## Contents

Preface ix

0 Notes to the Reader 1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 The structure of this book</td>
<td>2</td>
</tr>
<tr>
<td>0.2 A philosophy of teaching and learning</td>
<td>5</td>
</tr>
<tr>
<td>0.3 ISO standard C++</td>
<td>8</td>
</tr>
<tr>
<td>0.4 PPP support</td>
<td>11</td>
</tr>
<tr>
<td>0.5 Author biography</td>
<td>13</td>
</tr>
<tr>
<td>0.6 Bibliography</td>
<td>13</td>
</tr>
</tbody>
</table>

Part I: The Basics

1 Hello, World! 17

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Programs</td>
<td>18</td>
</tr>
<tr>
<td>1.2 The classic first program</td>
<td>18</td>
</tr>
<tr>
<td>1.3 Compilation</td>
<td>21</td>
</tr>
<tr>
<td>1.4 Linking</td>
<td>23</td>
</tr>
<tr>
<td>1.5 Programming environments</td>
<td>24</td>
</tr>
</tbody>
</table>
6 Completing a Program

6.1 Introduction ................................................................. 152
6.2 Input and output ............................................................ 152
6.3 Error handling ............................................................... 154
6.4 Negative numbers ......................................................... 156
6.5 Remainder: \% ............................................................... 157
6.6 Cleaning up the code ...................................................... 158
6.7 Recovering from errors ................................................... 164
6.8 Variables ................................................................. 167

7 Technicalities: Functions, etc.

7.1 Technicalities ............................................................... 180
7.2 Declarations and definitions .......................................... 181
7.3 Scope ........................................................................... 186
7.4 Function call and return ............................................... 190
7.5 Order of evaluation ...................................................... 206
7.6 Namespaces ............................................................... 209
7.7 Modules and headers .................................................... 211

8 Technicalities: Classes, etc.

8.1 User-defined types ....................................................... 222
8.2 Classes and members .................................................... 223
8.3 Interface and implementation ....................................... 223
8.4 Evolving a class: Date ................................................... 225
8.5 Enumerations ............................................................. 233
8.6 Operator overloading .................................................... 236
8.7 Class interfaces .......................................................... 237

Part II: Input and Output

9 Input and Output Streams

9.1 Input and output ............................................................ 252
9.2 The I/O stream model .................................................... 253
9.3 Files ........................................................................... 254
9.4 I/O error handling ....................................................... 258
9.5 Reading a single value .................................................. 261
9.6 User-defined output operators ....................................... 266
9.7 User-defined input operators ......................................... 266
9.8 A standard input loop .................................................. 267
13.5 Approximation ................................................................. 392
13.6 Graphing data ................................................................. 397

14 Graphical User Interfaces ...................................................... 409
14.1 User-interface alternatives ............................................... 410
14.2 The “Next” button ............................................................. 411
14.3 A simple window ............................................................... 412
14.4 Button and other Widgets ................................................. 414
14.5 An example: drawing lines ............................................... 419
14.6 Simple animation ............................................................... 426
14.7 Debugging GUI code .......................................................... 427

Part III: Data and Algorithms .................................................. 435

15 Vector and Free Store .......................................................... 435
15.1 Introduction ................................................................. 436
15.2 vector basics ................................................................. 437
15.3 Memory, addresses, and pointers ...................................... 439
15.4 Free store and pointers .................................................... 442
15.5 Destructors ................................................................. 447
15.6 Access to elements .......................................................... 451
15.7 An example: lists ............................................................. 452
15.8 The this pointer .............................................................. 456

16 Arrays, Pointers, and References ........................................ 463
16.1 Arrays ................................................................. 464
16.2 Pointers and references .................................................... 468
16.3 C-style strings ............................................................... 471
16.4 Alternatives to pointer use ............................................... 472
16.5 An example: palindromes .................................................. 475

17 Essential Operations .......................................................... 483
17.1 Introduction ................................................................. 484
17.2 Access to elements .......................................................... 484
17.3 List initialization ............................................................. 486
17.4 Copying and moving .......................................................... 488
17.5 Essential operations .......................................................... 495
Contents

17.6 Other useful operations ............................................................... 500
17.7 Remaining Vector problems ...................................................... 502
17.8 Changing size .............................................................................. 504
17.9 Our Vector so far .......................................................................... 509

18 Templates and Exceptions 513
18.1 Templates .................................................................................... 514
18.2 Generalizing Vector ..................................................................... 522
18.3 Range checking and exceptions .................................................. 525
18.4 Resources and exceptions ........................................................... 529
18.5 Resource-management pointers .................................................. 537

19 Containers and Iterators 545
19.1 Storing and processing data ........................................................ 546
19.2 Sequences and iterators ............................................................... 552
19.3 Linked lists .................................................................................. 555
19.4 Generalizing Vector yet again .................................................... 560
19.5 An example: a simple text editor ................................................ 566
19.6 vector, list, and string ................................................................... 572

20 Maps and Sets 577
20.1 Associative containers ................................................................. 578
20.2 map .............................................................................................. 578
20.3 unordered_map ............................................................................ 585
20.4 Timing ......................................................................................... 586
20.5 set ................................................................................................ 589
20.6 Container overview ................................................................. 591
20.7 Ranges and iterators ................................................................. 597

21 Algorithms 603
21.1 Standard-library algorithms ....................................................... 604
21.2 Function objects ................................................................. 610
21.3 Numerical algorithms ............................................................... 614
21.4 Copying ....................................................................................... 619
21.5 Sorting and searching ............................................................... 620

Index 625
 Damn the torpedoes!
 Full speed ahead.
 – Admiral Farragut

Preface

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. It can also be used by someone with some programming knowledge who wants a more thorough grounding in programming principles and contemporary C++.

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.

There are many kinds of programming. This book aims to serve those who want to write non-trivial 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, and provide exercises for it. If you just want to understand toy programs or write programs that just call code provided by others, you can get along with far less than I present. In such cases, you will
probably also be better served by a language that’s simpler than C++. 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.

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 become a good programmer 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!

