079
 - inheritance, 1078–1079
 - multiple inheritance, 1079
 - object layout, 497–499
 - overview, 496–497, 1078–1079
 - private** bases and members, 501–502
 - protected** bases and members, 502
 - public** bases and members, 502
 - specifying, 499
 - virtual functions, 1079–1080
 - Design, 35, 174, 177, 1173
 - Design for testing, 978–979
 - Destructors, 586–588, 1077, 1173
 - See also* Constructors.
 - containers, 1108–1109
 - debugging, 622–624
 - default, 1081
 - essential operations, 620–624
 - exceptions, 675–677
 - and free store, 589–590
 - freeing resources, 318, 675–677
 - generated, 588–589
 - RAII, 675–677
 - virtual, 589–590
 - where needed, 621
 - Device drivers, 340
 - Dictionary examples, 121–123, 756
 - difference()**, 1103
 - difference_type**, 1108
 - digit**, character class, 842, 1134
 - Digit, word origin, 1041
 - Dijkstra, Edsger, 792–793, 952
 - Dimensions, matrices, 862–865
 - Direct expression of ideas, ideals, 777–778
 - Dispatch, 496
 - Display model, 409–410
 - Divide /, 66, 1051
 - Divide and assign /=, 67, 1053
 - Divide and conquer, 93
 - Divide-by-zero error, 199–200
 - divides()**, 1123
 - Domain knowledge, 896
 - Dot product. *See* **inner_product()**.
 - double** floating-point type, 63, 66–67, 78, 1062
 - Doubly-linked lists, 598, 698. *See also* **list**.
 - draw()** example
 - fill color, 492
 - line visibility, 492
 - Shape**, 491–494
 - draw_lines()** example
 - See also* **draw()** example.
 - Closed_polyline**, 452
 - Marked_polyline**, 469
 - Open_polyline**, 451
 - Polygon**, 454–455
 - Rectangle**, 459
 - Shape**, 491–494
 - Dynamic dispatch, 496. *See also* Virtual functions.
 - Dynamic memory. *See* Free store.
 - dynamic_cast**, type conversion, 1058
 - exceptions, 1099
 - predictability, 894
- ## E
- Efficiency
 - ideals, 92–94, 775–776
 - vector** range checking, 670
 - Einstein, Albert, 780
 - Elements
 - See also* **vector**.
 - numbering, 627
 - pointers to, 628–631
 - variable number of, 628
 - Ellipse** example, 466–468
 - vs.* **Circle**, 467
 - else**, in **if**-statements, 102–103
 - Email example, 820–830
 - Embedded systems
 - coding standards, 935–937, 943
 - concurrency, 894
 - containers, 912–916
 - correctness, 891–892
 - delete** operator, 894
 - domain knowledge, 896

Embedded systems (*continued*)

- dynamic_cast**, 894
- error handling, 895–897
- examples of, 888–890
- exceptions, 894
- fault tolerance, 892
- fragmentation, 898, 899
- free-store, 898–902
- hard real time, 893
- ideals, 894–895
- maintenance, 891
- memory management, 902–904
- new** operator, 894
- predictability, 893, 894
- real-time constraints, 893
- real-time response, 890
- reliability, 890
- resource leaks, 893
- resource limitations, 890
- soft real time, 893
- special concerns, 890–891

Ellipsis ...

- arguments (unchecked), 1068–1069
- catch all exceptions, 150

Empty

- empty()**, is container empty?, 1111
- lists, 702
- sequences, 702
- statements, 100

Encapsulation, 496

- Enciphering (Encryption), example, 930–935

end()

- iterator, 1109
- string**, 815, 1132
- vector**, 695

- End of line **\$** (in regular expressions), 837, 1134

End of file

- eof()**, 349, 1127
- file streams, 360
- I/O error, 349
- stringstream**, 390–391

- Ending programs. *See* Terminating programs.

- endl** manipulator, 1130

- ends** manipulator, 1130

- English grammar *vs.* programming grammar, 191–192

- enum**, 314–317, 1008. *See also* Enumerations.

- Enumerations, 314–317, 1070

- enum**, 314–317, 1008

- enumerators, 314–317, 1070–1071

- EOF** macro, 1019–1020

- eof()** stream state, 349, 1127

- equal()**, 729, 1113

- Equal **==**, 67, 1052

- Equality operators, expressions, 1052

- equal_range()**, 728, 763–764

- equal_to()**, 1122

erase()

- list**, 713–715, 1110

- list operations, 600–602

- string**, 815, 1132

- vector**, 715–718

- errno**, error indicator, 880, 1138

- error()** example, 140–141

- passing multiple strings, 150

- “uncaught exception” error, 151

- Error diagnostics, templates, 661

Error handling

- See also* Errors; Exceptions.

- %** for floating-point numbers, 228–231

- catching exceptions, 238–240

- files fail to open, 385

- GUIs, 563

- hardware replication, 896

- I/O errors. *See* I/O errors.

- I/O streams, 1127

- mathematical errors, 880

- modular systems, 896–897

- monitoring subsystems, 897

- negative numbers, 227–228

- positioning in files, 389

- predictable errors, 895

- recovering from errors, 238–240

- regular expressions, 842–844

- resource leaks, 896

- self-checking, 896

- STL (Standard Template Library), 1098–1100
- testing for errors, 224–227
- transient errors, 895–896
- vector** resource exceptions, 677
- Error messages
 - See also* **error()**; Reporting errors; **runtime_error**.
 - exceptions, printing, 148–149
 - templates, 661
 - writing your own, 140
- Errors, 1173
 - See also* Debugging; Testing.
 - classifying, 132
 - compile-time, 48–50, 132, 134–135
 - detection ideal, 133
 - error()**, 140–141
 - estimating results, 155–156
 - incomplete programs, 134
 - input format, 64–65
 - link-time, 132, 137–138
 - logic, 132, 152–154
 - poor specifications, 134
 - recovering from, 238–240. *See also* Exceptions.
 - sources of, 134
 - syntax, 135–136
 - translation units, 137–138
 - type mismatch, 136–137
 - undeclared identifier, 256
 - unexpected arguments, 134
 - unexpected input, 134
 - unexpected state, 134
- Errors, run-time, 132, 138–140
 - See also* Exceptions.
 - callee responsibility, 141–143
 - caller responsibility, 140–141
 - hardware violations, 139
 - reasons for, 142–143
 - reporting, 143–144
- Estimating development resources, 175
- Estimating results, 155–156
- Examples
 - age distribution, 527–528
 - calculator. *See* Calculator example.
 - Date**. *See* **Date** example.
 - deciphering, 930–935
 - deleting repeated words, 71–73
 - dictionary, 121–123, 756
 - Dow Jones tracking, 750–753
 - email analysis, 820–830
 - embedded systems, 888–890
 - enciphering (encryption), 930–935
 - exponential function, 517–518
 - finding largest element, 687–690, 696–697
 - fruits, 747–750
 - Gaussian elimination, 874–876
 - graphics, 410–414, 432
 - graphing data, 527–528
 - graphing functions, 517–518
 - GUI (graphical user interface), 552–556, 560–561, 563–564
 - Hello, World!, 45–46
 - intrusive containers, 1025–1031
 - Lines_window**, 552–556, 560–561, 563–564
 - Link**, 598–607
 - list (doubly linked), 598–607
 - map** container, 747–753
 - Matrix**, 872–877
 - palindromes, 637–641
 - Pool** allocator, 902–903
 - Punct_stream**, 397–401
 - reading a single value, 353–357
 - reading a structured file, 361–370
 - regular expressions, 844–849
 - school table, 844–849
 - searching, 828–836
 - sequences, 696–698
 - Stack** allocator, 903–904
 - TEA (Tiny Encryption Algorithm), 930–935
 - text editor, 704–711
 - vector**. *See* **vector** example.
 - Widget** manipulation, 552–556, 1167–1170
 - windows, 552–556
 - word frequency, 745–477
 - writing a program. *See* Calculator example.

Examples (*continued*)

writing files, 346–348

ZIP code detection, 828–836

<exception>, 1097

Exceptions, 144–148, 1086

See also Error handling; Errors.

bounds error, 147

C++ and C, 992

catch, 145, 238–240, 1087**cerr**, 149**cout**, 149

destructors, 1088

embedded systems, 894

error messages, printing, 148–149

exception, 150, 1099–1100

failure to catch, 151

GUIs, 563

input, 148–151

narrow_cast example, 151

off-by-one error, 147

out_of_range, 147

overview, 144–145

RAII (Resource Acquisition Is Initialization), 1087

range errors, 146–148

re-throwing, 677, 1087

runtime_error, 140, 149, 151

stack unwinding, 1088

standard library exceptions, 1099–1100

terminating a program, 140

throw, 145, 1086

truncation, 151

type conversion, 151

uncaught exception, 151

user-defined types, 1087

vector range checking, 668–669**vector** resources. *See* **vector**.

Executable code, 48, 1173

Executing a program, 11, 1154

exit(), terminating a program, 1149**explicit** constructor, 621–622, 1003

Expression, 94–95, 1049–1054

coding standards, 940–941

constant expressions, 1056–1057

conversions, 1054–1056

debugging, 159

grouping **()**, 95, 831, 837, 840

lvalue, 94–95, 1054

magic constants, 96, 141, 231–233, 697

memory management, 1057–1058

mixing types, 98–99

non-obvious literals, 96

operator precedence, 95

operators, 97, 1049–1059

order of operations, 179

precedence, 1054

preserving values, 1054–1055

promotions, 98–99, 1054–1055

rvalue, 94–95, 1054

scope resolution, 1049

type conversion, 98–99, 1058–1059

usual arithmetic conversions, 1056

Expression statement, 99

Empty statement, 1001

extern, 257, 999

Extracting text from files, 820–825, 828–830

F**f/F** suffix, 1042**fail()** stream state, 349, 1127

Falling through end of functions, 272

false, 1003–1004

Fault tolerance, 892

fclose(), 1019–1020, 1140

Feature creep, 186, 199, 1173

Feedback, programming, 36

Fields, formatting, 383–384

FILE, 1019–1020

File I/O, 343–344

binary I/O, 387

converting representations, 368–370

close(), 346

closing files, 346, 1140–1141

modes, 1140–1141

- open()**, 346
 - opening files. *See* Opening files.
 - positioning in files, 389
 - reading. *See* Reading files.
 - writing. *See* Writing files.
- Files, 1173
 - See also* File I/O.
 - C++ and C, 1019–1020
 - opening and closing, C-style I/O, 1140–1141
- fill()**, 1116
- Fill color example, 456–459, 492
- fill_n()**, 1116
- find()**, 728–731
 - associative container operations, 1111
 - finding links, 600–602
 - generic use, 731–732
 - nonmodifying sequence algorithms, 1113
 - string operations, 815, 1132
- find_end()**, 1113
- find_first_of()**, 1113
- find_if()**, 728, 732–734
- Finding
 - See also* Matching; Searching.
 - associative container operations, 1111
 - elements, 728
 - links, 600–602
 - patterns, 828–830, 833–836
 - strings, 815, 1132
- fixed** format, 383
- fixed** manipulator, 381, 1130
- float** type, 1062
- <float.h>**, 858, 1136
- Floating-point, 63, 855, 1173
 - %** remainder (modulo), 199
 - assigning integers to, 856–857
 - assigning to integers, 857
 - conversions, 1055
 - fixed** format, 383
 - general** format, 383
 - input, 180, 199–200
 - integral conversions, 1055
 - literals, 180, 1042–1043
 - mantissa, 857
 - output, formatting, 380–381
 - precision, 382–383
 - and real numbers, 855
 - rounding, 382–383
 - scientific** format, 383
 - truncation, 857
 - vector** example, 119–121
- floor()**, 879, 1137
- FLT_DIG** limit macro, 1137
- FLTK (Fast Light Toolkit), 414, 1158
 - code portability, 414
 - color, 447, 459–461
 - current style, obtaining, 492
 - downloading, 1158
 - fill, 459
 - in graphics code, 432
 - installing, 1159
 - lines, drawing, 449, 452–453
 - outlines, 459
 - rectangles, drawing, 459
 - testing, 1160
 - in Visual Studio, 1159–1160
 - waiting for user action, 547–548, 556–557
- FLT_MAX** limit macro, 1137
- FLT_MAX_10_EXP** limit macro, 1137
- FLT_MIN** limit macro, 1137
- flush** manipulator, 1130
- Flushing a buffer, 239–240
- Fonts for Graphics example, 463–464
- fopen()**, 1019–1020, 1140
- for**-statement, 110–112
- Ford, Henry, 772
- for_each()**, 1113
- Formal arguments. *See* Parameters.
- Formatting
 - See also* I/O streams, 1128–1129.
 - See also* C-style I/O, 1016–1019.
 - See also* Manipulators, 1129–1130.
 - case, 393–394
 - fields, 383–384
 - precision, 382–383
 - whitespace, 393

