PROGRAMMING

Principles and Practice Using C++

THIRD EDITION

BJARNE STROUSTRUP

THE CREATOR OF C++

Programming:
Principles and Practice Using C++
Third Edition

Bjarne Stroustrup

Addison-Wesley
Hoboken, New Jersey

Cover photo by Photowood Inc./Corbis.
Author photo courtesy of Bjarne Stroustrup.
Page 294: ‘‘Promenade a Skagen’’ by Peder Severin Kroyer.
Page 308: Photo of NASA’s Ingenuity Mars Helicopter, The National Aeronautics and Space Administration (NASA).
Page 354: Photo of Hurricane Rita as seen from space, The National Oceanic and Atmospheric Administration (NOAA).

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024932369

This book was typeset in Times and Helvetica by the author.

ISBN-13: 978-0-13-830868-1
ISBN-10: 0-13-83086-3
First printing, May 2024
$PrintCode

Contents

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. 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 nontrivial 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 page intentionally left blank

10

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**, **Polygon**s, **Axis**, and **Text** are examples of **Shape**s. 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?

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

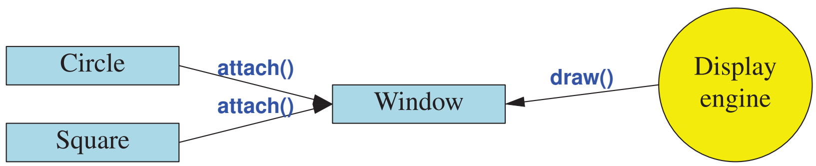
The iostream library is oriented toward reading and writing streams of characters as they might appear in a list of numeric values or a book. The only direct supports for the notion of graphical position are the newline and tab characters. You can embed notions of color and two-dimensional positions, 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:

```
<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> ...) with list items (<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: CC



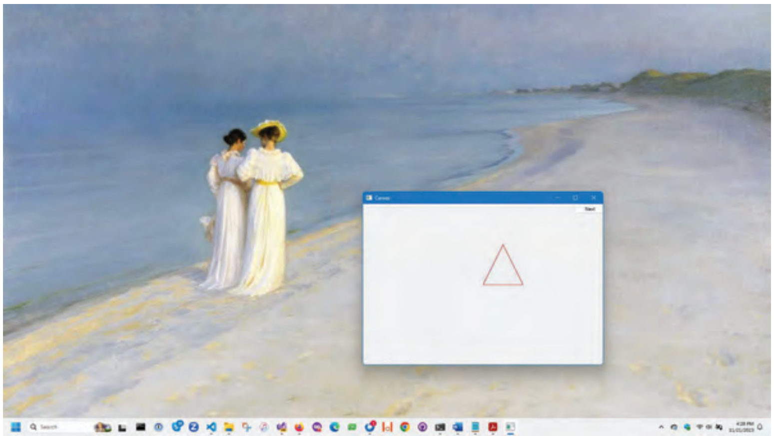
The "display engine" draws lines on the screen, places strings of text on the screen, colors areas of the screen, etc. For simplicity, we'll use the phrase "our GUI library" or even "the system" for the display engine even though our GUI library does much more than just drawing the objects. In the same 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:

```
#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:

```
#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:

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

Then we start our display engine (§10.2):

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

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