There is more to programming – much more – than following a few rules and reading the manual. This book is not focused on “the syntax of C++.” C++ is used to illustrate fundamental concepts. Understanding the fundamental ideals, principles, and techniques is the essence of a good programmer. Also, “the fundamentals” are what last: they will still be essential long after today’s programming languages and tools have evolved or been replaced.

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

Previous Editions
The third edition of Programming: Principles and Practice Using C++ is about half the size of the second edition. Students having to carry the book will appreciate the lighter weight. The reason for the reduced size is simply that more information about C++ and its standard library is available on the Web. The essence of the book that is generally used in a course in programming is in this third edition (“PPP3”), updated to C++20 plus a bit of C++23. The fourth part of the previous edition (“PPP2”) was designed to provide extra information for students to look up when needed and is available on the Web:

- Chapter 1: Computers, People, and Programming
- Chapter 11: Customizing Input and Output
- Chapter 22: Ideas and History
- Chapter 23 Text Manipulation
- Chapter 24: Numerics
- Chapter 25: Embedded Systems Programming
- Chapter 26: Testing
- Chapter 27: The C Programming Language
- Glossary

Where I felt it useful to reference these chapters, the references look like this: PPP2.Ch22 or PPP2 §27.1.
Acknowledgments

Special thanks to the people who reviewed drafts of this book and suggested many improvements: Clovis L. Tondo, Jose Daniel Garcia Sanchez, J.C. van Winkel, and Ville Voutilainen. Also, Ville Voutilainen did the non-trivial mapping of the GUI/Graphics interface library to Qt, making it portable to an amazing range of systems.

Also, thanks to the many people who contributed to the first and second editions of this book. Many of their comments are reflected in this third edition.
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 11 focusing on their implementation and Chapter 12 on design issues.

§10.1 Why graphics?
§10.2 A display model
§10.3 A first example
§10.4 Using a GUI library
§10.5 Coordinates
§10.6 Shapes
§10.7 Using Shape primitives
  Graphics headers and main; Axis; Graphing a function; Polygons; Rectangles; Fill;
  Text; Images; And much more
§10.8 Getting the first example to run
  Source files; Putting it all together
10.1 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 13 gives simple (but general) facilities for graphing data. Also consider browsers, games, animation, scientific visualization, phones, and control displays.
- *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 11 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 PPP2. §22.2.4), but it was soon applied to that, and graphics provide some of the most accessible and tangible 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.

10.2 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
Section 10.2  A display model  291

positions, etc. in a one-dimensional stream of characters. That’s what layout (typesetting, “markup”) languages such as Troff, TeX, Word, Markup, HTML, and XML (and their associated graphical packages) do. For example:

```html
<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 (`<ul>` ... `</ul>`) with list items (`<li>` ... `</li>`), 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 sitting behind the screen,” then takes the objects we have attached to our window and draw them on the screen:

![Diagram](attachment:diagram.png)

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.
10.3 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:

```c++
#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
    Application app; // start a Graphics/GUI application
    Point tl (900,500); // 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:

In the background of our window, we see a laptop screen (cleaned up for the occasion). For people who are curious about irrelevant details, we can tell you that my background is a famous painting
by the Danish painter Peder Severin Krøyer. The ladies are Anna Ancher and Marie Krøyer, both well-known painters. If you look carefully, you’ll notice that we have the Microsoft C++ compiler running, but we could just as well have used some other compiler (such as GCC or Clang). Let’s go through the program line by line to see what was done.

First we #include our graphics interface library:

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

Why don’t we use a module Graph_lib (§7.7.1)? One reason is at the time of writing not all implementations are up to using modules for this relatively complex task. For example, the system we use to implement our graphics library, Qt, exports its facilities using header files (§7.7.2). Another reason is that there is so much C++ code “out there” using header files (§7.7.2) that we need to show a realistic example somewhere.

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

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

Then we start our display engine (§10.2):

```cpp
Application app;               // start a Graphics/GUI application
```

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

```cpp
Point tl {900,500};            // to become top left corner of window
```

Next, we create a window on the screen:

```cpp
Simple_window win {tl, 600, 400, "Canvas"}; // make a simple window
```

We use a class called `Simple_window` to represent a window in our `Graph_lib` interface library. 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.

Next, we put an object in the window:

```cpp
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:

```cpp
poly.set_color(Color::red);    // adjust properties of poly
```
Finally, we attach `poly` to our window, `win`:

```cpp
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:

```cpp
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 in the top right corner 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.

For the rest of the Graphics-and-GUI chapters, we eliminate the distractions around our window and just show the window itself:

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 14, 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”).

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 X
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).

10.4 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 Qt from www.qt.io. Our code is portable wherever Qt is available (Windows, Mac, Linux, many embedded systems, phones, browsers, 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 Qt documentation is thousands of pages. You can download Qt from www.qt.io, 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 Chapter 10 – Chapter 14 can be used with any popular GUI toolkit. We will of course explain how our interface classes map to Qt 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 14).

### 10.5 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:

$$(0,0) \quad (200,0)$$

$$(50,50)$$

$$(0,100) \quad (200,100)$$

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 and varies a lot (e.g., 600-by-1024, 1280-by-1024, 1920-by-1080, 2412-by-1080, and 2880-by-1920).

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 like a screen. Basically, we see a window as a small screen. For example, when we said

```cpp
Simple_window win tl,600,400,"Canvas";
```

we requested a rectangular area 600 pixels wide and 400 pixels high that we can address as 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.
10.6 Shapes

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

- Window
- Simple_window
- Line_style
- Color
- Shape
- Point
- Polygon
- Lines
- Rectangle
- Text
- Image

An arrow indicates that the class pointing 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, Circle, Ellipse, etc.
- Color, Line_style, Point
- Axis

Later (Chapter 14), 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.

10.7 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.

10.7.1 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:

```cpp
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
    // with 10 "notches"
    // label the axis "x axis"
win.attach(xa); // attach xa to the window, win
win.set_label("X axis"); // re-label 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:

![x axis](image)

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. Natu-
rally, we chose to place the x axis somewhere near the bottom of the window. In real life, we’d rep-
resent 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 (§3.3.1,
§13.6.3).