- Fortran language, 786–788
 - array indexing, 863
 - complex**, 882
 - subscripting, 863
- Forward declarations, 259
- Forward iterators, 722–723, 1103
- fprintf()**, 1017, 1141
- Fragmentation, embedded systems, 898, 899
- free()**, deallocate, 1009–1010, 1147
- Free store (heap storage)
 - allocation, 578–579
 - C++ and C, 1009–1011
 - deallocation, 584–586
 - delete**, 584–586, 586–590
 - and destructors. *See* destructors.
 - embedded systems, 898–902
 - garbage collection, 585
 - leaks, 584–586, 586–590
 - new**, 578–579
 - object lifetime, 1048
- Freeing memory. *See* Deallocating memory.
- friend**, 1003, 1073–1074
- from_string()** example, 817–818
- front()**, first element, 1109
- front_inserter()**, 1122
- fstream()**, 1126
- <fstream>**, 1096
- fstream** type, 344–346
- Fully qualified names, 291–293
- Function** example, 439, 515–518
- Function, 47, 112–114
 - See also* Member functions.
 - accessing class members, 1073–1074
 - arguments. *See* Function arguments.
 - in base classes, 496
 - body, 47, 113
 - C++ and C, 994–998
 - callback, GUIs, 544–546
 - calling, 1066
 - cleaning up, 233–234
 - coding standards, 940–941
 - common style, 482–483
 - debugging, 158
 - declarations, 115–116, 1066
 - definition, 112, 269, 1173
 - in derived classes, 493, 496
 - falling through, 272
 - formal arguments. *See* Function parameters.
 - friend** declaration, 1073–1074
 - generic code, 483
 - global variables, modifying, 267
 - graphing. *See* **Function** example.
 - inline, 312, 992
 - linkage specifications, 1069
 - naming. *See* Namespaces.
 - nesting, 267
 - organizing. *See* Namespaces.
 - overload resolution, 1067–1068
 - overloading, 316–318, 516, 992
 - parameter, 113. *See also* Function parameters.
 - pointer to, 1000–1002
 - post-conditions, 163–164
 - pre-conditions, 161–163
 - pure virtual, 1175
 - requirements, 151. *See also* Pre-conditions.
 - return type, 47, 270–271
 - return**, 112–113, 271–272, 1066
 - standard mathematical, 518, 1137–1138
 - types as parameters. *See* Templates.
 - uses for, 114–115
 - virtual**, 1000–1002. *See also* Virtual functions.
- Function activation record, 284
- Function argument
 - See also* Function parameter; Parameters.
 - checking, 281–282
 - conversion, 281–282
 - declaring, 270–271
 - formal. *See* Parameters.
 - naming, 270–271
 - omitting, 270
 - passing. *See* Function call.
- Function call, 282
 - ()** operator, 735–736

- call stack, 287
- expression()** call example, 284–287
- function activation record, 284
- history of, 785
- memory for, 577
- pass by **const** reference, 273–276, 279–281
- pass by non-**const** reference, 279–281
- pass by reference, 276–281
- pass by value, 273, 279–281
- recursive, 286
- stack growth, 284–287. *See also*
 - Function activation record.
 - temporary objects, 280
- Function-like macros, 1022–1023
- Function member
 - definition, 301–302
 - same name as class. *See* Constructors.
- Function object, 734–736
 - ()** function call operator, 735–736
 - abstract view, 736–737
 - adaptors, 1123
 - arithmetic operations, 1123
 - parameterization, 736–737
 - predicates, 737–738, 1122–1123
- Function parameter (formal argument)
 - ...** ellipsis, unchecked arguments, 1068
 - pass by **const** reference, 273–276, 279–281
 - pass by non-**const** reference, 279–281
 - pass by reference, 276–281
 - pass by value, 273, 279–281
 - temporary objects, 280
 - unused, 270
- Function template
 - algorithms, 659–661
 - argument deduction, 664–665
 - parameterized functions, 659–661
- <functional>**, 1095, 1122–1123
- Functional cast, 1058
- Functional programming, 788
- Fused multiply-add, 868

G

- Gadgets. *See* Embedded systems.
- Garbage collection, 585, 900–901
- Gaussian elimination, 874–875
- gcount()**, 1128
- general** format, 383
- general** manipulator, 381
- generate()**, 1116
- generate_n()**, 1116
- Generic code, 483
- Generic programming, 659–661, 782, 1173
- Geometric shapes, 423
- get()**, 1128
- getc()**, 1018, 1145
- getchar()**, 1019, 1145
- getline()**, 391–392, 815, 819, 1128
- gets()**, 1018
 - C++ alternative **>>**, 1019
 - dangerous, 1018
 - scanf()**, 1144–1145
- get_token()** example, 194
- GIF images, 473–475
- Global scope, 264, 267, 1046
- Global variables
 - functions modifying, 267
 - memory for, 577
 - order of initialization, 288–290
- Going out of scope, 266–267, 287
- good()** stream state, 349, 1127
- GP. *See* Generic programming.
- Grammar example
 - alternation, patterns, 192–193
 - English grammar, 191–192
 - Expression** example 186–191, 195–198, 200–201
 - parsing, 188–191
 - repetition, patterns, 192–193
 - rules *vs.* tokens, 192–193
 - sequencing rules, 192–193
 - terminals. *See* Tokens.
 - writing, 187, 192–193
- Graph.h**, 417–418
- Graphical user interfaces. *See* GUIs.
- Graphics, 408

Graphics (*continued*)

See also Color; Graphics example; Shape.

display model, 409–410

displaying, 472–475

drawing on screen, 419–420

encoding, 473

filling shapes, 427

formats, 473

geometric shapes, 423

GIF, 473–475

graphics libraries, 474–475

graphs, 422–423

images from files, 429–430

importance of, 408–409

JPEG, 473–475

line style, 427

loading from files, 429–430

screen coordinates, 415–416

selecting a sub-picture from, 473

user interface. *See* GUIs (graphical user interfaces).

Graphics example

Graph.h, 417–418

GUI system, giving control to, 419

header files, 417–418

main(), 417–418

Point.h, 440

points, 422–423

Simple_window.h, 440

wait_for_button(), 419

Window.h, 440

Graphics example, design principles

access control. *See* Access control.

attach() *vs.* **add()**, 483–484

class diagram, 497

class size, 481–482

common style, 482–483

data modification access, 484–485

generic code, 483

inheritance, interface, 504–505

inheritances, implementation, 504–505

mutability, 484–485

naming, 483–484

object-oriented programming, benefits of, 504–505

operations, 482–483

private data members, 484–485

protected data, 484–485

public data, 484–485

types, 480–482

width/height, specifying, 482

Graphics example, GUI classes, 438–440

See also Graphics example (interfaces).

Button, 439

In_box, 439

Menu, 439

Out_box, 439

Simple_window, 418–420, 439

Widget, 548–550, 1163–1164

Window, 439, 1164–1166

Graphics example, interfaces, 438–439

See also Graphics example (GUI classes).

Axis, 420–422, 439, 518–521

Circle, 464–467, 489

Closed_polyline, 451–453

Color, 445–447

Ellipse, 466–468

Function, 439, 514–518

Image, 439, 472–475

Line, 441–444

Lines, 443–445, 489

Line_style, 448–450

Mark, 470–472

Marked_polyline, 468–469

Marks, 469–470, 489

Open_polyline, 450–451, 489

Point, 422–423, 441

Polygon, 423–424, 453–455, 489

Rectangle, 424–427, 455–459, 489

Shape, 440–441, 445, 485–499, 504–505

Text, 427–429, 462–464

Graphing data example, 527–534

Graphing functions example, 510–514, 521–526

Graph_lib namespace, 417–418

Graph example
 See also Grids.
 Axis, 420–422
 coordinates, 422–423
 drawing, 422–423
 points, labeling, 468–469
greater(), 1122
 Greater than **>**, 67, 1052
 Greater than or equal **>=**, 1052
greater_equal(), 1122
 Green marginal alerts, 3
 Grids, drawing, 444–445, 448–450
 Grouping regular expressions, 831, 837, 840
 Guarantees, 676–678
 GUI system, giving control to, 419
 Guidelines. *See* Ideals.
 GUIs (graphical user interfaces), 540–541
 See also Graphics example (GUI classes).
 callback functions, 544–546
 callback implementation, 1162–1163
 cb_next() example, 544–546
 common problems, 562–564
 control inversion, 556–557
 controls. *See* Widgets.
 coordinates, computer screens, 415–416
 debugging, 562–564
 error handling, 563
 examples, 552–556, 560–561, 563–564
 exceptions, 563
 FLTK (Fast Light Toolkit), 414
 layers of code, 544–545
 next() example, 546
 pixels, 415–416
 portability, 414
 standard library, 414–415
 system tests, 969–973
 toolkit, 414
 vector of references, simulating, 1166–1167
 vector_ref example, 1166–1167

wait loops, 547–548
wait_for_button() example, 547–548
 waiting for user action, 547–548, 556–557
Widget example, 548–556, 1163–1164, 1167–1170
Window example, 552–556, 1164–1166

H

.h file suffix, 46
 Half open sequences, 694–695
 Hard real-time, 893, 942
 Hardware replication, error handling, 896
 Hardware violations, 139
 Hash function, 753–754
 Hash tables, 753
 Hash values, 753
 Hashed container. *See* **unordered_map**.
 Hashing, 753
 Header files, 46, 1173
 C standard library, 1097
 declarations, managing, 261–262
 definitions, managing, 261–262
 graphics example, 417–418
 including in source files, 262–264, 1090–1091
 multiple inclusion, 1025
 standard library, 1095–1097
 Headers. *See* Header files.
 Heap algorithm, 1119–1120
 Heap memory, 897–898. *See also* Free store.
 Hejlsberg, Anders, 796
 “Hello, World!” program, 45–47
 Helper functions
 != inequality, 328
 == equality, 328
 class interfaces, 326–328
 Date example, 305–306, 327
 namespaces, 328
 validity checking date values, 306
hex manipulator, 378–379, 1130
 Hexadecimal digits, 393

Hexadecimal number system, 377–379,
1041–1042
Hiding information, 1173
Hopper, Grace Murray, 789–790
Hyperbolic cosine, **cosh()**, 879
Hyperbolic sine, **sinh()**, 879
Hyperbolic tangent, **tanh()**, 879

I

I/O errors

bad() stream state, 349
clear(), 349–352
end of file, 349
eof() stream state, 349
error handling, 1127
fail() stream state, 349
good() stream state, 349
ios_base, 351
stream states, 349
recovering from, 349–352
unexpected errors, 349
unset(), 349–352

I/O streams, 1124–1125

<< output operator, 819
>> input operator, 819
cerr, standard error output stream,
149, 1125, 1144
cin standard input, 341
class hierarchy, 819, 1126–1127
cout standard output, 341
error handling, 1127
formatting, 1128–1129
fstream, 384–386, 389, 1126
get(), 819
getline(), 819
header files, 1096
ifstream, 384–386, 1126
input operations, 1128
input streams, 341–343
iostream library, 341–343, 1124–
1125
istream, 341–343, 1125–1126
istreamstream, 1126
ofstream, 384–386, 1126
ostream, 341–343, 1124–116

ostreamstream, 384–386, 1126
output operations, 1128–1129
output streams, 341–343
standard manipulators, 378, 1129–
1131
standard streams, 1125
states, 1127
stream behavior, changing, 378
stream buffers, **streambufs**, 1125
stream modes, 1126
string, 819
stringstream, 390–391, 1126
throwing exceptions, 1127
unformatted input, 1128

IBM, 786–788

Ichbiah, Jean, 797

IDE (interactive development environ-
ment), 52

Ideals

abstraction level, 778–779
bottom-up approach, 776–777
class interfaces, 318
code structure, 776
coding standards, 936–937
consistency, 780
correct approaches, 776–777
correctness, 775
definition, 1173
direct expression of ideas, 777–778
efficiency, 775–776
embedded systems, 894–895
importance of, 8
KISS, 780
maintainability, 775
minimalism, 780
modularity, 779–780
on-time delivery, 776
overview, 774–775
performance, 775–776
software, 34–37
top-down approach, 776–777

Identifiers, 1045. *See also* Names.

reserved, 75–76. *See also* Keywords.

