6.1 A problem

Writing a program starts with a problem; that is, you have a problem that you'd like a program to help solve. Understanding that problem is key to a good program. After all, a program that solves the wrong problem is likely to be of little use to you, however elegant it may be. There are happy accidents when a program just happens to be useful for something for which it was never intended, but let's not rely on such rare luck. What we want is a program that simply and cleanly solves the problem we decided to solve.

At this stage, what would be a good program to look at? A program that

- Illustrates design and programming techniques
- Gives us a chance to explore the kinds of decisions that a programmer must make and the considerations that go into such decisions
- Doesn't require too many new programming language constructs
- Is complicated enough to require thought about its design
- Allows for many variations in its solution
- Solves an easily understood problem
- Solves a problem that's worth solving
- Has a solution that is small enough to completely present and completely comprehend

We chose "Get the computer to do ordinary arithmetic on expressions we type in"; that is, we want to write a simple calculator. Such programs are clearly useful; every desktop computer comes with such a program, and you can even buy computers specially built to run nothing but such programs: pocket calculators.

For example, if you enter

```
2+3.1*4
```

the program should respond

```
14.4
```

Unfortunately, such a calculator program doesn't give us anything we don't already have available on our computer, but that would be too much to ask from a first program.

6.2 Thinking about the problem

So how do we start? Basically, think a bit about the problem and how to solve it. First think about what the program should do and how you'd like to interact with it. Later, you can think about how the program could be written to do that. Try writing down a brief sketch of an idea for a solution, and see what's wrong with that first idea. Maybe discuss the problem and how to solve it with a friend. Trying to explain something to a friend is a marvelous way of figuring out what's wrong with ideas, even better than writing them down; paper (or a computer) doesn't talk back at you and challenge your assumptions. Ideally, design isn't a lonely activity.

Unfortunately, there isn't a general strategy for problem solving that works for all people and all problems. There are whole books that claim to help you be better at problem solving and another huge branch of literature that deals with program design. We won't go there. Instead, we'll present a page's worth of suggestions for a general strategy for the kind of smaller problems an individual might face. After that, we'll quickly proceed to try out these suggestions on our tiny calculator problem.

When reading our discussion of the calculator program, we recommend that you adopt a more than usually skeptical attitude. For realism, we evolve our program through a series of versions, presenting the reasoning that leads to each version along the way. Obviously, much of that reasoning must be incomplete or even faulty, or we would finish the chapter early. As we go along, we provide examples of the kinds of concerns and reasoning that designers and programmers deal with all the time. We don't reach a version of the program that we are happy with until the end of the next chapter.

Please keep in mind that for this chapter and the next, the way we get to the final version of the program – the journey through partial solutions, ideas, and mistakes – is at least as important as that final version and more important than the language-technical details we encounter along the way (we will get back to those later).

6.2.1 Stages of development

Here is a bit of terminology for program development. As you work on a problem you repeatedly go through these stages:

- *Analysis:* Figure out what should be done and write a description of your (current) understanding of that. Such a description is called a *set of requirements* or a *specification*. We will not go into details about how such requirements are developed and written down. That's beyond the scope of this book, but it becomes increasingly important as the size of problems increases.
- *Design:* Create an overall structure for the system, deciding which parts the implementation should have and how those parts should communicate. As part of the design consider which tools – such as libraries – can help you structure the program.
- *Implementation:* Write the code, debug it, and test that it actually does what it is supposed to do.

6.2.2 Strategy

Here are some suggestions that – when applied thoughtfully and with imagination – help with many programming projects:

- What is the problem to be solved? The first thing to do is to try to be specific about what you are trying to accomplish. This typically involves constructing a description of the problem or – if someone else gave you such a statement – trying to figure out what it really means. At this point you should take the user's point of view (not the programmer/implementer's view); that is, you should ask questions about what the program should do, not about how it is going to do it. Ask: "What can this program do for me?" and "How would I like to interact with this program?" Remember, most of us have lots of experience as users of computers on which to draw.
  - Is the problem statement clear? For real problems, it never is. Even for a student exercise, it can be hard to be sufficiently precise and specific. So we try to clarify it. It would be a pity if we solved the wrong problem. Another pitfall is to ask for too much. When we try to figure out what we want, we easily get too greedy/ambitious. It is almost always better to ask for less to make a program easier to specify, easier to understand, easier to use, and (hopefully) easier to implement. Once it works, we can always build a fancier "version 2.0" based on our experience.