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

Now, let’s add a y axis:

```cpp
Axis ya {Axis::y, Point(20, 300), 280, 10, "y axis");
    ya.set_color(Color::cyan); // choose a color for the y axis
    ya.label.set_color(Color::dark_red); // choose a color for the text
```
Section 10.7.1

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 often use color with more enthusiasm than taste.

10.7.2 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:

```cpp
double dsin(double d) { return sin(d); } // chose the right sin() (§13.3)

Function sine {dsin,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("Sine");
win.wait_for_button();
```

Here, the `Function` named `sine` will draw a sine curve using the standard-library function `sin(double)` to generate values. We explain details about how to graph functions in §13.3. 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):

![Graph of a function](image)

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.

### 10.7.3 Polygons

A graphed function is an example of data presentation. We’ll see much more of that in Chapter 11. 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:

```cpp
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);
```
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 §10.3, 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 (§11.5). We get

10.7.4 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("Rectangle");
win.wait_for_button();
From that, we get

![Image of a polygon]

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:

```cpp
Closed_polyline poly_rect;
poly_rect.add(Point{100,50});
poly_rect.add(Point{200,50});
poly_rect.add(Point{200,100});
poly_rect.add(Point{100,100});

win.set_label("Polyline");
win.attach(poly_rect);
win.wait_for_button();
```

That polygon looks exactly – to the last pixel – like a rectangle:
However, it only looks like a Rectangle. No Rectangle has four points:

```cpp
poly_rect.add(Point(50,75));
win.set_label("Polyline 2");
win.wait_for_button();
```

No rectangle has five points:
In fact, the *image* on the screen of the 4-point `poly_rect` *is* a rectangle. However, the `poly_rect` object in memory is not a `Rectangle` and it does not "know" anything about rectangles.

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 guarantees 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.

### 10.7.5 Fill

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

```cpp
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));
poly_rect.set_fill_color(Color::green);
win.set_label("Fill");
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) and filled it with green:

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 (§11.7, §11.7.2). Rectangles are just special in how easy (and fast) they are to fill.
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:

```cpp
Text t(Point(150,150), "Hello, graphical world!");
win.attach(t);
win.set_label("Text");
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` (§10.7.1) is simply a `Text` object. In addition, we can choose a font and set the size of the characters:

```cpp
t.set_font(Font::times_bold);
t.set_font_size(20);
win.set_label("Bold text");
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:
10.7.7 Images

We can also load images from files:

This was done by:
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:

```cpp
copter.move(100, 250);
win.set_label("Move");
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.

### 10.7.8 And much more

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

```cpp
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'};
m.set_color(Color::red);
```
```cpp
ostringstream oss;
oss << "screen size: " << x_max() << "x" << y_max() << "; window size: " << win.x_max() << "x" << win.y_max();
Text sizes {Point(100,20),oss.str()};

Image scan( Point(275,225),"scandinavia.jfif" );
scan.scale(150,200);

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

win.attach(sizes);
win.attach(scan);
win.set_label("Final!");
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 an `ostringstream` (§9.11) to format the text object displaying sizes. The string composed in `oss` is referred to as `oss.str()`.
10.8 Getting the first example 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 GUI system we use is installed and correctly linked to ours. Previous editions of the PPP code used the FLTK library; the current version uses the more modern Qt library. Both work over a wide range of systems.

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 Qt library
• The C++ standard library

Indirectly, we also use the operating system.

10.8.1 Source files

Our graphics and GUI interface library consists of just five header files:
• Headers meant for users (aka “user-facing headers”):
  • Point.h
  • Window.h
  • Simple_window.h
  • Graph.h
  • GUI.h
  • To implement the facilities offered by those headers, a few more files are used. Implementation headers:
  • Qt headers
  • GUI_private.h
  • Image_private.h
  • Colormap.h
  • Code files:
  • Window.cpp
  • Graph.cpp
  • GUI.cpp
  • GUI_private.cpp
  • Image_private.cpp
  • Colormap.cpp
  • Qt code

We can represent the user-facing headers like this:
An arrow represents a `#include`. Until Chapter 14 you can ignore the GUI header.

A code file implementing a user-facing header `#includes` that header plus any headers needed for its code. For example, we can represent `Window.cpp` like this:

```
int main() { ... }
```

In this way, we use files to separate what a user sees (the user-facing headers, such as `Window.h`) and what the implementation of such headers uses (e.g., Qt headers and `GUI_private.h`). In modules, that distinction is controlled by `export` specifiers (§7.7.1).

This “mess of files” is tiny compared to industrial systems, where many thousands of files are common, not uncommonly tens of thousands of files. That’s one reason we prefer modules; they help organize code. Fortunately, we don’t have to think about more than a few files at a time to get work done. This is what we have done here: the many files of the operating system, the C++ standard library, and Qt are invisible to us as users of our graphics interface library.
10.8.2 Putting it all together

Different systems (such as Windows, Mac, and Linux) have different ways of installing a library (such as Qt) and compiling and linking a program (such as ours). Worse, such set-up procedures change over time. Therefore, we place the instructions on the Web: www.stroustrup.com/programming.html and try to keep those descriptions up to date. When setting up your first project, be careful and be prepared for possible frustration. Setting up a relatively complex system like this can be very simple, but there are usually “things” that are not obvious to a novice. If you are part of a course, your teacher or teaching assistant can help, and might even have found an easier way to get you started. In any case, installing a new system or library is exactly where a more experienced person can be of significant help.

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 Qt library, #include Graph.h and Simple_window.h in your code, and compile and link Graph.cpp and Window.cpp into your program.

[2] Now add the examples from §10.7 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?
[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)?
You made a Window object, but nothing appears on your screen. What are some possible reasons for that?

What library do we use to implement our graphics/GUI interface library? Why don’t we use the operating system directly?

<table>
<thead>
<tr>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
</tr>
<tr>
<td>GUI</td>
</tr>
<tr>
<td>library</td>
</tr>
<tr>
<td>color</td>
</tr>
<tr>
<td>image</td>
</tr>
</tbody>
</table>

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 3-by-3 tic-tac-toe board of alternating white and red squares.
5. Draw a red 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 source file diagram from §10.8.1.
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. Hint: The trigonometric functions are found in <cmath> and module std (PPP2.§24.8).
12. A superellipse is a two-dimensional shape defined by the equation ${\left| \frac{x}{a} \right|}^m + {\left| \frac{y}{b} \right|}^n = 1$; where $m > 0$ and $n > 0$.
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 to which to connect a point another argument or just use $N-1$, i.e., all the other points).

[13] Find a way to add color to the lines 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/GUI is just an illustration 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.
This page intentionally left blank
Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it. – Samuel Johnson

**Token**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>not equal</td>
</tr>
<tr>
<td>&amp;</td>
<td>address of</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>reference to</td>
</tr>
<tr>
<td>()</td>
<td>move</td>
</tr>
<tr>
<td>*</td>
<td>contents of</td>
</tr>
<tr>
<td>*=</td>
<td>dereference</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
</tr>
<tr>
<td>:</td>
<td>application, operator</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>//</td>
<td>comment</td>
</tr>
<tr>
<td>::</td>
<td>member</td>
</tr>
<tr>
<td>/*</td>
<td>comment</td>
</tr>
<tr>
<td>::</td>
<td>namespace</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>output operator</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>==</td>
<td>and</td>
</tr>
<tr>
<td>+=</td>
<td>addition</td>
</tr>
<tr>
<td>-&gt;</td>
<td>auto</td>
</tr>
<tr>
<td></td>
<td>dereference</td>
</tr>
<tr>
<td></td>
<td>member access</td>
</tr>
<tr>
<td>-</td>
<td>member</td>
</tr>
<tr>
<td>*</td>
<td>dereference</td>
</tr>
<tr>
<td>/</td>
<td>member access</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than operator</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>==</td>
<td>assignment operator</td>
</tr>
<tr>
<td>delete</td>
<td>delete</td>
</tr>
<tr>
<td>initializer</td>
<td>initializer</td>
</tr>
<tr>
<td>Vector</td>
<td>Vector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Token</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>500</td>
</tr>
<tr>
<td>&amp;</td>
<td>439</td>
</tr>
<tr>
<td>address of</td>
<td>194, 196</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>494</td>
</tr>
<tr>
<td>move</td>
<td>494</td>
</tr>
<tr>
<td>rvalue reference</td>
<td>494</td>
</tr>
<tr>
<td>()</td>
<td>application, operator 610</td>
</tr>
<tr>
<td>call, operator</td>
<td>610</td>
</tr>
<tr>
<td>initializer</td>
<td>46</td>
</tr>
<tr>
<td>vector initializer</td>
<td>72</td>
</tr>
<tr>
<td>*</td>
<td>contents of 439</td>
</tr>
<tr>
<td>dereference</td>
<td>440, 444</td>
</tr>
<tr>
<td>iterator</td>
<td>553</td>
</tr>
<tr>
<td>*=</td>
<td>scaling 40</td>
</tr>
<tr>
<td>+</td>
<td>addition 34</td>
</tr>
<tr>
<td>concatenation</td>
<td>36</td>
</tr>
<tr>
<td>++</td>
<td>increment 39, 58</td>
</tr>
<tr>
<td>iterator</td>
<td>553</td>
</tr>
<tr>
<td>auto</td>
<td>205</td>
</tr>
<tr>
<td>dereference</td>