if-statements, 101–103

#ifdef, 1024–1025

#ifndef, 1025

- ifstream** type, 344–346
- imag()**, imaginary part, 881, 1139
- Image** example, 439, 472–475
- Images. *See* Graphics.
- Imaginary part, 881
- Immutable values, class interfaces, 325–326
- Implementation, 1173
 - class, 302–304
 - inheritance, 504–505
 - programs, 35
- Implementation-defined feature, 1039
- Implicit conversions, 621–622
- In-class member definition, 1074–1075
- in** mode, 385, 1126
- In_box** example, 439, 550–551
- #include**, 46, 262–264, 1090
- Include guard, 1025
- includes()**, 1119
- Including headers, 1090–1091. *See also* **#include**.
- Incrementing **++**, 66, 695
 - iterators, 694–695, 721, 1101–1104
 - pointers, 630
 - variables, 73–74, 97–98
- Indenting nested code, 269
- Inequality **!=** (not equal), 67, 1052, 1064
 - complex**, 881, 1138
 - containers, 1111
 - helper function, 328
 - iterators, 695, 1102
 - string**, 67, 815, 1132
- Infinite loop, 1173
- Infinite recursion, 196, 1173
- Information hiding, 1173
- Information processing, 31–32
- Inheritance
 - class diagram, 497
 - definition, 496
 - derived classes, 1078–1079
 - embedded systems, 912–916
 - history of, 799
 - implementation, 504–505
 - interface, 504–505
 - multiple, 1079
 - pointers *vs.* references, 598
 - templates, 661–662
- Initialization, 69–73, 1173
 - arrays, 582–583, 633–634
 - constants, 260, 324–325, 1062
 - constructors, 306–309
 - Date** example, 305–309
 - default, 261, 322–323, 1048
 - invariants, 309, 676–677
 - menus, 558
 - pointer targets, 582–583
 - pointers, 582–583, 635
 - Token** example, 183
- inline**, 1003
- Inline
 - functions, 992
 - member functions, 312
- inner_product()**, 729
 - See also* Dot product.
 - description, 742–743
 - generalizing, 743–744
 - matrices, 868
 - multiplying sequences, 1139
 - standard library, 729, 739
- inplace_merge()**, 1118
- Input, 60–62
 - See also* I/O streams; Input **>>**.
 - binary I/O, 386–389
 - C++ and C, 1017–1019
 - calculator example, 177, 180, 183–184, 199–200, 204–206
 - case sensitivity, 64
 - cin**, standard input stream, 61
 - dividing functions logically, 353–356
 - files. *See* File I/O.
 - format errors, 64–65
 - individual characters, 392–394
 - integers, 379–380
 - istreamstream**, 390
 - line-oriented input, 391–392
 - newline character **\n**, 61–62, 64
 - potential problems, 352–357
 - prompting for, 61, 177
 - separating dialog from function, 356–357

Input (*continued*)

- a series of values, 350–352
- a single value, 352–357
- source of exceptions, 148–151
- stringstream**, 390–391
- tab character **\t**, 64
- terminating, 61–62
- type sensitivity, 64–65
- whitespace, 64

Input **>>**, 61

- case sensitivity, 64
- complex**, 881, 1139
- formatted input, 1128
- multiple values per statement, 65
- strings, 815, 1132
- text input, 815, 819
- user-defined, 359
- whitespace, ignoring, 64

Input devices, 340–341

Input iterators, 722–723, 1103

Input loops, 359–361

Input/output, 341–343

- See also* Input; Output.
- buffering, 342, 402
- C++ and C. *See* **stdio**.
- computation overview, 91
- device drivers, 340
- errors. *See* I/O errors.
- files. *See* File I/O.
- formatting. *See* Manipulators; **printf()**.
- irregularity, 376
- istream**, 341–348
- natural language differences, 402
- ostream**, 341–348
- regularity, 376
- streams. *See* I/O streams
- strings, 819
- text in GUIs, 550–551
- whitespace, 393, 394–401

Input prompt **>**, 221Input streams, 341–343. *See also* I/O streams.

Inputs, testing, 961

insert()

- list**, 600–602, 713–715

map container, 750, 751**string**, 815, 1110, 1132**vector**, 715–718**insert()**, 1122

Inserters, 1121–1122

Inserting

- list** elements, 713–715
- into **strings**, 815, 1110, 1132
- vector** elements, 715–718

Installing

- FLTK (Fast Light Toolkit), 1159
- Visual Studio, 1152

Instantiation, templates, 658–659, 1084–1085

int, integer type, 66–67, 78, 1062

- bits in memory, 78, 917

Integers, 77–78, 854–855, 1174

- assigning floating-point numbers to, 857

- assigning to floating-point numbers, 856–857

decimal, 377–379

input, formatting, 379–380

largest, finding, 879

literals, 1041

number bases, 377–379

octal, 377–379

output, formatting, 377–379

reading, 379–380

smallest, finding, 879

Integral conversions, 1055

Integral promotion, 1054–1055

Interactive development environment (IDE), 52

Interface classes. *See* Graphics example (interfaces).

Interfaces, 1174

- classes. *See* Class interfaces.

inheritance, 504–505

user. *See* User interfaces.**internal** manipulator, 1130**INT_MAX** limit macro, 1136**INT_MIN** limit macro, 1136

Intrusive containers, example, 1025–1031

Invariants, 309, 1174
See also Post-conditions; Pre-conditions.
 assertions, 161
 debugging, 160–161
 default constructors, 620
 documenting, 780
Date example, 309
 invention of, 793
Polygon example, 455
 Invisible. *See* Transparency.
<iomanip>, 1096, 1129
<ios>, 1096, 1129
<iosfwd>, 1096
istream
 buffers, 402
 C++ and C, 1016
 exceptions, 1099
 library, 341–343
<istream>, 1096, 1129
 Irregularity, 376
is_open(), 1126
isalnum() classify character, 393, 1131
isalpha() classify character, 247, 393, 1131
iscntrl() classify character, 393, 1131
isdigit() classify character, 393, 1131
isgraph() classify character, 393, 1131
islower() classify character, 393, 1131
isprint() classify character, 393, 1131
ispunct() classify character, 393, 1131
isspace() classify character, 393, 1131
istream, 341–343, 1125–1126
 >>, text input, 815, 1128
 >>, user-defined, 359
 binary I/O, 386–389
 connecting to input device, 1126
 file I/O, **fstream**, 343–348, 1126
 get(), get a single character, 393
 getline(), 391–392, 1128
 stringstreams, 390–391
 unformatted input, 391–393, 1128
 using together with **stdio**, 1016–1017
<istream>, 1096, 1124, 1128–1129

istream_iterator type, 758–761
istreamstream, 390
isupper() classify character, 393, 1131
isxdigit() classify character, 393, 1131
 Iteration
 See also Iterators.
 control variables, 109
 definition, 1174
 example, 708–711
 for-statements, 110–112
 linked lists, 701–703, 708–711
 loop variables, 109
 strings, 815
 through values. *See* **vector**.
 while-statements, 108–109
iterator, 1108
<iterator>, 1095, 1121
 Iterators, 694–696, 1100–1101, 1174
 See also STL iterators.
 bidirectional iterator, 722–723
 category, 722–723, 1103–1105
 containers, 1109, 1104–1105
 empty list, 702
 example, 708–711
 forward iterator, 722–723
 header files, 1095–1096
 input iterator, 722–723
 operations, 695, 1102–1103
 output iterator, 722–723
 vs. pointers, 1101
 random-access iterator, 723
 sequence of elements, 1101–1102
iter_swap(), 1116

J

Japanese age distribution example, 527–528
 JPEG images, 473–475

K

Kernighan, Brian, 802–803, 988–989
key_comp(), 1112
key_compare, 1108

key_type, 1108

Key,value pairs, containers for, 744

Keywords, 1003–1004, 1045–1046

KISS, 780

Knuth, Don, 774–775

K&R, 802, 988

L

/L suffix, 1041

\l, “lowercase character,” **regex**, 837, 1135

\L, “not lowercase,” **regex**, 939, 1135

Label

access control, 302, 502

case, 104–107

graph example, 518–521

of statement, 1059

Largest integer, finding, 879

Laws of optimization, 893

Layers of code, GUIs, 544–545

Layout rules, 939–940, 1000

Leaks, memory, 584–586, 586–590, 899

Leap year, 305

left manipulator, 1130

Legal programs, 1039

length(), 815, 1132

Length of strings, finding, 815, 1012, 1132

less(), 1122

Less than **<**, 1052

Less than or equal **<=**, 67, 1052

less_equal(), 1122

Letters, identifying, 247, 393

lexical_cast, 819

Lexicographical comparison

< comparison, 815, 1132

<= comparison, 1132

> comparison, 1132

>= comparison, 1132

C-style strings, 1012

lexicographical_compare(), 1121

Libraries, 51, 1174

See also Standard library.

role in debugging, 158

uses for, 175

Lifetime, objects, 1048–1049, 1174

Limit macros, 1136–1137

Limits, 858

<limits>, 858, 1096, 1135

<limits.h>, 858, 1136

Line comment **//**, 45

Line example, 441–443

vs. **Lines**, 444

Line-oriented input, 391–392

Linear equations example, 872–877

back_substitution(), 874–875

classic_elimination(), 874–875

Gaussian elimination, 874–875

pivoting, 875–876

testing, 876–877

Lines example, 443–445, 489

vs. **Line**, 444

Lines (graphic), drawing

See also **draw_lines()**; Graphics.

on graphs, 518–521

line styles, 448–450

multiple lines, 443–445

single lines, 441–443

styles, 427, 449

visibility, 492

Lines (of text), identifying, 707–708

Line_style example, 448–450

Lines_window example, 552–556, 560–561, 563–564

Link example, 598–607

Link-time errors. *See* Errors, link-time.

Linkage convention, C, 999

Linkage specifications, 1069

Linked lists, 698. *See also* Lists.

Linkers, 51, 1174

Linking programs, 51

Links, 598–602, 606–607, 699

Lint, consistency checking program, 801

Lisp language, 790–791

list, 700, 1107–1111

add(), 600–602

advance(), 600–602

back(), 708

erase(), 600–602, 713–715

find(), 600–602

- `insert()`, 600–602, 713–715
 - operations, 600–602
 - properties, 712–713
 - referencing last element, 708
 - sequence containers, 1105
 - subscripting, 700
 - `<list>`, 1095
 - Lists
 - containers, 1110
 - doubly linked, 598, 698
 - empty, 702
 - erasing elements, 713–715
 - examples, 598–600, 704–711
 - finding links, 600–602
 - getting the *n*th element, 600–602
 - inserting elements, 600–602, 713–715
 - iteration, 701–703, 708–711
 - link manipulation, 600–602
 - links, examples, 598–600, 606–607, 699
 - operations, 699–700
 - removing elements, 600–602
 - singly linked, 598, 698
 - `this` pointer, 603–605
 - Literals, 62, 1041, 1174
 - character, 159, 1043–1044
 - decimal integer, 1041
 - in expressions, 96
 - `f/F` suffix, 1042
 - floating-point, 1042–1043
 - hexadecimal integer, 1041
 - integer, 1041
 - `I/L` suffix, 1041
 - magic constants, 96, 141, 231–233, 697
 - non-obvious, 96
 - null pointer, `0`, 1044–1045
 - number systems, 1041–1042
 - octal integer, 1041
 - special characters, 1043–1044
 - string, 159, 1044
 - termination, debugging, 159
 - for types, 63
 - `u/U` suffix, 1041
 - unsigned, 1041
 - Local (automatic) objects, lifetime, 1048
 - Local classes, nesting, 268
 - Local functions, nesting, 268
 - Local scope, 265–266, 1046
 - Local variables, array pointers, 636–637
 - Locale, 402
 - `<locale>`, 1097
 - `log()`, 879, 1137
 - `log10()`, 879, 1137
 - Logic errors. *See* Errors, logic.
 - Logical and `&&`, 1052, 1057
 - Logical operations, 1057
 - Logical or `||`, 1053, 1057
 - `logical_and()`, 1122
 - `logical_not()`, 1122
 - `logical_or()`, 1122
 - Logs, graphing, 517–518
 - `long` integer, 917, 1062
 - `LONG_MAX` limit macro, 1137
 - `LONG_MIN` limit macro, 1137
 - Look-ahead problem, 202–207
 - Loop, 109, 111, 1174
 - examples, parser, 198
 - infinite, 196, 1173
 - testing, 965–966
 - variable, 109, 111
 - Lovelace, Augusta Ada, 797
 - `lower`, 842, 1134
 - `lower_bound()`, 764, 1112, 1117
 - Lower case. *See* Case.
 - Lucent Bell Labs, 803
 - Lvalue, 94–95, 1054
- ## M
- Machine code. *See* Executable code.
 - Macro substitution, 1090–1091
 - Macros, 1021–1022
 - conditional compilation, 1024–1025
 - `#define`, 1022–1024, 1090–1091
 - function-like, 1022–1023
 - `#ifdef`, 1024–1025
 - `#ifndef`, 1025
 - `#include`, 1024, 1090
 - include guard, 1025

Macros (*continued*)

- naming conventions, 1021
- syntax, 1023–1024
- uses for, 1021

Maddoc, John, 830

Magic constants, 96, 141, 231–233, 697

Magical approach to programming, 10

main(), 46–47

- arguments to, 1040
- global objects, 1040
- return values, 47, 1039–1040
- starting a program, 1039–1040

Maintainability, software, 34, 775

Maintenance, 891

make_heap(), 1119

make_pair(), 751, 1124

make_vec(), 677

malloc(), 1009, 1147

Manipulators, 378, 1129–1131

- complete list of, 1129–1130

dec, 1130

endl, 1130

fixed, 1130

hex, 1130

noskipws, 1129

oct, 1130

resetiosflags(), 1130

scientific, 1130

setiosflags(), 1130

setprecision(), 1130

skipws, 1129

Mantissa, 857

map, associative array, 744–750

See also **set**; **unordered_map**.