```
Point tl {900,500};                           // 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 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:

```
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**:

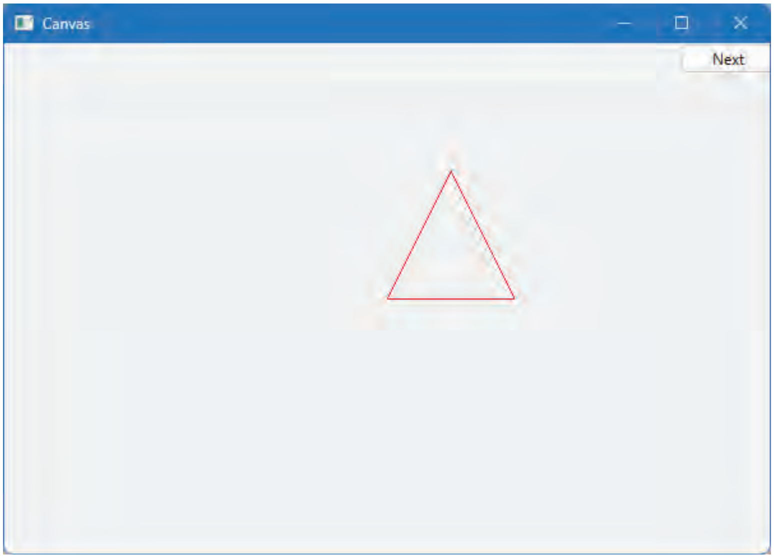
```
win.attach(poly);                    // connect poly to the window
```

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 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**).

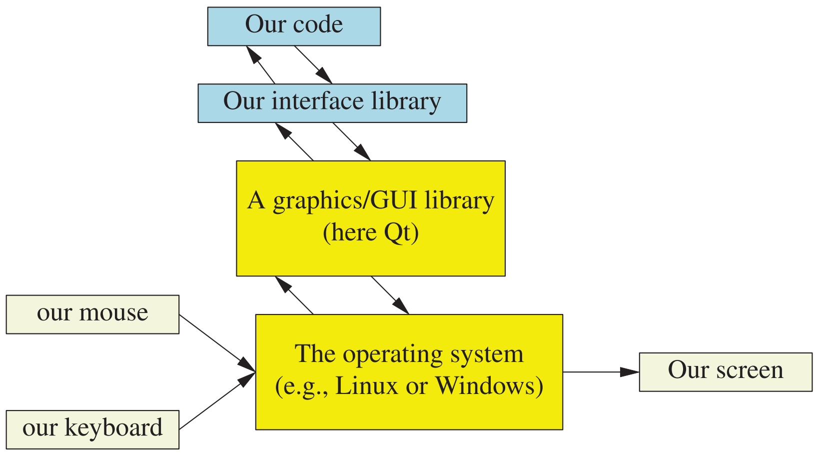
10.4 Using a GUI library

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. CC

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.

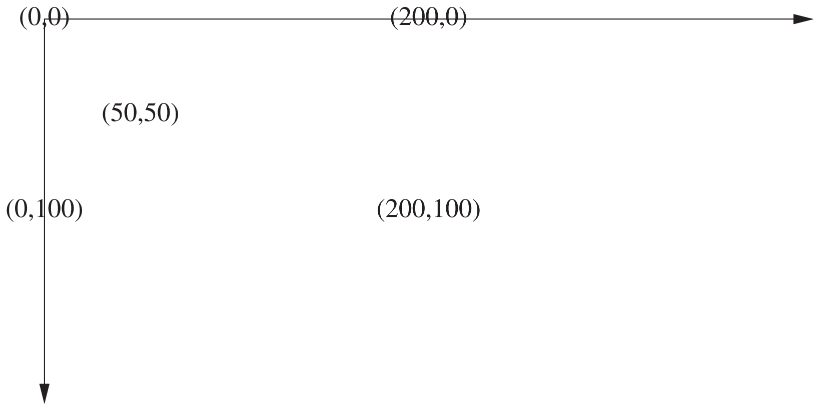
We can illustrate the parts of our "graphics world" like this: CC



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

CC 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:



XX 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).

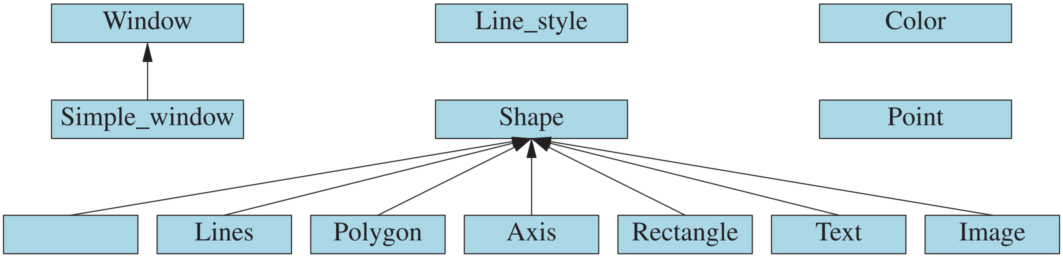
In the context of interacting with a computer using a screen, a window is a rectangular region of the screen devoted to some specific purpose and controlled by a program. A window is addressed exactly like a screen. Basically, we see a window as a small screen. For example, when we said

```
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:



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.