<td>444</td>
</tr>
<tr>
<td>member access</td>
<td>444</td>
</tr>
<tr>
<td>member</td>
<td>73, 223</td>
</tr>
<tr>
<td>member access</td>
<td>444</td>
</tr>
<tr>
<td>comment</td>
<td>163</td>
</tr>
<tr>
<td>comment</td>
<td>19</td>
</tr>
<tr>
<td>::</td>
<td>144</td>
</tr>
<tr>
<td>member</td>
<td>229</td>
</tr>
<tr>
<td>namespace</td>
<td>209</td>
</tr>
<tr>
<td>;</td>
<td>semicolon 59</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>output operator 19</td>
</tr>
<tr>
<td>user-defined</td>
<td>266</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than 500</td>
</tr>
<tr>
<td>less-than operator</td>
<td>58</td>
</tr>
<tr>
<td>order</td>
<td>604</td>
</tr>
<tr>
<td>=</td>
<td>494</td>
</tr>
<tr>
<td>==</td>
<td>35</td>
</tr>
<tr>
<td>assignment</td>
<td>36</td>
</tr>
<tr>
<td>assignment operator</td>
<td>57, 490</td>
</tr>
<tr>
<td>delete</td>
<td>375</td>
</tr>
<tr>
<td>initializer</td>
<td>32, 46</td>
</tr>
<tr>
<td>Vector</td>
<td>Vector</td>
</tr>
</tbody>
</table>
B

\texttt{b, format()} 282
\texttt{bad()} 258
\texttt{balanced tree} 580
\texttt{base class} 318, 367
\texttt{basic guarantee} 533
\texttt{begin()} 594
\texttt{end()} 500
\texttt{benefits of OOP} 376
\texttt{bibliography} 13
\texttt{big-O} 585, 621
\texttt{binary tree} 580
\texttt{binary\_operation, concept} 520
\texttt{binary\_search()} 621
\texttt{binding, structured} 580
Bjarne Stroustrup 10, 13
\texttt{block, \{\}} 66
\texttt{body, function} 69
\texttt{bool} 32
\texttt{bottom-up, top-down} 128
\texttt{box, dialog} 417
Brian Kernighan 17
\texttt{browser I/O} 410
\texttt{buffer} 143
\texttt{I/O} 253
\texttt{overflow} 465
\texttt{builder, GUI} 431
\texttt{built-in array} 594
\texttt{type} 222
\texttt{Button} 415
\texttt{byte} 439

C

\texttt{C++} 10
\texttt{C++ and} 10
\texttt{with classes} 10
\texttt{C++ and C} 10
\texttt{and Simula} 10
\texttt{compiler} 12
\texttt{Core Guidelines} 10, 12
\texttt{design} 11
\texttt{evolution} 11
\texttt{Foundation} 12
\texttt{history} 10
\texttt{ISO} 8
\texttt{stability} 11
\texttt{C++11} 10
\texttt{C++14} 11
\texttt{C++17} 11
\texttt{C++20} 11
\texttt{C++98} 10
\texttt{calculator example} 119
\texttt{call}
\texttt{cost of virtual implementation, function} 200
\texttt{operator()} 610
\texttt{recursive function} 203
\texttt{stack, function} 203
callee error handling 91
caller error handling 90
capacity(), \texttt{Vector} 506
capture, lambda 613
case 62
cat() 471
catch
exception 95
\texttt{try} 530
category, iterator 597
\texttt{CC} 2
\texttt{CG} 12
\texttt{char} 32
character literal 32
\texttt{Checked\_iterator} 599
classification, I/O 252
cleaning code 158
clearing code 158
file, object 21
index 1
file, source 21
generalize 547
pseudo 119
ugly 56, 190
Color 323
invisible 339
RGB 325
color comment 102, 162
// 19
/* 163
comparison operator 500
compatibility 527
compilation 21
compiler C++ 12
evaluator 12
compile-time
computation 204
error 24, 84, 86
completing a program 152
computation 52
compile-time 204
concatenation, + 36
concept, predicate 620
concept 518
binary_operation 520
convertible_to 520
copyable 520
derived_from 520
equality_comparable 519
equality_comparable_with 519
floating_point 520
forward_iterator 519
indirect_unary_predicate 520
input_iterator 519
integral 520
invocable 616, 618
invocable 520
moveable 520
output_iterator 519
predicate 519
random_access_iterator 519
random_access_range 519
range 519
regular 520
semiregular 520
sortable 520
totally_ordered 520
totally_ordered_with 520
console I/O 410
const 57
declaration 184
member function 242
reference, pass-by 194
constant
expression 56
magic 159
symbolic 159
constexpr 205
constexpr 56, 204
constraint on solution 527
constructor
class 227
copy 239, 489
default 240, 496
explicit 497
move 494
container
currency 553
almost 593
and inheritance 520
associative 578
list 591
map 578
multimap 591
multiset 591
overview 591
set 589
STL 592
unordered_map 585
vector 591
contents of, * 439
contract 104
core 414
access 362
inversion 411, 422
conversion 44
function argument 199
narrowing 45
to enum 234
widening 45
copyable, concept 520
copy() 488
assignment 490
constructor 239, 489
deep 492
default 488
elision 494
I/O example 594
shallow 492
copy() 487, 619
copyable, concept 520
copy() 620
Core Guidelines, C++ 10, 12
correctness 53
corruption, memory 440
cost of virtual call 370
cost of virtual, memory 370
Courtney 449
cout output stream 19
cpp 21
cppreference 12
C-style string 471

data 546
graphing 397
date 589
Date example 266, 270
David Wheeler 65, 545
deallocation
delete 446
delete[] 446
debugging 101, 498
GUI 427
declaration 42, 181
argument 190
const 184
function 71
return type 190
using 210
variable 184
deep copy 492
default
argument 387
constructor 240, 496
copy 488
destructor 448
initialization 34, 185
member initializer 242
default 62
default_random_engine 108
definition 42, 182
function 69
in-class 144
member function 229
operator 236, 501
delete
= 375
deallocation 446
naked 450
new and 446
delete[] deallocation 446
dereference
-> 444
= 440, 444
nullptr 469
derived 366
class 367, 370
derived_from_concept 520
design 117
C++ 11
strategy 117
destroy() 523
destroy_at() 564
destructor 447–448
default 448
generated 448
name 448
pointer 496
resource 496
virtual 449, 496
development strategy 117
device
input 252
output 252
dialog box 417
directive, using 210
dispatch 367
display model 290
distribution, random number 108
divide-and-conquer 54
domain, application 356
Donald Knuth 606
double 32
int 89
doubly-linked list 556, 591
Doug McIlroy 577
Dow Jones example 583
draw() 291
draw_all[] example 518
Drill 3, 24
duration 587
duration_cast 588
dynamic memory 442
editor example 566