[], subscripting, 745, 1111

balanced trees, 748–750

binary search trees, 747

case sensitivity, **No_case** example, 762–763

counting words example, 745–747

Dow Jones example, 750–753

email example, 820–836

erase(), 749, 1110

finding elements in, 745, 749, 1111–1112

fruits example, 747–750

insert(), 750, 751, 1110

iterators, 1105

key storage, 745

make_pair(), 751

No_case example, 750, 762–763

Node example, 747–750

red-black trees, 747

vs. **set**, 756

standard library, 1107–1112

tree structure, 747–750

without values. *See* **set**.

<map>, 744, 1095

mapped_type, 1108

Marginal alerts, 3

Mark example, 470–472

Marked_polyline example, 468–469

Marks example, 469–470, 489

Mars Rover, 32–33

Matching

See also Finding; Searching.

regular expressions, **regex**, 1133–1135

text patterns. *See* Regular expressions.

Math functions, 518, 1137–1138

Mathematics. *See* Numerics.

Mathematical functions, standard

abs(), absolute value, 879

acos(), arccosine, 879

asin(), arcsine, 879

atan(), arctangent, 879

ceil(), 879

<cmath>, 879, 1096

<complex>, 881

cos(), cosine, 879

cosh(), hyperbolic cosine, 879

errno, error indicator, 880

error handling, 880

exp(), natural exponent, 879

floor(), 879

log(), natural logarithm, 879

log10(), base-10 logarithm, 879

sin(), sine, 879

sinh(), hyperbolic sine, 879

sqrt(), square root, 879

tan(), tangent, 879

tanh(), hyperbolic tangent, 879

Matrices, 863–865, 869

Matrix library example, 863–865, 869
 (), subscripting (Fortran style), 863
 [], subscripting (C style), 860, 863
 accessing array elements, 863–865
apply(), 867
 broadcast functions, 867
clear_row, 870
 columns, 864–865, 870
 dimensions, 862–865
 dot product, 868
 fused multiply-add, 868
 initializing, 870
inner_product, 868
 input/output, 870–871
 linear equations example, 874–877
 multidimensional matrices, 862–872
 rows, 864–865, 870
scale_and_add(), 868
slice(), 865–866, 869
start_row, 870
 subscripting, 863–865, 869
swap_columns(), 870
swap_rows(), 870

max(), 1120–1121

max_element(), 1121

max_size(), 1111

McCarthy, John, 791

McIlroy, Doug, 802, 998

Medicine, computer use, 30

Member, 301–303

See also Class.

allocated at same address, 1082–1083

class, nesting, 268

definition, 1071

definitions, 1074–1075

in-class definition, 1074–1075

out-of-class definition, 1074–1075

static const int members, 1075

Member access

See also Access control.

. (dot), 1072

-> (arrow), 593, 1072

:: scope resolution, 310, 1072

notation, 182

operators, 593

this pointer, 1073

by unqualified name, 1072

Member function

See also Class members; Constructors; Destructors; **Date** example.

calls, 118

nesting, 267

Token example, 182–183

Member initializer list, 183

Member selection, expressions, 1051

Member types

containers, 1108

templates, 1086

memchr(), 1147

memcmp(), 1147

memcpy(), 1147

mem_fun() adaptor, 1123

mem_fun_ref() adaptor, 1123

memmove(), 1147

Memory, 574–576

addresses, 574

allocating. *See* Allocating memory.

automatic storage, 577

bad_alloc exception, 1058

C standard library functions, 1146–1147

for code, 577

deallocating, 584–586

embedded systems, 902–904

exhausting, 1058

free store, 577–579

freeing. *See* Deallocating memory.

for function calls, 577

for global variables, 577

heap. *See* Free store.

layout, 577

object layout, 497–499

object size, getting, 576–577

pointers to, 574–576

sizeof, 576–577

stack storage, 577

static storage, 577

text storage, 577

<memory>, 1095

memset(), 1147

Menu example, 439, 551, 557–562
merge(), 728, 1118
 Messages to the user, 551
min(), 1120–1121
min_element(), 1121
 Minimalism, ideals, 780
 Minus **-**. *See* Subtraction.
minus(), 1123
 Missing copies, 624
 MIT, 791, 803
 Modifying sequence algorithms, 1114–1116
 Modular systems, error handling, 896–897
 Modularity, ideals, 779–780
 Modulo (remainder) **%**, 66. *See also* Remainder.
modulus(), 1123
 Monitoring subsystems, error handling, 897
move(), 494, 549
 Move backward **-=**, 1064
 Move forward **+=**, 1064
 Multi-paradigm programming languages, 783
 Multidimensional matrices, 862–872
multimap, 744, 824–825, 1105
<multimap>, 744
 Multiplicative operators, expressions, 1051
multiplies(), 1123
 Multiply *****, 66, 1051
 Multiply and assign ***=**, 67
multiset, 744, 1105
<multiset>, 744
 Mutability, 484–485, 1174
 class interfaces, 326–328
 and copying, 494–496
mutable, 1003
 Mutating sequence algorithms, 1114–1116

N

\n newline, character literal, 61–62, 64, 1043
 Named character classes, in regular expressions, 841–842
 Names, 74–77
 _ (underscore), 75, 76
 capital letters, 76–77
 case sensitivity, 75
 confusing, 77
 conventions, 74–75
 declarations, 255–256
 descriptive, 76
 function, 47
 length, 76
 overloaded, 138, 500, 1067–1068
 reserved, 75–76. *See also* Keywords.
namespace, 269, 1003
 Namespaces, 290, 1088
 See also Scope.
 :: scope resolution, 291
 C++ and C, 1008
 fully qualified names, 291–293
 helper functions, 328
 objects, lifetime, 1048
 scope, 264, 1046
 std, 291–292
 for the STL, 1098
 using declarations, 291–293
 using directives, 291–293, 1089
 variables, order of initialization, 288–290
 Naming conventions, 74–77
 coding standards, 939–940
 enumerators, 316
 functions, 483–484
 macros, 1021
 role in debugging, 158
 scope, 267
narrow_cast example, 151
 Narrowing conversions, 80–83
 Narrowing errors, 151
 Natural language differences, 402
 Natural logarithms, 879
 Naur, Peter, 792–793
negate(), 1123
 Negative numbers, 227–228
 Nested blocks, 268–269
 Nested classes, 268

Nested functions, 268

Nesting

blocks within functions, 268–269

classes within classes, 268

classes within functions, 268

functions within classes, 267

functions within functions, 268

indenting nested code, 269

local classes, 268

local functions, 268

member classes, 268

member functions, 267

structs, 1003

new, 578, 582

C++ and C, 992, 1003

and **delete**, 1057–1058

embedded systems, 894, 898–901,
901–902

example, 578–579

exceptions, 1099

types, constructing, 1050, 1051

<new>, 1097

New-style casts, 1006

next_permutation(), 1120

No-throw guarantee, 677

noboolalpha, 1129

No_case example, 750

Node example, 747–750

Non-algorithms, testing, 961–968

Non-errors, 137

Non-intrusive containers, 1025

Nonmodifying sequence algorithm,
1113–1114

Nonstandard separators, 394–401

norm(), 881, 1138

Norwegian Computing Center, 798–
800

noshowbase, 379, 1129

noshowpoint, 1129

noshowpos, 1129

noskipws, 1129

not, synonym for **!**, 1003, 1004

Not-conforming constructs, 1039

Not **!**, 1050

not1() adaptor, 1123

not2() adaptor, 1123

Notches, graphing data example, 518–
521, 532–534

Not equal **!=** (inequality), 67, 1052,
1064

not_eq, synonym for **!=**, 1003, 1004

not_equal_to(), 1122

nouppercase manipulator, 1130

nth_element(), 1117

Null pointer, 583–584, 634–635,
1044–1045

Number example, 187

Number systems

base-2, binary, 1042

base-8, octal, 377–380, 1041–1042

base-10, decimal, 377–380, 1041–
1042

base-16, hexadecimal, 377–380,
1041–1042

<numeric>, 1096, 1139

Numerical algorithms. *See* Algorithms,
numerical.

Numerics, 854–855

absolute values, 879

arithmetic function objects, 1123

arrays. *See* **Matrix** library example.

<cmath>, 879

columns, 859

complex, 881, 1138–1139

<complex>, 881

floating-point rounding errors,
856–857

header files, 1096

integer and floating-point, 856–857

integer overflow, 854–857

largest integer, finding, 879

limit macros, 1136–1137

limits, 858

mantissa, 857

mathematical functions, 879–880

Matrix library example, 861–872

multi-dimensional array, 859–861

numeric_limits, 1135–1136

numerical algorithms, 1139

overflow, 854–858

precision, 854–858

rand(), random number, 878

Numerics (*continued*)

- random numbers, 877–879
- real numbers, 855. *See also* Floating-point.
- results, plausibility checking, 855
- rounding errors, 855
- rows, 859
- size, 854–858
- sizeof()**, 856
- smallest integer, finding, 879
- srand()**, seed random number generator, 878
- standard mathematical functions, 879–880, 1137–1138
- truncation, 857
- valarray**, 1139
- whole numbers. *See* Integers.

Nygaard, Kristen, 798–800

O

- .obj** file suffix, 48
- Object code, 48, 1174. *See also* Executable code.
- Object-oriented programming, 1174
 - “from day one,” 10
 - vs.* generic programming, 660
 - for graphics, benefits of, 504–505
 - history of, 781–783, 798–799
- Object, 60, 1174
 - aliases. *See* References.
 - behaving like a function. *See* Function object.
 - constructing, 182–183
 - copying, 1077–1078, 1081
 - current (**this**), 312–313
 - Date** example, 328–332
 - initializing, 322–325. *See also* Constructors.
 - layout in memory, 304, 497–499
 - lifetime, 1048–1049
 - named. *See* Variables.
 - Shape** example, 487
 - sizeof()**, 576–577
 - state, 301. *See also* Value.
 - type, 77–78
 - value. *See* Value.
- oct** manipulator, 378–379, 1130
- Octal number system, 377–379, 1041–1042
- Off-by-one error, 147
- ofstream**, 345–346
- Old-style casts, 1006
- On-time delivery, ideals, 776
- One-dimensional (1D) matrices, 865–868
- \ooo** octal, character literal, 1043
- OOP. *See* Object-oriented programming.
- Opaque types, 1026
- open()**, 346, 1126
- Open modes, 385–386
- Open shapes, 450–451
- Opening files, 344–346
 - See also* File I/O.
 - app** mode (“append”), 385
 - ate** mode (“at end”), 385
 - binary files, 386–389
 - binary** mode, 385
 - C-style I/O, 1140–1141
 - failure to open, 385
 - file streams, 344–346
 - in** mode (“for reading”), 385
 - nonexistent files, 385–386
 - open modes, 385–386
 - out** mode (“for writing”), 385
 - testing after opening, 346
 - trunc** mode (“truncate”), 385
- Open_polyline** example, 450–451, 489
- Operations, 66–69, 301, 1174
 - chaining, 178–179
 - graphics classes, 482–483
- operator**, 1003
- Operator overloading, 316
 - C++ standard operators, 317–318
 - restrictions, 317
 - user-defined operators, 317
 - uses for, 316–318
- Operator, 97
 - !** not, 1050
 - !=** not-equal (inequality), 1052
 - &** (unary) address of, 574, 1050