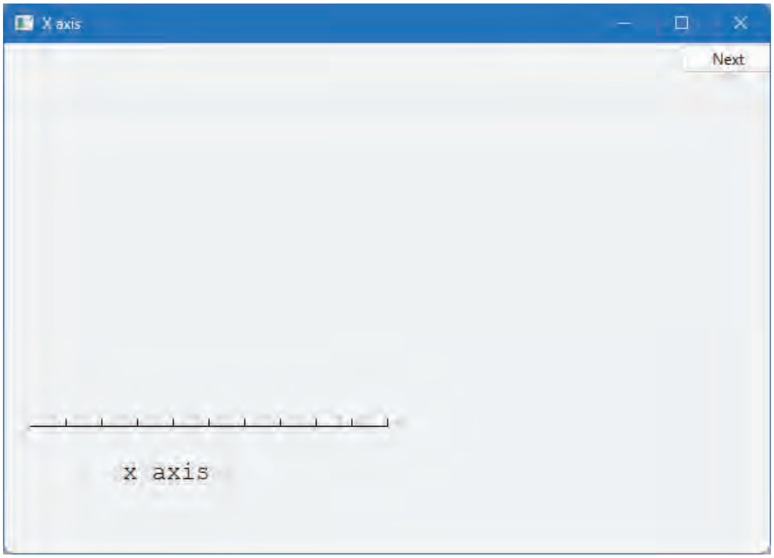
We will now walk through a simple program, explaining the code line by line and showing the effect of each on the screen. When you run the program, you'll see how the image changes as we add shapes to the window and modify existing shapes. Basically, we are "animating" the progress through the code by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

```
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:



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** (§3.3.1, §13.6.3).

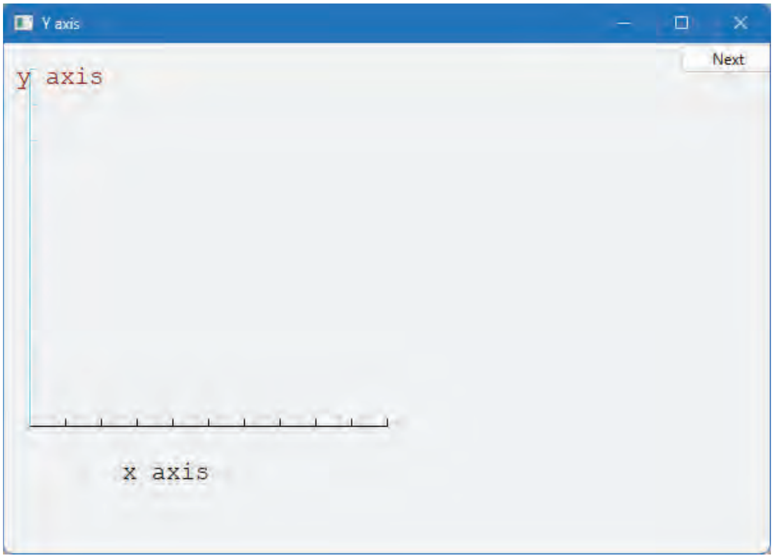
To help identify our output we relabeled the screen to **X axis** using **Window**'s member function **set_label()**.

Now, let's add a *y* axis:

```
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
```

```
win.attach(ya);
win.set_label("Y axis");
win.wait_for_button();                    // display!
```

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



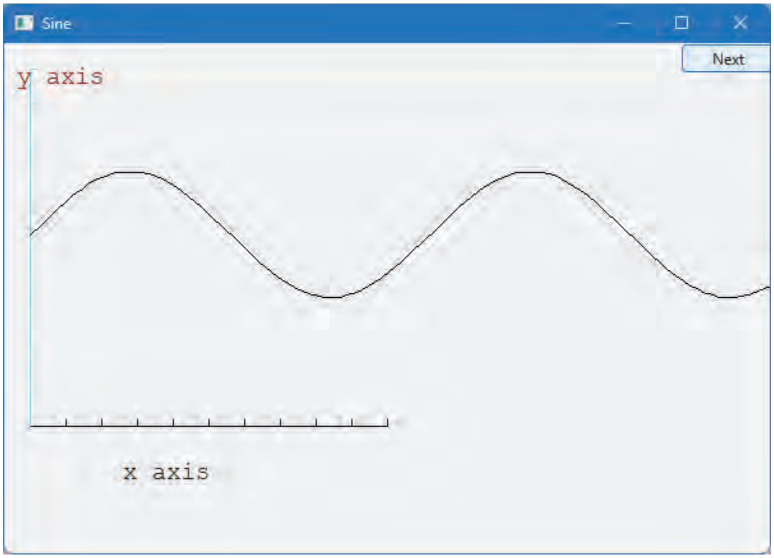
We don't actually think that it is a good idea to use different colors for *x* and *y* axes. We just wanted to show you how you can set the color of a shape and of individual elements of a shape. Using lots of color is not necessarily a good idea. In particular, novices often use color with more enthusiasm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