efficiency 53, 527
Einstein, Albert 151
elision, copy 494
Ellipse 344
Circle and 345
else 60
empty
statement 59
string 34
string 72
encapsulation private 368
end() 594
begin() 500
fail 604
ingine, random number 108, 588
entity 184
enum
class 233
conversion to 234
enumeration 233

E
plain 235
scoped 233
underlying type 234
enumeration, enum 233
enumerator 233
environment, programming 24
eof() 258
equal, == 34, 500, 604
equality operator, == 57
equality_comparable, concept 519
equality_comparable_with, concept 519
equal_range() 622
erase()
  list 562
  Vector 564
  vector 562
error
  argument 94
  avoiding 99
  compile-time 24, 84, 86
  finding 99
  handling 154
  handling alternatives, I/O 261
  handling, callee 91
  handling, caller 90
  input 97
  I/O 258
  link-time 24, 84, 88
  logic 24, 84, 89
  range 95, 465
  reporting 93
  run-time 24, 84, 89
  sources of 85
  syntax 84, 86
  throw on I/O 260
  transient 465
  type 84, 87
error() 90
essential operation 495
estimation 100
Euler, Leonhard 1
evaluation
  order of 206–207
  short-circuit 207
evolution, C++ 11
example
  age group 397
  calculator 119
  copy I/O 594
  Date 266, 270
  Dow Jones 583
  draw_all() 518
  editor 566
  exponentiation 392
  Expression 128
  Fruit 589
get10() 261
get_int() 264
gods 452
grid 321
grow() 503
int_to_month() 234
Jack-and-Jill 546, 554
Larger_than 610
Lines_window 419
Link 452
Menu 418
No_case 620
Output_range 598
palindrome() 76, 475
Random 588
Reading 257
read_one_value 261
Record 612
skip_to_int() 263
suspicious() 530
TC++PL 579
temperature 74, 256
Text_iterator 569
to_int() 234
Token 121
Token_stream 142
traffic-light 426
Vector 437, 451, 502, 514, 522, 534, 560, 564
word counting 578
exception 525
  catch 95
  exception 98
  out_of_range 96
  resource and 529
  runtime_error 98
  throw 94
exception exception 98
executable file 23
Exercise 3
expect() 105
explicit constructor 497
explorer, compiler 12
exponential_distribution 108
exponentiation example 392
export 211
expression 55
  constant 56
  lambda 106, 389, 613
Expression example 128
extern 182

F
fail
  algorithm 604
  end() 604
fail() 258
fall through 192
[[fallthrough]] attribute 64
false 32
feature creep 127, 135
Feynman, Richard 251
file 254
executable 23
header 25, 213
object code 21
read 256, 269
source code 21
stream, fstream 255
stream, ifstream 255
stream, ofstream 255
write 256
fill 304
finally() 542
find() 605
find_if() 608
finding error 99
first, pair 580
five, rule of 496
floating-point literal 32
floating_point, concept 520
Font 347
for
range 73, 562
statement 67
formal argument 190
format, output 281
format()
argument () 281
:b 282
:d 282
:o 282
:x 282
forward_iterator, concept 519
forward_list 592
Foundation, C++ 12
framework, test 108
free store 442
free() 472
Fruit example 589
fstream file stream 255
function 68
activation record 201
argument checking 199
argument conversion 199
body 69
call implementation 200
call, recursive 203
call stack 203
class member 226
cost member 242
declaration 71
definition 69
definition, member 229
drawing, member 229
graphing 382
hash 585
local 613
member 73
modifying 243
object 610–611
parameterized 517
pure virtual 374
purpose of 69
table, virtual 369
template 517
utility 265
virtual 367, 370
Function 299, 386

G
Gavin 571
generalize code 547
generated destructor 448
generator
random number 109
type 516
generic programming 517–518
Gerald Weinberg 51
get10() example 261
get_int() example 264
gif 351
global
initialization of 208
scope 186
gods example 452
good() 258
grammar 127
notation 129
granularity 357
graphical layout 400
graphics 290, 356
interface class 316
model 295
graphing
data 397
function 382
Graph_lib namespace 292
grid example 321
grow, vector 73
grow1() example 503
guarantee
basic 533
no-throw 533
resource 533
strong 533
GUI
builder 431
debugging 427
interface class 316
I/O 410
starting with 311
Guidelines, C++ Core 10, 12
gui_main() 367
Application 421

H
handling, error 154
hash
  function 585
table 578
header
  file 25, 213
  PPP.h 11
  PPPHeaders.h 11
heap 442
Hein, Piet 221
Hello, World! 18
hiding, implementation 453
hierarchy, class 368
high-level programming 7
history, C++ 10

I
ideal 380
ideals, STL 549
if statement 60
ifstream file stream 255
Image 306, 350
implementation 117
class 223
  function call 200
  hiding 453
  inheritance 376
implicit
  release 530
  release of resource 448
import 20, 211
ln_box 416
in-class
  definition 144
  initializer 242
#include 25, 213
increment, ++ 39, 58
indentation 190
indirect_unary_predicate, concept 520
inheritance 367
  container and 520
  implementation 376
  interface 376
initialization  
  assignment and 38
  default 34, 185
  of global 208
  with list 486
initializer
  {} 36, 46
  () 46
  =() 46
  = 32, 46
  () vector 72
default member 242
in-class 242
new 445
order, member 420, 487
initializer_list 486
inline 231
in-memory representation 42, 270
inner_product() 617
input 52
  device 252
  error 97
  operator, >> 31, 33
  stream, cin 31
insert()
  list 562
  Vector 565
  vector 562
installation instructions 311
installing Qt 311
instantiation, template 516
instructions, installation 311
int 32
to double 89
integer literal 32
integral, concept 520
interface
  class 223, 237
class, graphics 316
class, GUI 316
  inheritance 376
  minimal 357
int_to_month() example 234
invariant 229, 335
inversion, control 411, 422
invisible, Color 339
invocable
  concept 616, 618
  concept 520
I/O
  browser 410
  buffer 253
  classification 252
  console 410
  error 258
  error handling alternatives 261
error, throw on 260
GUI 410
stream 253
iota() 614
is kind of 318, 367
ISO
C++ 8
standard 245
istream 253
width() 477