- Does the problem seem manageable, given the time, skills, and tools available? There is little point in starting a project that you couldn't possibly complete. If there isn't sufficient time to implement (including testing) a program that does all that is required, it is usually wise not to start. Instead, acquire more resources (especially more time) or (best of all) modify the requirements to simplify your task.

- Try breaking the program into manageable parts. Even the smallest program for solving a real problem is large enough to be subdivided.
  - Do you know of any tools, libraries, etc. that might help? The answer is almost always yes. Even at the earliest stage of learning to program, you have parts of the C++ standard library. Later, you'll know large parts of that standard library and how to find more. You'll have graphics and GUI libraries, a matrix library, etc. Once you have gained a little experience, you will be able to find thousands of libraries by simple web searches. Remember: There is little value in reinventing the wheel when you are building software for real use. When learning to program it is a different matter; then, reinventing the wheel to see how that is done is often a good idea. Any time you save by using a good library can be spent on other parts of your problem, or on rest. How do you know that a library is appropriate for your task and of sufficient quality? That's a hard problem. Part of the solution is to ask colleagues, to ask in discussion groups, and to try small examples before committing to use a library.
  - Look for parts of a solution that can be separately described (and potentially used in several places in a program or even in other programs). To find such parts requires experience, so we provide many examples throughout this book. We have already used **vector**, **string**, and **iostream**s (**cin** and **cout**). This chapter gives the first complete examples of design, implementation, and use of program parts provided as user-defined types (**Token** and **Token_stream**). Chapters 8 and 13–15 present many more examples together with their design rationales. For now, consider an analogy: If we were to design a car, we would start by identifying parts, such as wheels, engine, seats, door handles, etc., on which we could work separately before assembling the complete car. There are tens of thousands of such parts of a modern car. A real-world program is no different in that respect, except of course that the parts are code. We would not try to build a car directly out of raw materials, such as iron, plastics, and wood. Nor would we try to build a major program directly out of (just) the expressions, statements, and types provided by the language. Designing and implementing such

parts is a major theme of this book and of software development in general; see the discussions of user-defined types (Chapter 9), class hierarchies (Chapter 14), and generic types (Chapter 20).

- Build a small, limited version of the program that solves a key part of the problem. When we start, we rarely know the problem well. We often think we do (don't we know what a calculator program is?), but we don't. Only a combination of thinking about the problem (analysis) and experimentation (design and implementation) gives us the solid understanding that we need to write a good program. So, we build a small, limited version
  - To bring out problems in our understanding, ideas, and tools.
  - To see if details of the problem statement need changing to make the problem manageable. It is rare to find that we had anticipated everything when we analyzed the problem and made the initial design. We should take advantage of the feedback that writing code and testing give us.

  Sometimes, such a limited initial version aimed at experimentation is called a *prototype*. If (as is likely) our first version doesn't work or is so ugly and awkward that we don't want to work with it, we throw it away and make another limited version based on our experience. Repeat until we find a version that we are happy with. Do not proceed with a mess; messes just grow with time.
- Build a full-scale solution, ideally by using parts of the initial version. The ideal is to grow a program from working parts rather than writing all the code at once. The alternative is to hope that by some miracle an untested idea will work and do what we want.

6.3 Back to the calculator!

How do we want to interact with the calculator? That's easy: we know how to use **cin** and **cout**, but graphical user interfaces (GUIs) are not explained until Chapter 16, so we'll stick to the keyboard and a console window. Given expressions as input from the keyboard, we evaluate them and write out the resulting value to the screen. For example:

```
Expression: 2+2
Result: 4
Expression: 2+2*3
Result: 8
Expression: 2+3–25/5
Result: 0
```