- &** (binary) bitwise and, 917, 1052, 1057
- &&** logical and, 1052, 1057
- &=** and and assign, 1053
- %** remainder (modulo), 1051
- %=** remainder (modulo) and assign, 1053
- *** (binary) multiply, 1051
- *** (unary) object contents, pointing to, 1050
- *=** multiply and assign, 1053
- +** add (plus), 1051
- ++** increment, 1050
- +=** add and assign, 1053
- subtract (minus), 1051
- decrement, 1050
- =** subtract and assign, 1053
- >** (arrow) member access, 1050–1051
- .** (dot) member access, 1050, 1051
- /** divide, 1051
- /=** divide and assign, 1053
- ::** scope resolution, 1049
- <** less than, 1052
- <<** shift left, 1051. *See also* **ostream**.
- <<=** shift left and assign, 1053
- <=** less than or equal, 1052
- =** assign, 1053
- ==** equal, 1052
- >** greater than, 1052
- >=** greater than or equal, 1052
- >>** shift right, 1051. *See also* **istream**.
- >>=** shift right and assign, 1053
- ?:** conditional expression (arithmetic if), 1053
- []** subscript, 1050
- ^** bitwise exclusive or, 1052, 1057
- ^=** xor and assign, 1053
- |** bitwise or, 1052, 1057
- |=** or and assign, 1053
- ||** logical or, 1053, 1057
- ~** complement, 1050
- additive operators, 1051
- const_cast**, 1050, 1058
- delete**, 1051, 1057–1058
- delete[]**, 1051, 1057–1058
- dereference. *See* Contents of.
- dynamic_cast**, 1050, 1058
- expressions, 1049–1059
- new**, 1050, 1051, 1057–1058
- reinterpret_cast**, 1050, 1058
- sizeof**, 1050, 1057
- static_cast**, 1050, 1058
- throw**, 1053
- typeid**, 1050
- Optimization, laws of, 893
- or**, synonym for **|**, 1003, 1004
- Order of evaluation, 287–288
- or_eq**, synonym for **|=**, 1003, 1004
- ostream**, 341–343, 1124–1125
 - <<**, text output, 815, 819
 - <<**, user-defined, 357–359
 - binary I/O, 386–389
 - connecting to output device, 1126
 - file I/O, **fstream**, 343–348, 1126
 - stringstream**, 390–391
 - using together with **stdio**, 1016–1017
- <ostream>**, 1096, 1124, 1129
- ostream_iterator** type, 758–761
- ostreamstring**, 390
- out** mode, 385, 1126
- Out-of-class member definition, 1074–1075
- Out-of-range conditions, 581
- Out_box** example, 439, 550–551
- out_of_range**, 147
- Output, 1174
 - See also* Input/output; I/O streams.
 - devices, 340–341
 - to file. *See* File I/O, writing files.
 - floating-point values, 380–381
 - format specifier **%**, 1141
 - formatting. *See* Input/output (formatting).
 - integers, 377–379
 - iterator, 722–723, 1103
 - operations, 1128–1129
 - streams. *See* I/O stream model.
 - to string. *See* **stringstream**.
 - testing, 961

Output `<<`, 47, 67, 1129
 complex, 881, 1139
 string, 815
 text output, 815, 819
 user-defined, 357–359
 Overflow, 854–858, 1174
 Overloading, 1067–1068, 1174
 alternative to, 516
 C++ and C, 992
 on **const**, 626–627
 linkage, 138
 operators. *See* Operator overloading.
 and overriding 500
 resolution, 1067–1068
 Override, 500–501, 1174

P

Padding, C-style I/O, 1143
pair, 1123–1124
 reading sequence elements, 1112–1113
 searching, 1117–1118
 sorting, 1117–1118
 Palindromes, example, 637–638
 Paradigm, 781–783, 1174
 Parameterization, function objects, 736–737
 Parameterized type, 659–661
 Parameters, 1174
 functions, 47, 113
 list, 113
 naming, 270–271
 omitting, 270
 templates, 656–659, 662–664
 Parametric polymorphism, 659–661
 Parsers, 188, 193
 functions required, 194
 grammar rules, 192–193
 Expression example, 187, 195–198, 200–201
 rules *vs.* tokens, 192–193
 Parsing
 expressions, 188–191
 grammar, English, 191–192
 grammar, programming, 188–191

 tokens, 188–191
partial_sort(), 1117
partial_sort_copy(), 1117
partial_sum(), 739, 1139
partition(), 1118
 Pascal language, 794–796
 Passing arguments
 by **const** reference, 273–276, 279–281
 copies of, 273
 modified arguments, 276–279
 by non-**const** reference, 279–281
 by reference, 276–281
 temporary objects, 280
 unmodified arguments, 275
 by value, 273, 279–281
 Patterns. *See* Regular expressions.
 Performance
 C++ and C, 990
 ideals, 775–776
 testing, 979–981
 timing, 981–983
 Permutations, 1120
 Petersen, Lawrence, 15
 Pictures. *See* Graphics.
 Pivoting, 875–876
 Pixels, 415–416
plus(), 1123
Point example, 441–423
pointer, 1108
 Pointers, 579–580
 See also Array; Iterators; Memory.
 * contents of, 579–580
 * pointer to (in declarations), 573, 1062
 -> (arrow) member access, 593, 1050–1051
 [] subscripting, 579–580
 arithmetic, 630–631
 array. *See* Pointers and arrays.
 casting. *See* Type conversion.
 to class objects, 591–593
 conversion. *See* Type conversion.
 to current object, **this**, 603–605
 debugging, 634–637
 declaration, C-style strings, 1015–1016

- decrementing, 630
- definition, 573–574, 1175
- deleted, 636
- explicit type conversion. *See* Type conversion.
- to functions, 1000–1002
- incrementing, 630
- initializing, 582–583, 635
- vs.* iterators, 1101
- literal (**0**), 1044–1045
- to local variables, 636–637
- moving around, 630
- to nonexistent elements, 635–636
- null, **0**, 583–584, 634–635, 1044–1045
- NULL** macro, 1144
- vs.* objects pointed to, 579
- out-of-range conditions, 581
- palindromes, example, 640–641
- ranges, 580–582
- reading and writing through, 579–582
- semantics, 619
- size, getting, 576–577
- subscripting [], 579–580
- this**, 654–655
- unknown, 593–595
- void***, 593–595
- Pointers and arrays
 - converting array names to, 631–633
 - pointers to array elements, 628–631
- Pointers and inheritance
 - polymorphism, 912–916
 - a problem 905–909
 - a solution, 909–912
 - user-defined interface class, 909–912
 - vector** alternative, 909–912
- Pointers and references
 - differences, 595–596
 - inheritance, 598
 - list example, 598–607
 - this** pointer, 603–605
 - parameters, 596–597
- polar()**, 881, 1138
- Polar coordinates, 881, 1138
- Polygon** example, 423–424, 453–455, 489
 - vs.* **Closed_polyline**, 453
 - invariants, 455
- Polyline example
 - closed, 451–453
 - marked, 468–469
 - open, 450–451
 - vs.* rectangles, 425–427
- Polymorphism
 - ad hoc, 659–661
 - embedded systems, 912–916
 - parametric, 659–661
 - run-time, 496
 - templates, 660–661
- Pools, embedded systems, 902–903
- Pop-up menus, 559
- pop_back()**, 1110
- pop_front()**, 1110
- pop_heap()**, 1120
- Portability, 11
 - C++, 1039
 - FLTK, 414, 1158
- Positioning in files, 389
- Post-conditions, 163–164, 961–962, 1175. *See also* Invariants.
- Post-decrement **--**, 1050, 1064
- Post-increment **++**, 1050, 1064
- Postfix expressions, 1050
- Pre-conditions, 161–163, 961–962, 1175. *See also* Invariants.
- Pre-decrement **--**, 1050, 1064
- Pre-increment **++**, 1050, 1064
- Precedence, in expressions, 1054
- Precision, numeric, 382–383, 854–858
- Predicates, 733
 - on class members, 737–738
 - function objects, 1122–1123
 - passing. *See* Function objects.
 - searching, 733–734
- Predictability, 893
 - error handling, 895
 - features to avoid, 894
 - memory allocation, 898, 902
- Preprocessing, 263

- Preprocessor directives
 - #define**, macro substitution, 1090–1091
 - #ifdef**, 1024
 - #ifndef**, 1025
 - #include**, including headers, 1090–1091
- Preprocessor, 1090
 - coding standards, 939
- prev_permutation()**, 1120
- Princeton University, 803
- print**, character class, 842, 1134
- Printable characters, identifying, 393
- printf()** family
 - %**, conversion specification, 1141
 - conversion specifications, 1141–1143
 - gets()**, 1018, 1144–1145
 - output formats, user-defined types, 1144
 - padding, 1143
 - printf()**, 1016–1017, 1141
 - scanf()**, 1017–1019, 1144–1145
 - stderr**, 1144
 - stdin**, 1144
 - stdio, 1144–1145
 - stdout**, 1144
 - synchronizing with I/O streams, 1016–1017
 - truncation, 1143
- Printing
 - error messages, 148–149
 - variable values, 246
- priority_queue** container adaptor, 1106
- Private, 308
 - base classes, 502
 - implementation details, 208, 302–304, 308–309
 - members, 484–485, 496, 501–502
 - private**: label, 302, 1003
- Problem analysis, 173
 - development stages, 174
 - estimating resources, 175
 - problem statement, 174–175
 - prototyping, 176
 - strategy, 174–176
- Problem statement, 174–175
- Procedural programming languages, 781
- Programmers
 - See also* Programming.
 - communication skills, 22
 - computation ideals, 92–94
 - skills requirements, 22–23
 - stereotypes of, 21–22
 - worldwide numbers of, 807
- Programming, xxiii, 1175
 - See also* Computation; Software.
 - abstract-first approach, 10
 - analysis stage, 35
 - author's philosophy, 6–9
 - bottom-up approach, 9
 - C first approach, 9
 - concept-based approach, 6
 - concrete-first approach, 6
 - depth-first approach, 6
 - design stage, 35
 - environments, 52
 - feedback, 36
 - generic, 1173
 - implementation, 35
 - magical approach, 10
 - object-oriented, 10, 1174
 - programming stage, 35
 - software engineering principles
 - first approach, 10
 - stages of, 35
 - testing stage, 35
 - top-down approach, 10
 - writing a program. *See* Calculator example.
- Programming languages, 783–784, 786, 807
 - Ada, 796–798
 - Algol family, 791–798
 - Algol60, 792–794
 - assemblers, 785
 - auto codes, 785
 - BCPL, 803
 - C, 800–804
 - C#, 796
 - C++, 804–806

- COBOL, 788–790
- Common Lisp, 790
- Delphi, 796
- Fortran, 786–788
- Lisp, 790–791
- Pascal, 794–796
- Scheme, 790
- Simula, 798–800
- Turbo Pascal, 796
- Programming philosophy, 772–773, 1175.
 - See also* C++ programs; Programming ideals; Programming languages.
- Programming ideals
 - abstraction level, 778–779
 - aims, 772–774
 - bottom-up approach, 776–777
 - code structure, 776
 - consistency, 780
 - correct approaches, 776–777
 - correctness, 775
 - data abstraction, 781
 - desirable properties, 773
 - direct expression of ideas, 777–778
 - efficiency, 775–776
 - generic programming, 782
 - KISS, 780
 - maintainability, 775
 - minimalism, 780
 - modularity, 779–780
 - multi-paradigm, 783
 - object-oriented programming, 781–783
 - on-time delivery, 776
 - overview, 774–775
 - paradigms, 781–783
 - performance, 775–776
 - philosophies, 772–774
 - procedural, 781
 - styles, 781–783
 - top-down approach, 776–777
- Programming, history, 783–784
 - See also* Programming languages.
- BNF (Backus-Naur) Form, 788, 793
- classes, 799
- CODASYL committee, 789
- early languages, 784–786
- first documented bug, 790
- first modern stored program, 784–786
- first programming book, 785
- function calls, 785
- functional programming, 788
- inheritance, 799
- K&R, 802
- lint, 801
- object-oriented design, 798–799
- STL (Standard Template Library), 805–806
- virtual functions, 799
- Programs, 44, 1175
 - See also* Computation; Software.
 - audiences for, 46
 - compiling. *See* Compilation.
 - computing values. *See* Expressions.
 - conforming, 1039
 - experimental. *See* Prototyping.
 - flow, tracing, 72
 - implementation defined, 1039
 - legal, 1039
 - linking, 51
 - not-conforming constructs, 1039
 - run. *See* Visual Studio; Command line.
 - starting execution, 46–47, 1039–1040
 - stored on a computer, 108
 - subdividing, 175–176
 - terminating, 206–207, 1039–1040
 - text of. *See* Source code.
 - translation units, 51
 - troubleshooting. *See* Debugging.
 - unspecified constructs, 1039
 - valid, 1039
 - writing, example. *See* Calculator example.
 - writing your first, 45–47
- Program organization
 - See also* Programming ideals.
 - abstraction, 92–93
 - divide and conquer, 93
- Projects, Visual Studio, 1153–1154

