Chapter 3
How to Write a Program

In This Chapter

▶ Designing your program

▶ Understanding the technical details

▶ Choosing a programming language

▶ Defining how the program should work

▶ Knowing the life cycle of a typical program

Although you can sit down at your computer and start writing a program right now without any planning whatsoever, the result would likely to prove as messy as trying to bake a cake by throwing all the ingredients together without following a recipe.

You can write a simple program that displays your cat's name on-screen without much planning, but for anything more complex, you want to take time to design your program on paper before you even touch a computer. After you're sure that you know what you want your program to do and how you want it to look on-screen, you can worry about writing a program that actually accomplishes this task.

Before You Write Your Program

If you design your program before writing it, you don't waste time writing a program that doesn't work or that solves the wrong problem and isn't worth trying to salvage afterward. By planning ahead of time, you increase the odds that your program actually works and performs the task that you want.

The following three items are crucial to consider in designing a program:

- **The user:** Who's going to use your program?
- **The target computer:** Which computer do people need to run your program? Is it a Windows 98/Me/NT/2000/XP computer, a Macintosh, a mainframe, a computer running Linux, a handheld Palm or PocketPC, or a supercomputer?

- **You:** Are you going to write the entire thing yourself or get help from others? If you're going to get others to help you, which parts of the program are they going to write?

When you're designing a program, you don't need to worry about which programming language you're going to use. Once you know exactly what you want your program to do, then you can choose which programming language might be easiest to use.

The program's users

If you're the only person who's going to use your program, you can pretty much make your program look and act any way you want, just as long as you know how to make it work. But if you plan to give or sell your program to others, you need to know who's going to use your program.

Knowing your program's typical user is critical. If users don't like your program for any reason, they're unlikely to use it. Whether the program actually works is often irrelevant.

By designing your program with the user in mind, you increase the odds that people use your program and (you hope) buy a copy for themselves.

Even if you write a program that works perfectly, users still may ignore it because they don't like the way it looks, they don't understand how to give it commands, it doesn't work the same way as the old program they currently use, the colors don't look right to them, and so on. The goal is to make your program meet your users' needs, no matter how weird, bizarre, or illogical they may seem. (The needs — not the users.)

The target computer

After you identify the user, you need to know what type of computer the user intends to run the program on. The type of computer that your program runs on can determine which computer languages you can use, the hardware that your program can expect to find, and even the maximum size of your program.

If you're writing a program to run on a Macintosh, for example, your program can take advantage of sound, color graphics, a large hard disk, and plenty of memory. You need to rewrite that same program drastically, however, to run it on a Palm handheld computer, with its limited sound capability, much simpler color graphics, and limited amount of memory and storage space.

Portability and cross-platform issues

Rather than pick a single computer, many programmers try to write programs that can run on a variety of computers, such as the Macintosh and Windows 98/Me/NT/2000/XP. Any program that can run on two or more different types of computers is *cross-platform*. Microsoft Word is a cross-platform program because you can buy separate versions that run in the Macintosh and Windows environments.

A program that can run on multiple computers increases your number of potential customers, but also increases the number of potential problems that you must face. Some of the problems include offering customer support for each version of your program and trying to make each program version work the same although they may run on completely different operating systems and computers with totally different capabilities.

At one time, WordPerfect offered versions of its word processor that ran on MS-DOS, Windows, the Macintosh, the Amiga, and the Atari ST. So besides hiring enough programmers to work on each word-processor version, the makers of WordPerfect also needed to hire technical support people who knew how to answer questions for each computer type. Needless to say, this situation cost the company a bundle every month. Developing and supporting so many different versions of WordPerfect cut into the company's profits, so WordPerfect dropped support for the Amiga, Macintosh, and Atari ST because keeping them wasn't worth the cost.

Currently, two of the most popular cross-platform compilers include Real Basic and Delphi/Kylix. Real Basic lets you write programs that can run on Macintosh, Linux, and Windows with minor modifications, while Delphi and Kylix let you write programs that can run on both Windows and Linux. If you want your program to run on different operating systems, you'll need to use a cross-platform compiler. As another alternative, you can always write your program in C/C++ and then compile it on different operating systems and tweak each version to make it run under specific operating systems.