```
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):



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:

```
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);
```

```
win.attach(poly);
win.set_label("Triangle");
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 §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. CC

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:

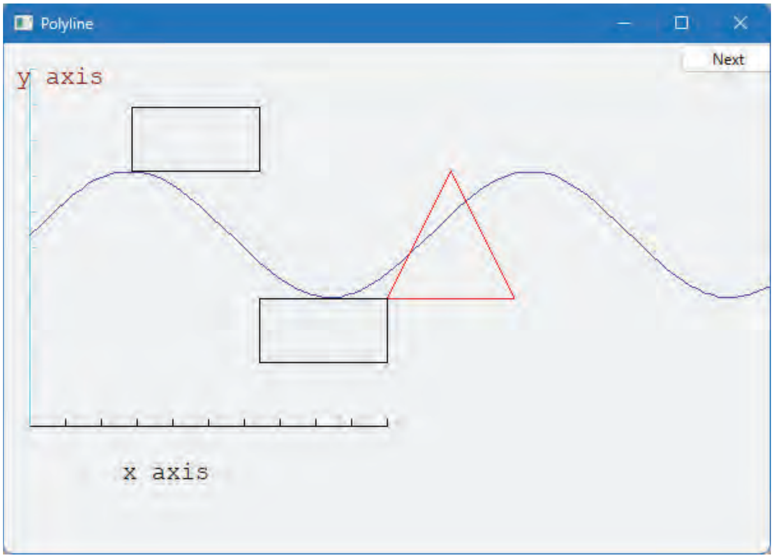
```
Rectangle r {Point{200,200}, 100, 50};          // top left corner, width, height

win.attach(r);
win.set_label("Rectangle");
win.wait_for_button();
```

From that, we get

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{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();
```

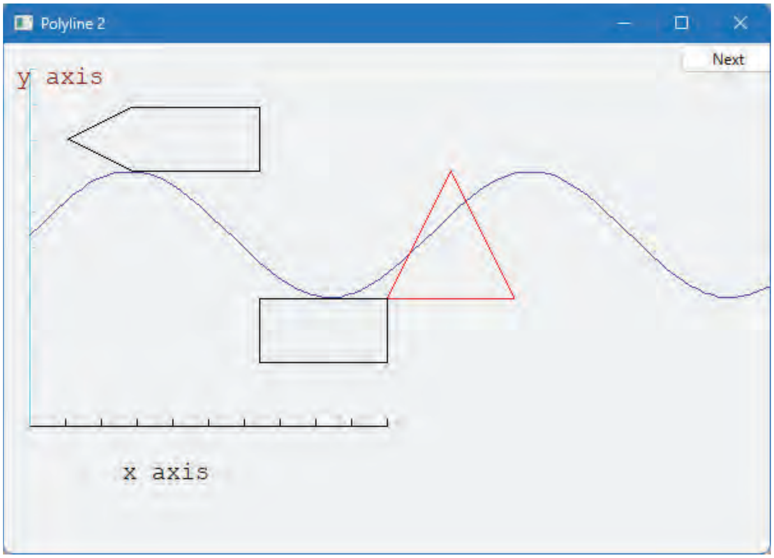
That polygon looks exactly – to the last pixel – like a rectangle:



However, it only looks like a **Rectangle**. No **Rectangle** has four points:

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

No rectangle has five points:



CC 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.