Promotions, 98–99, 1054–1055
 Prompting for input, 61
 >, input prompt, 221
 calculator example, 177
 sample code, 220–223
 Proofs, testing, 952
protected, 484–485, 496, 502, 1003
 Prototyping, 176
 Pseudo code, 177, 1175
 Public, 302, 1003
 base class, 500–501
 interface, 208, 488–491
 member, 302
 public by default, **struct**, 303
 public: label, 302
punct, punctuation character class,
 842, 1134
Punct_stream example, 397–401
 Pure virtual functions, 487, 1175
push_back()
 growing a **vector**, 118–119
 queue operations, 1110
 resizing **vector**, 652–653
 stack operations, 1110
 string operations, 1132
push_front(), 1110
push_heap(), 1119
put(), 1129
putback()
 naming convention, 209
 putting tokens back, 204–205
 return value, disabling, 210
putc(), 1145
putchar(), 1145
 Putting back input, 204–206

Q

qsort(), 1149
<queue>, 1095
queue container adaptor, 1106
 Queue operations, 1110

R

\r carriage return, character literal,
 1043

r, reading file mode, 1140
++, reading and writing file mode, 1140
 RAII (Resource Acquisition Is Initialization)
 definition, 1175
 exceptions, 675–676, 1087
 testing, 964–965
 for **vector**, 678–680
rand(), 878, 1149
<random>, 1096
 Random numbers, 877–879
 Random-access iterators, 723, 1104
 Range
 definition, 1175
 errors, 146–148
 pointers, 580–582
 regular expressions, 841–842
 Range checking
 [], 628–631, 668–671
 arrays, 628–631
 at(), 668–669
 compatibility, 670
 constraints, 670
 design considerations, 670–671
 efficiency, 670
 exceptions, 668–669
 macros, 671–672
 optional checking, 671
 overview, 668–669
 pointer, 628–631
 vector, 668–671
rbegin(), 1109
 Re-throwing exceptions, 677, 1087
read(), unformatted input, 1128
 Readability
 expressions, 95
 indenting nested code, 269
 nested code, 269
 Reading
 dividing functions logically, 353–
 356
 files. *See* Reading files.
 with iterators, 1101–1102
 numbers, 212–213
 potential problems, 352–357
 separating dialog from function,
 356–357

- a series of values, 350–352
 - a single value, 352–357
 - into **strings**, 815
 - tokens, 183–184
- Reading files
 - binary I/O, 387
 - converting representations, 368–370
 - to end of file, 360
 - example, 346–348
 - fstream** type, 344–346
 - ifstream** type, 344–346
 - in-memory representation, 362–364
 - input loops, 359–361
 - istream** type, 343–348, 387
 - ostream** type, 387
 - process steps, 344
 - structured files, 361–370
 - structured values, 364–368
 - symbolic representations, 368–370
 - terminator character, specifying, 360
- real()**, 881, 1138
- Real numbers, 855
- Real part, 881
- Real-time constraints, 893
- Real-time response, 890
- realloc()**, 1010, 1147
- Recovering from errors, 238–240, 349–352. *See also* Error handling; Exceptions.
- Rectangle** example, 424–427, 455–459, 489
- Recursion
 - definition, 1175
 - infinite, 196, 1173
 - looping, 198
- Recursive function calls, 286
- Red-black trees, 747. *See also* Associative containers; **map**.
- Red margin alerts, 3
- Reference semantics, 619
- References, 227, 1175
 - See also* Aliases.
 - &** in declarations, 273–277
 - to arguments, 274–276
 - circular. *See* Circular reference.
 - to last **vector** element, **back()**, 708
 - vs.* pointers. *See* Pointers and references.
- <regex>**, 1096, 1131
- regex**. *See* Regular expressions.
- regex_error** exception, 1099
- regex_match()**, 1133
 - vs.* **regex_search()**, 847
- regex_search()**, 1133
 - vs.* **regex_match()**, 847
- Regression tests, 953
- Regular expressions, 830–832, 836, 1175
 - See also* **regex** pattern matching.
 - character classes, 837–838
 - error handling, 842–844
 - grouping, 831, 837, 840
 - syntax. *See* **regex** operators.
 - uses for, 830
 - ZIP code example, 844–849
- regex** pattern matching, 830–832
 - \$** end of line, 837, 1134
 - ()** grouping, 831, 837, 840
 - *** zero or more occurrences, 832, 837–838
 - +** one or more occurrences, 837, 838–839
 - range specification, 841
 - .** wildcard, 837
 - ?** optional occurrence, 831–832, 837, 838–839
 - []** character class, 837
 - ** escape character, 830–831, 837
 - ** as literal, 841
 - ^** negation, 837
 - ^** start of line, 837
 - {}** count, 831, 837–839
 - |** alternative (or), 831–832, 837, 840–841
 - alternation, 840–841
 - character classes. *See* **regex** character classes.
 - character sets, 841–842
 - definition, 834
 - grouping, 840
 - matches**, 834

- regex** pattern matching (*continued*)
 - pattern matching, 836–837
 - ranges, 841–842
 - regex_match()**, 1133
 - regex_search()**, 1133
 - repeating patterns, 838–840
 - searching with, 833–836, 844
 - smatch**, 834
 - special characters. *See* **regex** operators.
 - sub-patterns, 831, 834
 - regex** operators, 837, 1133–1134
- regex** character classes, 841–842
 - alnum**, 842
 - alpha**, 842
 - blank**, 842
 - cntrl**, 842
 - d**, 842
 - \D**, 838
 - \d**, 837
 - digit**, 842
 - graph**, 842
 - \L**, 838
 - \l**, 837
 - lower**, 842
 - print**, 842
 - punct**, 842
 - regex_match()** *vs.* **regex_search()**, 847
 - s**, 842
 - \S**, 838
 - \s**, 837
 - space**, 842
 - \U**, 838
 - \u**, 837
 - upper**, 842
 - w**, 842
 - \W**, 837
 - \w**, 837
 - xdigit**, 842
- Regularity, 376
- reinterpret_cast**, 594–595, 1058
 - casting unrelated types, 594
 - hardware access, 905
- Relational operators, 1052
- Reliability, software, 34, 890
- Remainder and assign **%=**, 1053
- Remainder **%** (modulo), 66, 1051
 - correspondence to ***** and **/**, 68
 - floating-point, 199, 228–231
 - integer and floating-point, 66
- remove()**, 1115
- remove_copy()**, 1115
- remove_copy_if()**, 1115
- rend()**, 1109
- Repeated words examples, 71–74
- Repeating patterns, 192–193
- Repetition, 1134. *See also* Iterating; **regex**.
- replace()**, 1114
- replace_copy()**, 1114–1115
- Reporting errors
 - Date** example, 313–314
 - debugging, 157
 - error()**, 140–141
 - run-time, 143–144
 - syntax errors, 135–136
- Representation, 301, 649–651
- Requirements, 1175
 - See also* Invariants; Post-conditions;
 - Pre-conditions.
 - for functions, 151
- reserve()**, 651–652, 717, 1111
- Reserved names, 75–76. *See also* Key-words.
- resetiosflags()** manipulator, 1130
- resize()**, 652, 1111
- Resource, 1175
 - leaks, 893, 896
 - limitations, 890
 - management. *See* Resource management.
 - testing, 962
- vector** example, 672–673
- Resource Acquisition Is Initialization (RAII), 1175
 - exceptions, 675–676, 1087
 - testing, 964–965
 - for **vector**, 678–680
- Resource management, 672–677
 - See also* **vector** example.
- auto_ptr**, 678
- basic guarantee, 677
- error handling, 677

- guarantees, 676–678
- make_vec()**, 677
- no-throw guarantee, 677
- problems, 673–675
- RAII, 675–676, 678–680
- resources, examples, 672–673
- strong guarantee, 677
- testing, 964–965
- Results, 91. *See also* Return values.
- return** statement, 271–272
- Return types, functions, 47, 270–271
- Return values, 112
 - functions, 1066
 - no return value, **void**, 210
 - omitting, 113
 - returning, 271–272
- reverse()**, 1115
- reverse_copy()**, 1115
- reverse_iterator**, 1108
- Revision history, 236–237
- Rho, 881
- Richards, Martin, 803
- right** manipulator, 1130
- Ritchie, Dennis, 801, 806, 988–989, 998
- Robot-assisted surgery, 30
- rotate()**, 1115
- rotate_copy()**, 1115
- Rounding, 382, 1175
 - See also* Truncation.
 - errors, 855
 - floating-point values, 382–383
- Rows, matrices, 864–865, 870
- Rules, for programming. *See* Ideals.
- Rules, grammatical, 192–193
- Run-time dispatch, 496. *See also* Virtual functions.
- Run-time errors. *See* Errors, run-time.
- Run-time polymorphism, 496
- runtime_error**, 140, 149, 151
- Rvalues, 94–95, 1054

S

- s**, character class, 842, 1134
- \s**, “not space,” **regex**, 838, 1135
- \s**, “space,” **regex**, 837, 1135
- Safe conversions, 79–80
- Safety, type. *See* Type, safety.
- Scaffolding, cleaning up, 233–234
- scale_and_add()** example, 868
- scale_and_multiply()** example, 876
- Scaling data, 531
- scanf()**, 1018, 1144–1145
- Scenarios. *See* Use cases.
- Scheme language, 790
- scientific** format, 383
- scientific** manipulator, 381, 1130
- Scope, 264–265, 1046–1047, 1175
 - class, 264, 1046
 - enumerators, 316
 - global, 264, 267, 1046
 - going out of, 266–267
 - kinds of, 264–265
 - local, 265–266, 1046
 - namespace, 264, 269, 1046
 - resolution **::**, 291, 1049
 - statement, 265, 1046
- Scope and nesting
 - blocks within functions, 268–269
 - classes within classes, 268
 - classes within functions, 268
 - functions within classes, 267
 - functions within functions, 268
 - indenting nested code, 269
 - local classes, 268
 - local functions, 268
 - member classes, 268
 - member functions, 267
 - nested blocks, 268–269
 - nested classes, 268
 - nested functions, 268
- Scope and object lifetime, 1048–1049
 - free-store objects, 1048
 - local (automatic) objects, 1048
 - namespace objects, 1048
 - static class members, 1048
 - temporary objects, 1048
- Scope and storage class, 1047–1048
 - automatic storage, 1047
 - free store (heap), 1047
 - static storage, 1047

Screens

See also GUIs (graphical user interfaces).

data graph layout, 530–531

drawing on, 419–420

labeling, 421

search(), 763–764, 1113

Searching

See also **find()**; **find_if()**; Finding;

Matching.

algorithms for, 1117–1118

binary searches, 747, 763–764

in C, 1149

for characters, 711

(key,value) pairs, by key. *See* Associative containers.

for links, 600–602

map elements. *See* **unordered_map**.

predicates, 733

with regular expressions, 833–836, 844–849, 1133–1135

search_n(), 1113

Self reference. *See* **this**.

Self assignment, 654

Self-checking, error handling, 896

Separators, nonstandard, 394–401

Sequence containers, 1105

Sequences, 694, 1175

algorithms. *See* standard library algorithms.

differences between adjacent elements, 739

empty, 702

example, 696–698

half open, 694–695

Sequencing rules, 192–193

Server farms, 31

set, 744, 755–757

iterators, 1105

vs. **map**, 756

subscripting, 756

set(), 590–591

<set>, 744, 1095

Set algorithms, 1118–1119

setbase() manipulator, 1130

set_difference(), 1119

setfill() manipulator, 1130

set_intersection(), 1119

setiosflags() manipulator, 1130

setprecision() manipulator, 382–383, 1130

set_symmetric_difference(), 1119

set_union(), 1119

setw() manipulator, 1130

Shallow copies, 619

Shape example, 485–486

abstract classes, 487–488

access control, 488–491

attaching to **Window**, 533–534

as base class, 441, 487–488

clone(), 496

copying objects, 494–496

draw(), 491–494

draw_lines(), 491–494

fill color, 492

implementation inheritance, 504–505

interface inheritance, 504–505

line visibility, 492

move(), 494

mutability, 494–496

number_of_points(), 445

object layout, 497–499

object-oriented programming, 504–505

point(), 445

slicing shapes, 495

virtual function calls, 493, 498–499

Shift operators, 1051

Shipping, computer use, 26–28

short, 917, 1062

Shorthand notation, regular expressions, 1135

showbase, manipulator, 379, 1129

showpoint, manipulator, 1129

showpos, manipulator, 1129

Shuffle algorithm, 1115–1116

Signed and unsigned integers, 922–926

signed type, 1062

Simple_window, 418–420, 439

Simplicity ideal, 92–94

Simula language, 798–800

sin(), sine, 879, 1137

Singly-linked lists, 598, 698

sinh(), hyperbolic sine, 879, 1137

Size

bit strings, 916–917

containers, 1110–1111

getting, **sizeof()**, 576–577

of numbers, 854–858

vectors, getting, 118–119

size()

container capacity, 1111

number of elements, 118, 815

string length, 815, 1132

vectors, 118, 121

sizeof(), 576–577, 1057

object size, 1050

value size, 856

size_type, 704, 1108

skipws, 1129

slice(), 865–866, 869

Slicing

matrices, 865–866, 869

objects, 495

Smallest integer, finding, 879

smatch, 834

Soft real-time, 893

Software, 19, 1175

See also Programming; Programs.

affordability, 34

correctness, 34

ideals, 34–37

maintainability, 34

reliability, 34

troubleshooting. *See* Debugging.

useful design, 34

uses for, 19–32

“Software engineering principles first”

approach to programming, 10

Software layers, GUIs, 544–545

sort(), 728, 762–763, 1117

sort_heap(), 1120

Sorting

algorithms for, 1117–1118

in C, **qsort()**, 1149

sort(), 728, 762–763, 1117

Source code

definition, 48, 1175

entering, 1154

Source files, 48, 1175

adding to projects, 1154

space, 842, 1134

Space exploration, computer use, 32–33

Special characters, 1043–1044

regular expressions, 1133–1134

Specialization, 658–659, 1084–1085

Specifications

definition, 1175

source of errors, 134

Speed of light, 96

sprintf(), 1141

sqrt(), square root, 879, 1137

Square of **abs()**, norm, 881

srand(), 878, 1149

<sstream>, 1096

stable_partition(), 1118

stable_sort(), 1117

<stack>, 1096

stack container adaptor, 1106

Stack of activation records, 284

Stack storage, 577

Stacks

container operations, 1110

embedded systems, 902, 903–904, 897–898

growth, 284–287

unwinding, 1088

Stages of programming, 35

Standard

conformance, 801, 935, 1039

ISO, 1039, 1175

manipulators. *See* Manipulators.

mathematical functions, 879–880

Standard library

See also C standard library; STL

(Standard Template Library).

algorithms. *See* Algorithms.

complex. *See* **complex**.

containers. *See* Containers.