istream_iterator 594
istringstream 283
iterate 65, 558
iteration statement 65
iterator
++ 553
*= 553
== 553
category 597
range 597
sequence and 552

J
Jack-and-Jill example 546, 554
Johnson, Samuel 627
jpg 306, 351

K
Kernighan, Brian 17
keyword 41
Knuth, Donald 606
Kristen Nygaard 115

L
lambda 412
[] 412
capture 613
expression 106, 389, 613
Larger_than example 610
layout
graphical 400
object 368
lcd() 614
lcm() 614
leak
memory 447
resource 530, 539
Leonhard Euler 1
less than, < 500
less-than operator, < 58
library 118
standard 20
lifting 555
Line 318
Lines 320
Line_style 325
Lines_window example 419
Link example 452
linked list 452
linking 23
link-time error 24, 84, 88
list
doubly-linked 556, 591
initialization with 486
linked 452
operation 453
 singly-linked 556, 591
List 555
list
container 591
erase() 562
insert() 562
string, vector 572
literal
character 32
floating-point 32
integer 32
string 19, 32
local
function 613
scope 186
static variable 208
logic error 24, 84, 89
look-ahead 121
loop
class 66–67
wait 413
Louis Pasteur 29
lowercase 620
low-level programming 7
lvalue 55, 58

M
magic constant 159
main() 20, 192
make_shared() 539
make_unique() 450, 538
malloc() 472
management, resource 530
map 580
[] 578
container 578
Mark 348
Marked polyline 330
Marks 332
Maurice Wilkes 83
McIlroy, Doug 577
measuring time 586
member
  73, 223
  :: 229
access, :: 444
access, -> 444
class 123, 223
function 73
function, class 226
function, const 242
function definition 229
initializer, default 242
initializer order 420, 487
memory 439
  corruption 440
  cost of virtual 370
  dynamic 442
  leak 447
  raw 524
Menu example 418
midpoint() 614
minimal interface 357
model 356
graphics 295
modifying function 243
module 23, 211
PPP_graphics 292, 317
PPP_support 11, 527
scope 186
std 20
move 493
  && 494
  assignment 494
  constructor 494
  return 537
moveable, concept 520
multimap container 591
multiset container 591
mutability 359

N
\n 19
naked
  delete 450
  new 450
  new 539
name 40
  ~ destructor 448
  argument 191
namespace 209
  :: 209
  Graph_lib 292
  scope 186
  std 19
naming style 358

narrow() 200
narrowing conversion 45
Negroponte, Nicholas 409
nested scope 188
new 442
  allocation 443
  and delete 446
  initializer 445
  naked 450
  naked 539
Nicholas 31
  Negroponte 409
  no op 365
No_case example 620
[nodiscard] attribute 616
Norah 55
  not equal, != 500
notation
  grammar 129
  shorthand 519
  no-throw guarantee 533
  not_null 474
  now() 586
  null
    pointer 468
    reference 468
  nulptr 446
    dereference 469
    test 469
  numerical algorithm 614
  Nygaard, Kristen 115

O
\in, format() 282
object 30, 42
  code file 21
  function 610–611
  layout 368
  object-oriented programming 368, 518
  of course 126
ofstream file stream 255
OOP 368
  benefits of 376
Open_polyline 328
operation
  essential 495
  list 453
  style 357
operator 34, 57
  () application 610
    = assignment 57, 490
    () call 610
    == equality 57
    >> input 31, 33
    < less-than 58
<< output 19
[] subscript 484
comparison 500
definition 236, 501
overloading 236, 501
relational 500
optional checking 527
order << 604
member initializer 420, 487
of evaluation 206–207
ostream 253
ostream_iterator 594
stringstream 283
out of range 465
Out_box 417
out_of_range exception 96
output 52
device 252
operator, << 19
range 598
stream, cout 19
output_iterator, concept 519
Output_range example 598
overflow 396
buffer 465
overloading, operator 236, 501
override 366, 370–371
overview, container 591

P
Painter, Qt 329
pair 582
first 580
second 580
palindrome example 76, 475
parallel algorithm 604
parameter 69, 190
type template 514
parameterized class 517
function 517
parametric polymorphism 517
parser 128, 130
recursive descent 140
partial_sum() 614
pass-by const reference 194
reference 196
value 193
value or const-reference 197
Pasteur, Louis 29
perfection 239
Perlis, Alan 483
philosophy, teaching 5
Piet Hein 221
Point 317
pointer 439
alternatives to 472
and reference 468–469
argument 470
arithmetic 466
destructor 496
null 468
problem 443, 464–465
semantics 492
this 456
use 537
Polygon 300
and Closed Polyline 334
polyline 328
polymorphism ad hoc 517
parametric 517
run-time 367
portability 9
postcondition 106
Postscript 4
PPP support 11
PPP_graphics module 292, 317
PPP_headers.h, header 11
PPP_support, module 11, 527
precondition 104
predicate 612
predicate concept 519
caption 620
preprocessor 215
printf() 281
private 223, 362
class 142
encapsulation 368
problem, pointer 443, 464–465
problems, startup 12
program 18
completing a 152
structure 146
programming environment 24
generic 517–518
high-level 7
low-level 7
object-oriented 368, 518
promotion 44
protected 368
prototype 119
pseudo code 119
public 223
class 142
pure
virtual, =0 374
virtual function 374
purpose of function 69
push_back() vector 507
vector 73

Q
Qt 12, 295
installing 311
Painter 329

R
RAII 532
random
number 108
number distribution 108
number engine 108, 588
number generator 109
number seed(), Random example 588
random_access_iterator, concept 519
random_access_range, concept 519
random_int() 109, 588
range
checking 525
type declaration 190
return move 537
type, auto 205
type, suffix 205
return
move 537
type declaration 190
value 192
Review 4, 26
RGB Color 325
Richard Feynman 251
round_to() 200
rule
of all 496
of five 496
of zero 496
run-time error 24, 84, 89
polymorphism 367
runtime_error exception 98
rvalue 55
reference, && 494

S
safety, type 43
Samuel Johnson 627
scaling 401
*a= 40
scope 186
class 186
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>357</td>
</tr>
<tr>
<td>subclass</td>
<td>367</td>
</tr>
<tr>
<td>subscript</td>