If you can copy and run your program on another computer with little or no modification, your program is considered *portable*. The computer language that you use to write your program can affect its portability. That's why so many people use C/C++ — C and C++ programs tend to be more portable than other programming languages.

Your own programming skill

When designing any program, consider your own programming skill. You may get a great idea for a program, but if you're a beginner with little experience, writing your program may take a long time — if you don't give up out of frustration first.

Beware of the golden handcuffs

Rather than learn programming themselves, many people hire someone to write programs for them. But take care! Freelance programmers sometimes live by a rule known as the "golden handcuffs," which means that they get the gold and you get the handcuffs.

Here's how the golden handcuffs work: You hire someone to write your program, and the programmer takes your money. Then that person writes a program that doesn't work quite the way that you want. Rather than lose the money you already invested in developing the program, you pay the programmer more money, and then this programmer develops a new version of your program that doesn't quite work either.

At this point, you're handcuffed. Do you keep paying money to a programmer who never completely finishes the job, or do you give up altogether? What's worse, you can't hire a new programmer to work on the same program because the original programmer owns your program's source code, so nobody else can modify it. Thus the only way that you can modify the program is to hire the original programmer again and again and again and. . . .

Your programming skill and experience also determine the programming language that you choose. Experienced programmers may think nothing about writing entire programs in C or C++. But novices may need to spend a long time studying C and C++ before writing their programs, or they may choose an easier programming language, such as BASIC.

Some novices take the time to learn difficult languages, such as C/C++, and then go off and write their program. Others take an easier approach and choose a simpler language such as Visual Basic so they can create (and market) their programs right away. Don't be afraid to tackle a heavy-duty language such as C/C++, but don't be afraid to use a simpler language such as Visual Basic either. The important goal is to finish your program so you can start using it and (possibly) start selling it to others.

Many programmers create their program by using a language such as Visual Basic and then later hire more experienced programmers to rewrite their programs in a more complex language such as C/C++, which can make the program faster and more efficient.

The Technical Details of Writing a Program

Few people create a program overnight. Instead, most programs evolve over time. Because the process of actually typing programming commands can prove so tedious, time-consuming, and error-prone, programmers try to avoid actually writing their programs until they're absolutely sure that they know what they're doing.

Prototyping

To make sure that they don't spend months (or years) writing a program that doesn't work right or that solves the wrong problem, programmers often *prototype* their programs first. Just as architects often build cardboard or plastic models of skyscrapers before a construction crew starts welding I-beams together, programmers create mock-ups (prototypes) of their programs first.

A prototype usually shows the user interface of the program, such as windows, pull-down menus, and dialog boxes. The prototype may look like an actual program, but clicking menus doesn't do anything. The whole idea of the prototype is to show what the program looks like and how it acts, without taking the time to write commands to make the program actually work.

After the programmer is happy with the way the prototype looks, she can proceed, using the prototype as a guideline toward completing the final program.

General purpose versus specialized programming languages

General purpose programming languages, such as C/C++, BASIC, Pascal, assembly language, and so on, give you the ability to create practically anything you want, but it may take a long time to do so. To make programming faster and easier, many people have developed specialized programming languages for solving specific types of problems.

For example, SNOBOL is a little known language specifically designed for manipulating text. If that's what you need, then writing a program in SNOBOL can be much quicker than using C/C++. Of course, if you want to do something else besides text manipulation, programming in SNOBOL will likely be a horrible choice.

Similarly, programmers often use LISP and Prolog to create artificially intelligent programs as both LISP and Prolog include commands for decision-making. While you could create an artificially intelligent program using COBOL or BASIC, you would have to write at least twice as many instructions in either FORTRAN, C/C++, or COBOL to accomplish what a single LISP or Prolog command could accomplish.

So the moral of the story is that you can make programming a lot easier if you just choose the right programming language to help you solve the right problem.