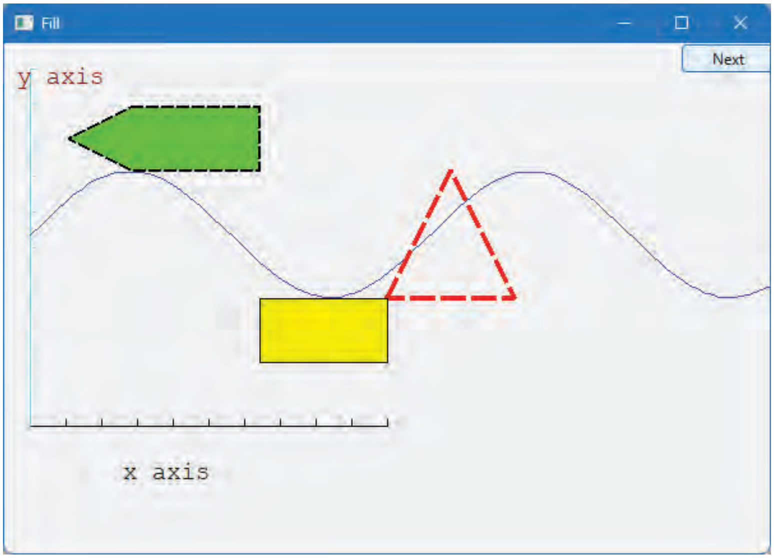
It is important for our reasoning about our code that a **Rectangle** doesn't just happen to look like a rectangle on the screen; it maintains the fundamental guarantees of a rectangle (as we know them from geometry). We write code that depends on a **Rectangle** really being a rectangle on the screen and staying that way.

10.7.5 Fill

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

```
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:



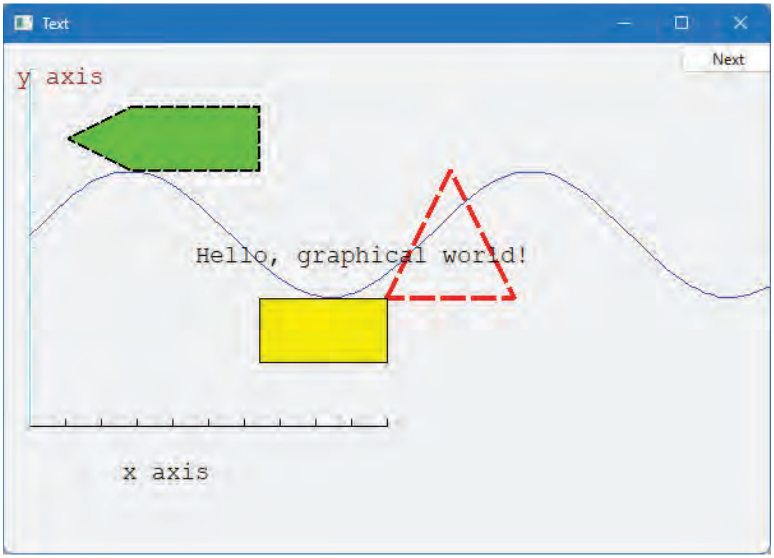
If you look carefully at **poly_rect**, you'll see that the outline is printed on top of the fill.

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.

10.7.6 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: CC