<td>444, 464, 466</td>
</tr>
<tr>
<td>operator</td>
<td>484</td>
</tr>
<tr>
<td>suffix return type</td>
<td>205</td>
</tr>
<tr>
<td>superclass</td>
<td>367</td>
</tr>
<tr>
<td>support, PPP</td>
<td>11</td>
</tr>
<tr>
<td>Sure!</td>
<td>135</td>
</tr>
<tr>
<td>suspicious() example</td>
<td>530</td>
</tr>
<tr>
<td>switch statement</td>
<td>62</td>
</tr>
<tr>
<td>symbolic constant</td>
<td>159</td>
</tr>
<tr>
<td>syntax error</td>
<td>84, 86</td>
</tr>
<tr>
<td>sys_days</td>
<td>589</td>
</tr>
<tr>
<td>system_clock</td>
<td>586</td>
</tr>
<tr>
<td>table</td>
<td></td>
</tr>
<tr>
<td>hash</td>
<td>578</td>
</tr>
<tr>
<td>virtual function</td>
<td>369</td>
</tr>
<tr>
<td>TC++PL example</td>
<td>579</td>
</tr>
<tr>
<td>teaching philosophy</td>
<td>5</td>
</tr>
<tr>
<td>technicalities</td>
<td>180</td>
</tr>
<tr>
<td>temperature example</td>
<td>74, 256</td>
</tr>
<tr>
<td>template</td>
<td>514</td>
</tr>
<tr>
<td>argument, value</td>
<td>521</td>
</tr>
<tr>
<td>class</td>
<td>517</td>
</tr>
<tr>
<td>function</td>
<td>517</td>
</tr>
<tr>
<td>instantiation</td>
<td>516</td>
</tr>
<tr>
<td>parameter, type</td>
<td>514</td>
</tr>
<tr>
<td>Terms</td>
<td>4</td>
</tr>
<tr>
<td>test</td>
<td>107, 117, 162</td>
</tr>
<tr>
<td>framework</td>
<td>108</td>
</tr>
<tr>
<td>nullptr</td>
<td>469</td>
</tr>
<tr>
<td>testing</td>
<td>154</td>
</tr>
<tr>
<td>Text</td>
<td>305, 346</td>
</tr>
<tr>
<td>Text_iterator example</td>
<td>569</td>
</tr>
<tr>
<td>thinking</td>
<td>116</td>
</tr>
<tr>
<td>this pointer</td>
<td>456</td>
</tr>
<tr>
<td>throw</td>
<td></td>
</tr>
<tr>
<td>exception</td>
<td>94</td>
</tr>
<tr>
<td>on I/O error</td>
<td>260</td>
</tr>
<tr>
<td>time, measuring</td>
<td>586</td>
</tr>
<tr>
<td>time_point</td>
<td>587</td>
</tr>
<tr>
<td>timer_wait()</td>
<td>426</td>
</tr>
<tr>
<td>to_intr() example</td>
<td>234</td>
</tr>
<tr>
<td>Token example</td>
<td>121</td>
</tr>
<tr>
<td>Token_stream example</td>
<td>142</td>
</tr>
<tr>
<td>top-down bottom-up</td>
<td>128</td>
</tr>
<tr>
<td>totally_ordered, concept</td>
<td>520</td>
</tr>
<tr>
<td>totally_ordered_with, concept</td>
<td>520</td>
</tr>
<tr>
<td>traffic-light example</td>
<td>426</td>
</tr>
<tr>
<td>transient error</td>
<td>465</td>
</tr>
<tr>
<td>translation unit</td>
<td>23</td>
</tr>
<tr>
<td>transparency</td>
<td>325</td>
</tr>
<tr>
<td>traverse, vector</td>
<td>72</td>
</tr>
<tr>
<td>tree</td>
<td></td>
</tr>
<tr>
<td>balanced</td>
<td>580</td>
</tr>
<tr>
<td>binary</td>
<td>580</td>
</tr>
<tr>
<td>red-black</td>
<td>578</td>
</tr>
<tr>
<td>search</td>
<td>580</td>
</tr>
<tr>
<td>true</td>
<td>32</td>
</tr>
<tr>
<td>truncate</td>
<td>45</td>
</tr>
<tr>
<td>Try this</td>
<td>4</td>
</tr>
<tr>
<td>try catch</td>
<td>530</td>
</tr>
<tr>
<td>Trygve Reenskaug</td>
<td>603</td>
</tr>
<tr>
<td>type</td>
<td>30, 42</td>
</tr>
<tr>
<td>auto return</td>
<td>205</td>
</tr>
<tr>
<td>auto variable</td>
<td>46</td>
</tr>
<tr>
<td>built-in</td>
<td>222</td>
</tr>
<tr>
<td>enum underlying error</td>
<td>84, 87</td>
</tr>
<tr>
<td>generator</td>
<td>516</td>
</tr>
<tr>
<td>safety</td>
<td>43</td>
</tr>
<tr>
<td>suffix return</td>
<td>205</td>
</tr>
<tr>
<td>template parameter</td>
<td>514</td>
</tr>
<tr>
<td>user-defined</td>
<td>222</td>
</tr>
<tr>
<td>ugly code</td>
<td>56, 190</td>
</tr>
<tr>
<td>underlying type, enum</td>
<td>234</td>
</tr>
<tr>
<td>uniform_int_distribution</td>
<td>108</td>
</tr>
<tr>
<td>uninitialized variable</td>
<td>44</td>
</tr>
<tr>
<td>uninitialized_fill()</td>
<td>524</td>
</tr>
<tr>
<td>uninitialized_move()</td>
<td>523</td>
</tr>
<tr>
<td>unique_copy()</td>
<td>619</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>450, 538</td>
</tr>
<tr>
<td>Vector</td>
<td>539</td>
</tr>
<tr>
<td>unit</td>
<td>588</td>
</tr>
<tr>
<td>translation</td>
<td>23</td>
</tr>
<tr>
<td>unordered_map container</td>
<td>585</td>
</tr>
<tr>
<td>use</td>
<td></td>
</tr>
<tr>
<td>case</td>
<td>119</td>
</tr>
<tr>
<td>pointer</td>
<td>537</td>
</tr>
<tr>
<td>user-defined</td>
<td>267</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>266</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>266</td>
</tr>
<tr>
<td>type</td>
<td>222</td>
</tr>
<tr>
<td>using</td>
<td></td>
</tr>
<tr>
<td>declaration</td>
<td>210</td>
</tr>
<tr>
<td>directive</td>
<td>210</td>
</tr>
<tr>
<td>utility function</td>
<td>265</td>
</tr>
<tr>
<td>valarray</td>
<td>593</td>
</tr>
<tr>
<td>valid state</td>
<td>229</td>
</tr>
<tr>
<td>value</td>
<td>42</td>
</tr>
<tr>
<td>or const-reference, pass-by</td>
<td>197</td>
</tr>
</tbody>
</table>
pass-by 193
return 192
semantics 492
template argument 521
variable 30, 32, 42
declaration 184
local static 208
loop 66–67
type, auto 46
uninitialized 44
vector algorithm 604
Vector
assignment = 507
capacity() 506
erase() 564
example 437, 451, 502, 514, 522, 534, 560, 564
insert() 565
push_back() 507
representation 505
reserve() 506
resize() 506
unique_ptr 539
vector 436
container erase() 591
grow 73
initializer, () 72
insert() 562
list string insert() 572
push_back() 73
sequence 71
size() 72
traverse 72
vector<int>
vector<string>
virtual 365
=0 pure 374
call, cost of 370
destructor 449, 496
function 367, 370
function, pure 374
function table 369
memory cost of 370
Vitruvius 355
void 191
Voltaire 381
W
wait loop 413
wait_for_button() 359, 367, 411, 413
Web resources 12
weekday() 589
Weinberg, Gerald 51
Wheeler, David 65, 545
while statement 65
whitespace 33
widening conversion 45
Widget 414
and Window 415
width(), istream 477
Wilkes, Maurice 83
Window, Widget and Window 415
Winston Churchill 513
word counting example 578
write file 256
X
x, format() 282
XX 2
Y
year_month_date 245
Z
zero, rule of 496