Using multiple programming languages

Instead of writing an entire program using one programming language (such as C++), some compilers can convert source code into a special file known as an *object file*. The purpose of object files is that one programmer can write a program in C++, another in assembly language, and still a third in Pascal. Each programmer writes his portion of the program in his favorite language and stores it in a separate object file. Then the programmers connect (or link) all these object files together to create one big program. The program that converts multiple object files into an executable program is known as a *linker*.

In the world of Microsoft Windows, another way to write a program using multiple languages is to use *dynamic link libraries* (*DLLs*), which are special programs that don't have a user interface. One programmer can use C, another can use Java, and a third can use COBOL to create three separate DLL files. Then a fourth programmer can write a program using another language such as Visual Basic, which creates the user interface and uses the commands that each separate DLL file stores.

A third way to write a program is to use your favorite language (such as Pascal) and then write assembly language instructions directly in parts of your program. (Just be aware that not all compilers enable you to switch between different languages within the same program.)

Finally, Microsoft offers a programming framework dubbed *.NET*. By using the .NET framework, one programmer can program in C#, another can program in FORTRAN, and still another can program in BASIC. Then their different programs can share data and communicate with other programs through the .NET framework and create a single user interface that unifies these separate programs. The whole point to all of these different methods is that by using different programming languages, you can take advantage of each language's strengths, while minimizing its weaknesses.

Many programmers use Visual Basic because it's easy for creating prototypes quickly. After you use Visual Basic to create a prototype that shows how your user interface works, you can start adding actual commands to later turn your prototype into an honest-to-goodness working program.

Choosing a programming language

After you refine your prototype until it shows you exactly how your program is to look and act, the next step is choosing a programming language to use.

You can write any program by using any programming language. The trick is that some languages make writing certain types of programs easier.

The choice of a programming language to use can pit people against one another in much the same way that religion and politics do. Although you can't find a single "perfect" programming language to use for all occasions, you may want to consider a variety of programming languages. Ultimately, no one cares what language you use as long as your program works.

Defining how the program should work

After choosing a specific programming language, don't start typing commands into your computer just yet. Just as programmers create mock-ups (prototypes) of their program's user interface, they often create mock-up instructions that describe exactly how a program works. These mock-up instructions are known as *pseudocode*.

If you need to write a program that guides a nuclear missile to another city to wipe out all signs of life within a 100-mile radius, your pseudocode may look as follows:

```
1. Get the target's coordinates.
2. Get the missile's current coordinates.
3. Calculate a trajectory so the missile hits the target.
4. Detonate the nuclear warhead.
```

By using pseudocode, you can detect flaws in your logic before you start writing your program — places where the logic behind your program gets buried beneath the complexity of a specific programming language's syntax.

In the preceding example, you can see that each pseudocode instruction needs further refining before you can start writing your program. You can't just tell a computer, "Get the target's coordinates" because the computer wants to know, "Exactly how do I get the target's coordinates?" So rewriting the preceding pseudocode may look as follows:

```
1. Get the target's coordinates.
            a. Have a missile technician type the target
           coordinates.
            b. Make sure that the target coordinates are
           valid.
            c. Store the target coordinates in memory.
2. Get the missile's current coordinates.
3. Calculate a trajectory so the missile hits the target.
4. Detonate the nuclear warhead.
```

You can refine the instructions even further to specify how the computer works in more detail, as follows:

```
1. Get the target's coordinates.
            a. Have a missile technician type the target
           coordinates.
            b. Make sure that the target coordinates are
           valid.
                 1) Make sure that the target coordinates are
           complete.
```

(continued)

```
                  2) Check to make sure that the target
              coordinates are within the missile's range.
                  3) Make sure that the target coordinates
              don't accidentally aim the missile at friendly
              territories.
               c. Store the target coordinates in memory.
2. Get the missile's current coordinates.
3. Calculate a trajectory so the missile hits the target.
4. Detonate the nuclear warhead.
```

When programmers define the general tasks that a program needs to accomplish and then refine each step in greater detail, they say that they're doing a *top-down design*. In other words, they start at the top (with the general tasks that the program needs to do) and then work their way down, defining each task in greater detail until the pseudocode describes every possible step that the computer must go through.