C-style I/O. *See* **printf()** family.

C-style strings. *See* C-style strings.

date and time, 1147–1149

- Standard library (*continued*)
 - function objects. *See* Function objects.
 - I/O streams. *See* Input; Input/output; Output.
 - iterators. *See* Iterators.
 - mathematical functions. *See* Mathematical functions (standard).
 - numerical algorithms. *See* Algorithms (numerical); Numerics.
 - string**. *See* **string**.
 - time, 982–983, 1147–1149
 - valarray**. *See* **valarray**.
- Standard library header files, 1095–1097
 - algorithms, 1095–1096
 - C standard libraries, 1097
 - containers, 1095–1096
 - I/O streams, 1096
 - iterators, 1095–1096
 - numerics, 1096
 - string manipulation, 1096
 - utility and language support, 1097
- Standard library I/O streams, 1124–1125. *See also* I/O streams.
- Standard library string manipulation
 - character classification, 1131
 - containers. *See* **vector**; **map**; **set**; **unordered_map**.
 - input/output. *See* I/O streams.
 - regular expressions. *See* **regex**.
 - string manipulation. *See* **string**.
- Stanford University, 791
- Starting programs, 1039–1040. *See also* **main()**.
- State, 90, 1175
 - I/O stream, 1127
 - of objects, 301
 - source of errors, 134
 - testing, 961
 - valid state, 309
 - validity checking, 309
- Statement scope, 265, 1046
- Statements, 47
 - grammar, 1059–1061
 - named sequence of. *See* Functions.
 - terminator **;** (semicolon), 50, 99
- Static storage, 577, 1047
 - class members, lifetime, 1048
 - embedded systems, 897–898, 905
 - static**, 1047
 - static const**, 321. *See also* **const**.
 - static const int** members, 1075
 - static** local variables, order of initialization, 290
- std** namespace, 291–292, 1098
- stderr**, 1144
- <stdexcept>**, 1097
- stdin**, 1016, 1144. *See also* **stdio**.
- stdio**, standard C I/O, 1016, 1144–1145
 - EOF** macro, 1019–1020
 - errno**, error indicator, 880
 - fclose()**, 1019–1020
 - FILE**, 1019–1020
 - fopen()**, 1019–1020
 - getchar()**, 1019, 1044
 - gets()**, 1018, 1144–1145
 - input, 1017–1019
 - output, 1016–1017
 - printf()**, 1016–1017, 1141–1144
 - scanf()**, 1018, 1144
 - stderr**, **cerr** equivalent, 1144
 - stdin**, **cin** equivalent, 1016, 1144
 - stdout**, **cout** equivalent, 1016, 1144
- std_lib_facilities.h** header file, 1153–1154
- stdout**, 1016, 1144. *See also* **stdio**.
- Stepanov, Alexander, 694, 696, 805–806
- Stepping through code, 160
- Stereotypes of programmers, 21–22
- STL (Standard Template Library), 690, 1110–1124
 - See also* C standard library; Standard library.
 - algorithms. *See* STL algorithms.
 - containers. *See* STL containers.
 - function objects. *See* STL function objects.
 - history of, 805–806
 - ideals, 690–694

- iterators. *See* STL iterators.
 - namespace, [std](#), 1098
- STL algorithms, 1112–1121
 - See* Algorithms (STL).
 - alternatives to, 1150
 - built-in arrays, 718–719
 - computation *vs.* data, 691–693
 - heap, 1119–1120
 - [max\(\)](#), 1120–1121
 - [min\(\)](#), 1120–1121
 - modifying sequence, 1114–1116
 - mutating sequence, 1114–1116
 - nonmodifying sequence, 1113–1114
 - permutations, 1120
 - searching, 1117–1118
 - set, 1118–1119
 - shuffle, 1115–1116
 - sorting, 1117–1118
 - utility, 1116–1117
 - value comparisons, 1120–1121
- STL containers, 720–721, 1105–1112
 - almost, 721–722, 1106
 - assignments, 1108–1109
 - associative, 1105, 1111–1112
 - capacity, 1110–1111
 - comparing, 1111
 - constructors, 1108–1109
 - container adaptors, 1106
 - copying, 1111
 - destructors, 1108–1109
 - element access, 1109
 - information sources about, 720–721
 - iterator categories for, 722–723, 1104–1105, 1109
 - list operations, 1110
 - member types, 1108
 - operations overview, 1107
 - queue operations, 1110
 - sequence, 1105
 - size, 1110–1111
 - stack operations, 1110
 - swapping, 1111
- STL function objects, 1122–1123
 - adaptors, 1123
 - arithmetic operations, 1123
 - inserters, 1121–1122
 - predicates, 738–738, 1122–1123
- STL iterators, 1100–1105
 - basic operations, 695
 - categories, 1103–1105
 - definition, 694, 1100–1101
 - description, 695–696
 - empty lists, 702
 - example, 708–711
 - operations, 1102–1103
 - vs.* pointers, 1101
 - sequence of elements, 1101–1102
- Storage class, 1047–1048
 - automatic storage, 1047
 - free store (heap), 1047
 - static storage, 1047
- Storing data. *See* Containers.
- [str\(\)](#), [string](#) extractor, 390–391
- [strcat\(\)](#), 1012–1013, 1146
- [strchr\(\)](#), 1014, 1146
- [strcmp\(\)](#), 1012–1013, 1146
- [strcpy\(\)](#), 1012–1013, 1015, 1146
- Stream
 - buffers, 1125
 - iterators, 758–761
 - modes, 1126
 - states, 349
 - types, 1126
- [streambuf](#), 402, 1125
- [<streambuf>](#), 1096
- [<string>](#), 1096, 1128
- String literal, 62, 1044
- [string](#), 66, 815, 1175
 - See also* Text.
 - [+](#) concatenation, 68–69, 815, 1132
 - [+=](#) append, 815
 - [<](#) lexicographical comparison, 815
 - [<<](#) output, 815
 - [=](#) assign, 815
 - [==](#) equal, 815
 - [>>](#) input, 815
 - [\[\]](#) subscripting, 815
 - almost container, 1106
 - [append\(\)](#), 815
 - [basic_string](#), 816

string (*continued*)

C++ to C-style conversion, 815
c_str(), C++ to C-style conversion, 345, 815
erase(), removing characters, 815
 exceptions, 1099
find(), 815
from_string(), 817–818
getline(), 815
 input terminator (whitespace), 65
insert(), adding characters, 815
length(), number of characters, 815
lexical_cast example, 819
 literals, debugging, 159
 operations, 815, 1132
 operators, 66–67, 68
 palindromes, example, 637–638
 pattern matching. *See* Regular expressions.
 properties, 712–713
 size, 78
size(), number of characters, 815
 standard library, 816
string to value conversion, 817–818
stringstream, 816–818
 subscripting [], 815
to_string() example, 816–818
 values to string conversion, 816
vs. **vector**, 715
 whitespace, 818–819
stringstream, 390–391, 816–818, 1126
strlen(), 1012, 1146
strncat(), 1012–1013, 1146
strncmp(), 1012–1013, 1146
strncpy(), 1012–1013, 1146
 Strong guarantee, 677
 Stroustrup, Bjarne
 advisor, 785
 biography, 14–15
 Bell Labs colleagues, 801–804, 989
 education on invariants, 793
 inventor of C++, 804–806
 Kristen Nygaard, 799
strpbrk(), 1146
strrchr(), 1146
strstr(), 1146

strtod(), 1146
strtol(), 1146
strtoul(), 1146
struct, 303–304. *See also* Data structures.
struct tag namespace, 1002–1003
 Structure
 of data. *See* Data structures.
 of programs, 213–214
 Structured files, 361–370
 Style, definition, 1176
 Sub-patterns, 831, 834
 Subclasses, 496, 1116. *See also* Derived classes.
 Subdividing programs, 175–176
 Subscripting, 116–117
 () Fortran style, 863
 [] C Style, 669, 863
 arrays, 628, 863
 at(), checked subscripting, 669, 1109
 Matrix example, 863–865, 869
 pointers, 1064
 string, 815, 1132
 vector, 579–580, 592–593, 625–626
 Substrings, 827
 Subtract and assign **--**, 67, 1053, 1103
 Subtraction **-** (minus)
 complex, 881, 1138
 definition, 1051
 integers, 1064
 iterators, 1104
 pointers, 1064
 Subtype, definition, 1176
 Summing values. *See* **accumulate()**.
 Superclasses, 496, 1176. *See also* Base classes.
swap(), 279, 1111, 1116
 Swapping
 columns, 870
 containers, 1111
 ranges, 1116
 rows, 870, 876
swap_ranges(), 1116
switch-statements
 break, **case** termination, 104–107

- case** labels, 104–107
 - most common error, 107
 - vs.* **string**-based selection, 105
- Symbol tables, 246
- Symbolic constants
 - See also* Enumerations.
 - cleaning up, 231–233
 - defining, with **static const**, 321
- Symbolic names, tokens, 232
- Symbolic representations, reading, 368–370
- Syntax analyzers, 188
- Syntax checking, 48–50
- Syntax errors
 - examples, 48–50
 - overview, 135–136
 - reporting, 135–136
- Syntax macros, 1023–1024
- system()**, 1149
- System, definition, 1176
- System tests, 969–973

T

- \t** tab character, 108, 1043
- tan()**, tangent, 879, 1137
- tanh()**, hyperbolic tangent, 879, 1137
- TEA (Tiny Encryption Algorithm), 785, 930–935
- Technical University of Copenhagen, 793
- Telecommunications, 28–29
- Temperature data, example, 119–121
- template**, 1003
- Template, 656, 1083, 1176
 - arguments, 1083–1084
 - class, 658–661. *See also* Class template.
 - compiling, 661
 - containers, 661–662
 - error diagnostics, 661
 - function, 659–665. *See also* Function template.
 - generic programming, 659–661
 - inheritance, 661–662
 - instantiation, 658–659, 1084–1085
 - integer parameters, 662–664
 - member types, 1086
 - parameters, 656–659, 662–664
 - parametric polymorphism, 659–661
 - specialization, 1084–1085
 - type parameters, 656–659
 - typename**, 1086
 - weaknesses, 661
- Template-style casts, 1006
- Temporary objects, 280, 1048
- Terminals, in grammars. *See* Tokens.
- Termination
 - abort()** a program, 1149
 - on exceptions, 140
 - exit()** a program, 1149
 - input, 61–62, 177
 - normal program termination, 1039–1040
 - for **string** input, 65
 - zero, for C-style strings, 633
- Terminator character, specifying, 360
- Testing, 952–953, 1176
 - See also* Debugging.
 - algorithms, 961–968
 - for bad input, 102–103
 - black box, 952–953
 - branching, 966–968
 - bug reports, retention period, 953
 - calculator example, 223–227
 - classes, 973–976
 - code coverage, 968
 - debugging, 979
 - dependencies, 962–963
 - designing for, 978–979
 - faulty assumptions, 976–978
 - files, after opening, 346
 - FLTK, 1160
 - inputs, 961
 - loops, 965–966
 - non-algorithms, 961–968
 - outputs, 961
 - performance, 979–983
 - pre- and post-conditions, 961–962
 - proofs, 952
 - RAII, 964–965