```
Text t {Point{150,150}, "Hello, graphical world!"};
win.attach(t);
win.set_label("Text");
win.wait_for_button();
```

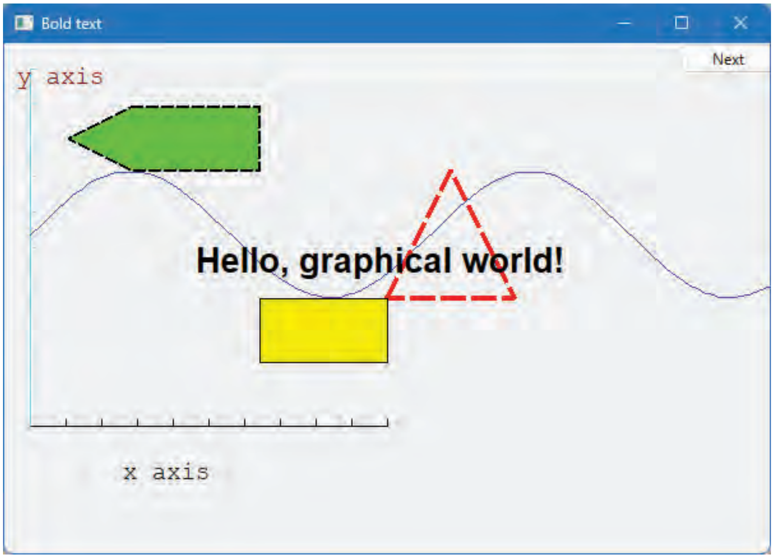


From the primitive graphics elements you see in this window, you can build displays of just about any complexity and subtlety. For now, just note a peculiarity of the code in this chapter: there are no loops, no selection statements, and all data was "hardwired" in. The output was just composed of primitives 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:

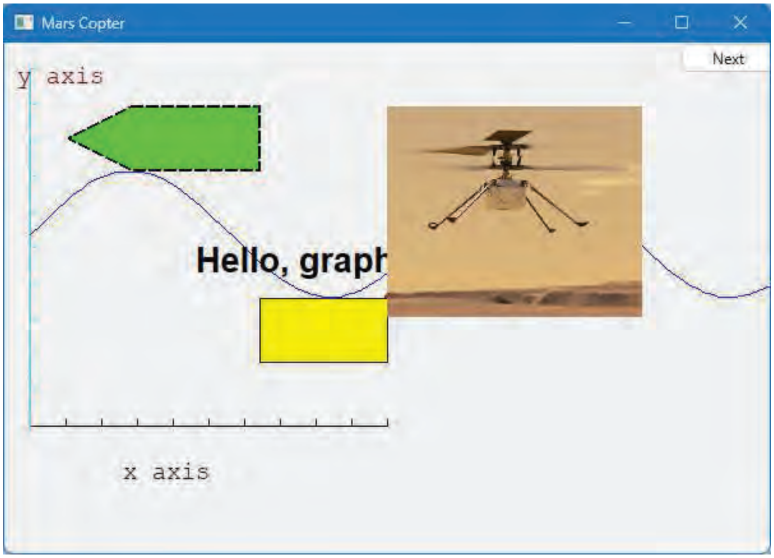
```
t.set_font(Font::times_bold);
t.set_font_size(20);
win.set_label("Bold text");
win.wait_for_button();
```

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:

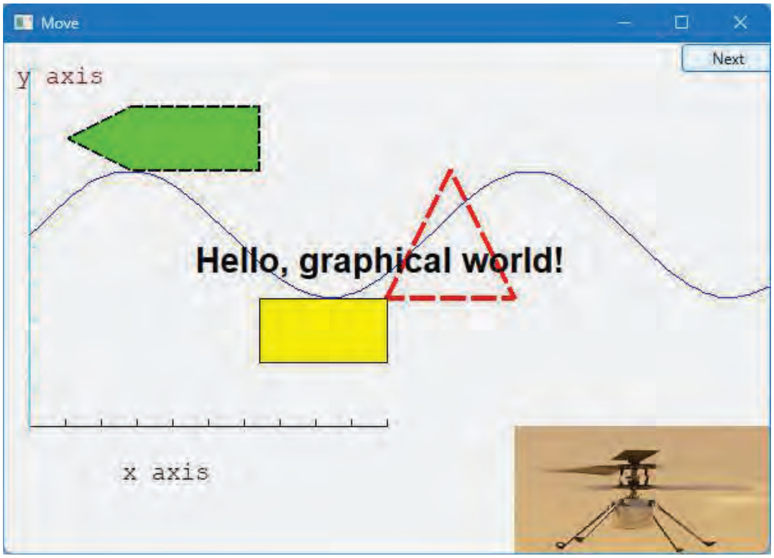


This was done by:

```
Image copter {Point{100,50},"mars_copter.jpg"};
win.attach(copter);
win.set_label("Mars copter");
win.wait_for_button();
```

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:

```
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.