Writing pseudocode can prove time-consuming. But the alternative is to start writing a program with no planning whatsoever, which is like hopping in your car and driving north and then wondering why you never seem to wind up in Florida.

Pseudocode is a tool that you can use to outline the structure of your program so that you can see all the possible data that the computer needs to accomplish a given task. The idea is to use English (or whatever language you understand best) to describe the computer's step-by-step actions so that you can use the pseudocode as a map for writing the actual program in whatever language (C/C++, FORTRAN, Pascal, Java, and so on) that you choose.

The Life Cycle of a Typical Program

Few programs are written, released, and left alone. Instead, programs tend to go through various cycles where they get updated continuously until they're no longer useful. (That's why many people buy a new word processor every few years even though the alphabet hasn't changed in centuries.)

Generally, a typical program goes through a development cycle (where you first create and release it), a maintenance cycle (where you eliminate any glaring bugs as quickly as possible), and an upgrade cycle (where you give the program new features to justify selling the same thing all over again).

The development cycle

Every program begins as a blank screen on somebody's computer. During the development cycle, you nurture a program from an idea to an actual working program. The following steps make up the development cycle:

1. **Come up with an idea for a program.**
2. **Decide the probable identity of the typical user of the program.**
3. **Decide which computer the program is to run on.**
4. **Pick one or more computer languages to use.**
5. **Design the program by using pseudocode or any other tool to outline the structure of the program.**
6. **Write the program.**
7. **Test the program.**

   This step is known as *alpha testing.*
8. **Fix any problems that you discover during alpha testing.**

   Repeat Steps 7 and 8 as often as necessary.
9. **Give out copies of the program to other people to test.**

   This step is known as *beta testing.*
10. **Fix any problems that people discover during beta testing.**

    Repeat Steps 9 and 10 as often as necessary.
11. **Release the program to the unsuspecting public and pray that it works as advertised.**

The maintenance cycle

Most programmers prefer to create new programs than maintain and modify existing ones, which can prove as unappealing as cleaning up somebody else's mess in an apartment. But the number of new programs that programmers create every year is far less than the number of existing programs, so at some point in your life, you're likely to maintain and update a program that either you or somebody else wrote.

The following list describes typical steps that you may need to follow to maintain an existing program:

1. **Verify all reports of problems (or *bugs*) and determine what part of the program may be causing the bug to appear.**
2. **Fix the bug.**
3. **Test the program to make sure that the bug is really gone and that any changes you make to the program don't introduce any new bugs.**
4. **Fix any problems that may occur during testing.**

5. **Repeat Steps 1 through 4 for each bug that someone reports in the program.**

   Given the buggy nature of software, these steps may go on continuously for years.

6. **Release a software *patch,* which users can add to an existing version of the program to incorporate corrections that you make to "patch up" the problems.**

The upgrade cycle

Companies don't make money fixing software and making it more stable, reliable, and dependable. Instead, companies make money by selling new versions of their programs that offer additional features and options that most people probably don't use or need in the first place.

Still, because so many programs undergo modification to take advantage of new hardware or software, you may find yourself occasionally upgrading a program by adding new features to it. The following steps make up the upgrade cycle:

1. **Determine what new feature you want to add to the program.**
2. **Plan how this new feature is to work (by using pseudocode or another tool to help structure your ideas).**
3. **Modify the program to add this new feature.**
4. **Test this new feature (by using alpha testing) to make sure that it works and doesn't introduce new bugs into the program.**
5. **Fix any problems that may occur during alpha testing.**
6. **Give out copies of the program to other people to beta test.**
7. **Fix any problems that the beta testers report.**
8. **Repeat Steps 1 through 7 for each new feature that you need to add to the program.**
9. **Release the program as a new version and wait for the public to start reporting bugs that keep the program from working correctly so that you can start the maintenance cycle all over again.**

Despite all the university courses and such important-sounding titles as "software engineer," programming is still less of a science and more of an art. Writing, modifying, and updating software doesn't require a high IQ or an advanced mathematics degree as much as it requires creativity, determination, and plenty of imagination. You can write a program any way that you want, but the best way to prevent possible problems later on is to be organized and methodical in your approach.