Testing (*continued*)

- regression tests, 953
- resource management, 964–965
- resources, 962
- stage of programming, 35
- state, 961
- system tests, 969–973
- test cases, definition, 164
- test harness, 957–959
- timing, 981–983
- white box, 952–953

Testing units

- formal specification, 954–955
- random sequences, 959–961
- strategy for, 955–957
- systematic testing, 954–955
- test harness, 957–959

Text

- character strings. *See* **string**; C-style strings.
- email example, 820–825, 828–830
- extracting text from files, 820–825, 828–830
- finding patterns, 828–830, 833–836
- in graphics. *See* Text.
- implementation details, 826–828
- input/output, GUIs, 550–551
- maps. *See* **map**.
- storage, 577
- substrings, 827
- vector** example, 121–123
- words frequency example, 745–747

Text example, 427–429, 462–464

Text editor example, 708–711

Theta, 881

this pointer, 603–605, 654–655

Thompson, Ken, 801–803

Three-way comparison, 1012

Throwing exceptions, 145, 1086

- I/O stream, 1127

- re-throwing, 677

- standard library, 1099–1100

- throw, 145, 1053, 1086–1088

- vector**, 672–673

Time

- date and time, 1147–1149

- measuring, 981–983

Timekeeping, computer use, 26

time_t, 1147

Tiny Encryption Algorithm (TEA),
785, 930–935

tm, 1147

Token example, 181–184

Token_stream example, 204–212

tolower(), 394, 1131

Top-down approach, 10, 776–777

to_string() example, 816–818

toupper(), 394, 1131

Tracing code execution, 160–161

Trade-off, definition, 1176

transform(), 1114

Transient errors, handling, 895–896

Translation units, 51, 137–138

Transparency, 447, 457

Tree structure, **map** container, 747–750

true, 1003, 1004

trunc mode, 385, 1126

Truncation, 82, 1176

- C-style I/O, 1143

- exceptions, 151

- floating-point numbers, 857

try-catch, 144–150, 668–669, 1003

Turbo Pascal language, 796

Two-dimensional matrices, 868–870

Two's complement, 922

Type conversion

- casting, 594–595

- const_cast**, casting away **const**,
594–595

- exceptions, 151

- explicit**, 594

- in expressions, 98–99

- function arguments, 281–282

- implicit, 621–622

- int** to pointer, 576

- operators, 1058–1059

- pointers, 576, 594–595

- reinterpret_cast**, 594

- safety, 79–83

- static_cast**, 594

- string** to value, 817–818
- truncation, 82
- value to **string**, 816
- Type conversion, implicit, 621–622
 - bool**, 1055
 - compiler warnings, 1055–1056
 - floating-point and integral, 1055
 - integral promotion, 1054–1055
 - pointer and reference, 1055
 - preserving values, 1054–1055
 - promotions, 1054–1055
 - user-defined, 1056
 - usual arithmetic, 1056
- Type safety, 78–79
 - implicit conversions, 80–83
 - narrowing conversions, 80–83
 - pointers, 580–583, 634–637
 - range error, 146–148, 580–582
 - safe conversions, 79–80
 - unsafe conversions, 80–83
- typedef**, 703
- typeid**, 1003, 1050, 1099
- <typeinfo>**, 1097
- typename**, 1003, 1086
- Type, 60, 77, 1176
 - aliases, 703
 - built-in. *See* Built-in types.
 - checking, C++ and C, 998–999
 - generators, 658–659
 - graphics classes, 480–482
 - mismatch errors, 136–137
 - mixing in expressions, 98–99
 - naming. *See* Namespaces.
 - objects, 77–78
 - operations, 301
 - organizing. *See* Namespaces.
 - parameterized, 659–661. *See also* Templates.
 - as parameters. *See* Templates.
 - pointers. *See* Pointer.
 - promotion, 98–99
 - representation of object, 304, 497–499
 - safety, 78–79, 82
 - subtype, 1176
 - supertype, 1176

- truncation, 82
- user-defined. *See* UDTs.
- uses for, 300
- values, 77
- variables. *See* Variables, types.

U

- u/U** suffix, 1041
- \U**, “not uppercase,” **regex**, 838, 1135
- \u**, “uppercase character,” **regex**, 837, 1135
- UDT (User-defined type). *See* Class; Enumeration.
- Unary expressions, 1050–1051
- “Uncaught exception” error, 151
- Unchecked conversions, 905
- “Undeclared identifier” error, 256
- Undefined order of evaluation, 261
- unset()**, 349–352
- unsetc()**, 1145
- Uninitialized variables, 322–325, 1176
- uninitialized_copy()**, 1116–1117
- uninitialized_fill()**, 1116–1117
- union**, 1082–1083
- unique()**, 1114
- unique_copy()**, 728, 757, 760–761, 1114
- Unit tests
 - formal specification, 954–955
 - random sequences, 959–961
 - strategy for, 955–957
 - systematic testing, 954–955
 - test harness, 957–959
- Unnamed objects, 459–461
- <unordered_map>**, 744, 1096
- unordered_map**, 744
 - See also* **map**.
 - finding elements, 753–755
 - hash tables, 753
 - hash values, 753
 - hashing, 753
 - iterators, 1105
- unordered_multimap**, 744, 1105
- unordered_multiset**, 744, 1105
- <unordered_set>**, 744, 1096
- unordered_set**, 744, 1105

- Unsafe conversions, 80–83
- unsetf()**, 380
- Unsigned and signed, 922–926
- unsigned** type, 1062
- Unspecified constructs, 1039
- upper**, character class, 842, 1134
- upper_bound()**, 764, 1112, 1117
- Uppercase. *See* Case.
- uppercase**, 1129
- U.S. Department of Defense, 796
- U.S. Navy, 789–790
- Use cases, 177, 1176
- User-defined conversions, 1056
- User-defined operators, 1054
- User-defined types (UDTs), 300
 - See also* Class; Enumeration.
 - exceptions, 1087
 - operator overloading, 1069–1070
 - operators, 1070
 - standard library types, 300
- User interfaces
 - console input/output, 540
 - graphical. *See* GUI.
 - web browser, 540–541
- using** declarations, 291–293
- using** directives, 291–293, 1089
- Usual arithmetic conversions, 1056
- Utilities, STL
 - function objects, 1122–1123
 - inserters, 1121–1122
 - make_pair()**, 1124
 - pair**, 1123–1124
- <utility>**, 1096, 1123–1124
- Utility algorithms, 1116–1117
- Utility and language support, header
 - files, 1097

V

- \v** vertical tab, character literal, 1043
- valarray**, 1106, 1139
- <valarray>**, 1096
- Valid programs, 1039
- Valid state, 309
- Validity checking, 309
 - constructors, 309
 - enumerations, 315
 - invariants, 309
 - rules for, 309
- Value semantics, 619
- value_comp()**, 1112
- Values, 77–78, 1176
 - symbolic constants for. *See* Enumerations.
 - and variables, 62, 73–74, 242
- value_type**, 1108
- Variables, 62–63, 116–117, 1061–1062
 - ++** increment, 73–74
 - =** assignment, 69–73
 - changing values, 73–74
 - composite assignment operators, 73–74
 - constructing, 287–288
 - declarations, 258, 260–261
 - going out of scope, 287
 - incrementing **++**, 73–74
 - initialization, 69–73
 - input, 60
 - naming, 74–77
 - type of, 66–67
 - uninitialized, class interfaces, 322–325
 - value of, 73–74
- <vector>**, 1096
- vector** example, 570–573, 612–618, 646–656
 - >** access through pointer, 593
 - .** (dot) access, 592–593
 - =** assignment, 653
 - []** subscripting, 625–626, 668–672
 - allocators, 666
 - at()**, checked subscripting, 669
 - changing size, 646–656
 - copying, 613–618
 - destructor, 506–590
 - element type as parameter, 656–659
 - erase()** (removing elements), 715–718
 - exceptions, 668–669, 678–680
 - explicit** constructors, 621–622
 - inheritance, 661–662

- `insert()` (adding elements), 715–718
 - overloading on `const`, 626–627
 - `push_back()`, 652–653, 667
 - representation, 650–651
 - `reserve()`, 651, 667, 679–680
 - `resize()`, 652, 668
 - subscripting, 579–580, 592–593, 625–626
 - vector**, standard library, 1107–1111
 - < less than, 1111
 - = assignment, 1109
 - == equality, 1111
 - [] subscripting, 1109
 - `assign()`, 1109
 - `at()`, checked subscripting, 1109
 - `back()`, reference to last element, 1109
 - `begin()`, iterator to first element, 1109
 - `capacity()`, 1111
 - `const_iterator`, 1108
 - constructors, 1108–1109
 - destructor, 1109
 - `difference_type`, 1108
 - `end()`, one beyond last element, 1109
 - `erase()`, removing elements, 1110
 - `front()`, reference to first element, 1109
 - `insert()`, adding elements, 1110
 - `iterator`, 1108
 - member functions, lists of, 1108–1111
 - member types, list of, 1108
 - `push_back()`, add element at end, 1110
 - `size()`, number of elements, 1111
 - `size_type`, 1108
 - `value_type`, 1108
 - vector** of references, simulating, 1166–1167
 - Vector_ref** example, 440, 1166–1167
 - virtual**, 1003
 - Virtual destructors, 590. *See also* Destructors.
 - Virtual function, 493, 498–499
 - declaring, 499–500
 - definition, 493, 1176
 - history of, 799
 - object layout, 497–499
 - overriding, 500–501
 - pure, 502–504
 - Shape** example, 493, 498–499
 - `vp_ptr`, 498–499
 - `vtbl`, 498
 - Visibility
 - See also* Scope; Transparency.
 - menus, 560–561
 - of names, 264–269, 290–293
 - widgets, 549
 - Visual Studio
 - FLTK (Fast Light Toolkit), 1159–1160
 - installing, 1152
 - running programs, 1153–1154
 - void**, 113
 - function results, 113, 270, 272
 - pointer to, 593–595
 - `putback()`, 210
 - void***, 593–595, 1007–1008, 1062
 - `vp_ptr`, virtual function pointer, 498–499
 - `vtbl`, virtual function table, 498
- ## W
- w**, writing file mode, 842, 1134, 1140
 - w+**, writing and reading file mode, 1140
 - \W**, “not word character,” **regex**, 837, 1135
 - \w**, “word character,” **regex**, 837, 1135
 - `wait()`, 547–548, 556–557
 - Wait loops, 547–548
 - `wait_for_button()` example, 547–548
 - Waiting for user action, 547–548, 556–557
 - wchar_t**, 1003, 1004
 - Web browser, as user interface, 540–541
 - Wheeler, David, 108, 785, 930
 - while**-statements, 108–109
 - White-box testing, 952–953
 - Whitespace
 - formatting, 393, 394–401

- Whitespace (*continued*)
 - identifying, 393
 - in input, 64
 - string**, 818–819
 - ws**, manipulator, 1130
 - Widget** example, 548–549
 - Button**, 418–420, 541–550
 - control inversion, 556–557
 - debugging, 563–564
 - hide()**, 549
 - implementation, 1163–1164
 - In_box()**, 550–551
 - line drawing example, 552–556
 - Menu**, 551, 557–562
 - move()**, 549
 - Out_box()**, 550–551
 - put_on_top()**, 1165
 - show()**, 549
 - technical example, 1167–1170
 - text input/output, 550–551
 - visibility, 549
 - Wild cards, regular expressions, 1133
 - Wilkes, Maurice, 785
 - Window** example, 416, 439
 - canvas, 416
 - creating, 418–420, 542–544
 - disappearing, 563
 - drawing area, 416
 - implementation, 1164–1166
 - line drawing example, 552–556
 - put_on_top()**, 1165
 - with “Next” button, 418–420
 - Window.h** example, 417–418
 - Wirth, Niklaus, 794–795
 - Word frequency, example, 745
 - Words (of memory), 1176
 - write()**, unformatted output, 1129
 - Writing files, 344
 - See also* File I/O.
 - appending to, 385
 - binary I/O, 387
 - example, 346–348
 - fstream** type, 344–346
 - ofstream** type, 345–346
 - ostream** type, 343–348, 387
 - ws** manipulator, 1130
- ## X
- xdigit**, 842, 1134
 - \xhhh**, hexadecimal character literal, 1043
 - xor**, synonym for **^**, 1003, 1004
 - xor_eq**, synonym for **^=**, 1003, 1004
- ## Z
- zero-terminated array, 1011. *See also* C-style string.
 - ZIP code example, 844–849