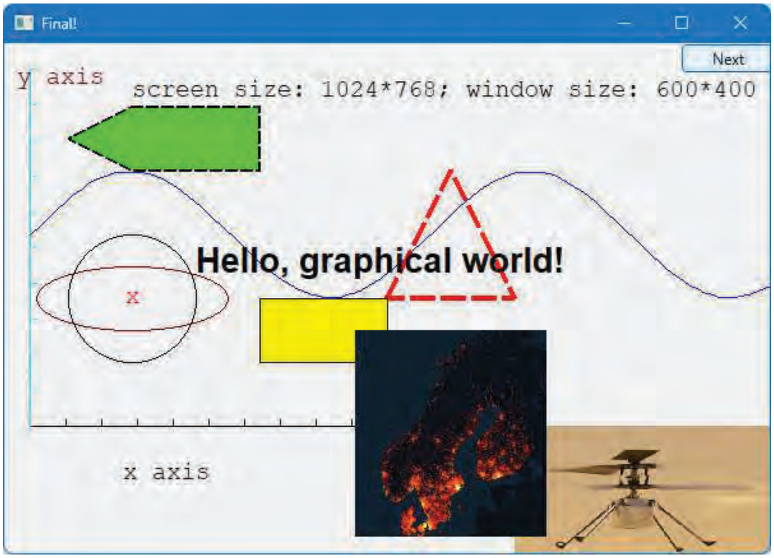
10.7.8 And much more

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

```
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);
```

```
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 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?



AA 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

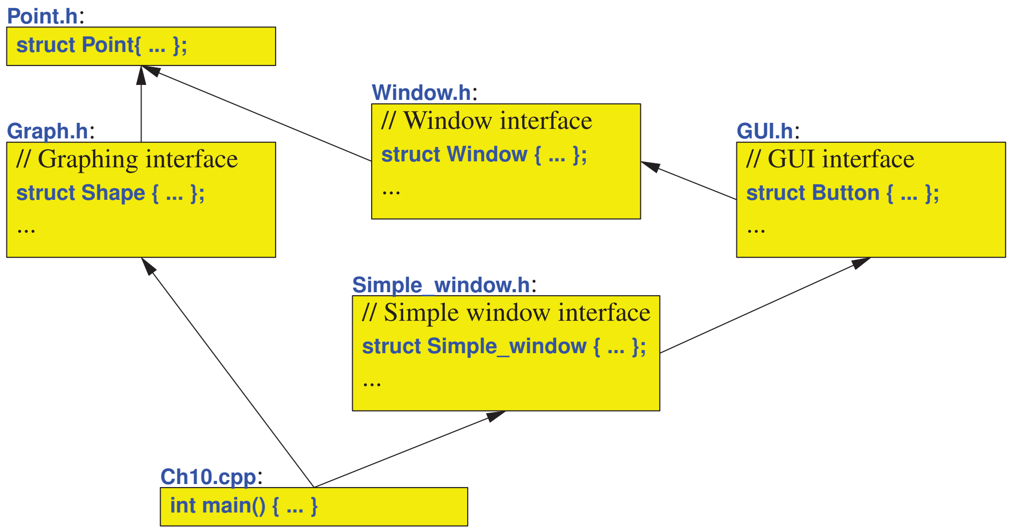
Indirectly, we also use the operating system.

10.8.1 Source files

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

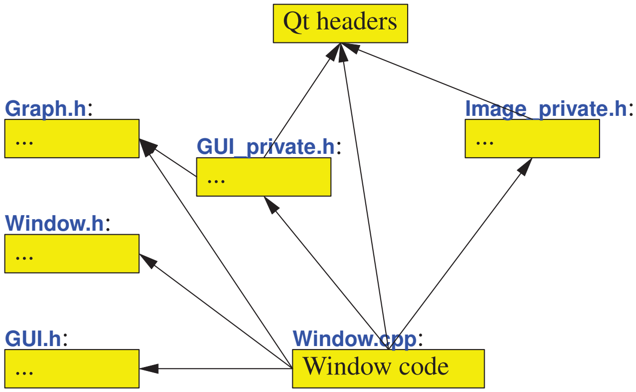
- Headers meant for users (aka "user-facing headers"):
  - **Point.h**
  - **Window.h**
  - **Simple_window.h**
  - **Graph.h**
  - **GUI.h**
- To implement the facilities offered by those headers, a few more files are used. Implementation headers:
  - Qt headers
  - **GUI_private.h**
  - **Image_private.h**
  - **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 **#include**s that header plus any headers needed for its code. For example, we can represent **Window.cpp** like this



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

Terms

color	graphic	JPEG	coordinates
GUI	line style	display	PPP_graphics
library	software layer	fill	Shape
color	HTML	window	Qt
image	XML	Simple_window	

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

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

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. AA

This page intentionally left blank

I

Index

Knowledge is of two kinds.
We know a subject ourselves,
or we know where
we can find information on it.
– Samuel Johnson

Token

B

C

D

E

F

G

H

I

J

K

L

M

P

T

U

V

W

X

Y

Z