Contents

Table of Figures

Introduction to Programming

An important note

You won't be surprised to hear that just about everyone has an opinion on what programming is, how to get started with it, and so on.

The purpose of this file is not to teach you how to program, but rather to teach you **what programming is**. That seems to be a question that's mostly ignored by those who would teach you to program - they assume that you already have an idea of what programming is, why you'd want to do it, what's basically involved, and so on. I will try to answer all those questions for you, so you can then decide whether you want to start learning **how** to program.

What is programming?

Given the general nature of that question, and the fact that programming is seen as a shockingly complex subject, you're probably expecting a highly complicated and technical answer. But you're not going to get one (sorry about that). In truth, it's quite easy to say what programming is, so I will:

Programming is breaking a task down into small steps.

That's just about the most honest and accurate answer I can give.
You're perhaps wondering what exactly I mean by breaking a task down into small steps, so I'll explain the point in more detail.

An example will help you to understand what I mean. Here's a task for you to do:

Put these words in alphabetical order: apple, zebra, abacus

I'm going to assume that you managed to put them in alphabetical order, and ended up with abacus, then apple, then zebra. If you didn't manage that, reading this article may soon become one of the worst experiences of your life so far.

Now think about exactly **how** you performed that task; what steps you took to put the words in alphabetical order, and what you required to know in order to do so. The most obvious thing you needed to know was the alphabet; the desired order of the letters. Then, if you're like me, you probably did something like this:

1. Look through the words for one beginning with "A".
2. If you found a word beginning with "A", put that word at the beginning of the list (in your mind).
3. Look for another word beginning with "A".
4. If there's another word beginning with "A", compare its second letter with the second letter of our first "A" word.
5. If the second letters are different, put the two words in alphabetical order by their second letter. If the second letters are the same, proceed to the third letter, and so on.
6. Repeat this whole process for "B" and each other letter in alphabetical order, until all the words have been moved to the appropriate place.

Your method may differ slightly, but probably not by much. The thing to notice (which I noticed particularly, since I'm having to type all this) is how much time it took to explain a process which happens without any real conscious thought. When you saw that you had to put some words in alphabetical order, you certainly didn't first sit down and draw up a plan of what you were going to do, detailing all the steps I listed above. Your mind doesn't need to; you learned to do it once when you were a child, and now it just happens. You have a kind of built-in shortcut to that sequence of steps.

Now let's try another simple example:

Count the number of words in this sentence: "Programming really can be fun."

Hopefully you decided that there were five words. But how did you come to that decision? First you had to decide what a word is, naturally. Let's assume for the moment that a word is a sequence of letters which is separated from other words by a space. Using that rule, you do indeed get five words when looking at the sentence above. But what about this sentence:

"Sir Cecil Hetherington-Smythe would make an excellent treasurer,minister."

Notice my deliberate mistake: I didn't leave a space after the comma. You might also feel it's a mistake to name anyone Cecil Hetherington-Smythe, but that's a debate for another time. Using our rule about sequences of letters which are separated from other sequences by spaces, you would decide that there were eight words in the new sentence. However, I think we can agree that there are in fact ten words, so our rule clearly isn't working. Perhaps if we revised our rule to say that words can be separated by spaces, commas or dashes, instead of just spaces. Using that new rule, you'd indeed find ten words. Now let's try another sentence:

"Some people just love to type really short lines, instead of using the full width of the page."

Although it might not be obvious, there is no space after "really", nor is there a space after "full". Instead, I took a new line by pressing the return key. So, using our newest rule, how many words would you find in that last sentence? I'll tell you: you'd get sixteen, when in fact there are eighteen. This means that we need to revise our rule yet again, to include returns as valid word-separators. And so on, until another sentence trips up our rule, and we need to revise it yet again.

You might wonder why we're doing this at all, because after all, we all **know** what we mean when we say "count the number of words". You can do it properly without thinking about any rules or valid word-

separators or any such thing. So can I. So can just about anyone. What this example has shown us is that we take for granted something which is actually a pretty sophisticated "program" in our minds. In fact, our own built-in word counting "program" is so sophisticated that you'd probably have a lot of trouble describing all the actual little rules it uses. So why bother?

The answer comes in the form of an exceptionally important truth which you must learn. It's to do with computers (even your own computer that you're using to read this). Here it is:

Computers are very, very stupid.

Nevertheless, it's true - computers are desperately stupid. Your computer will sit there and do whatever mindless task you tell it to, for days, weeks, months or years on end, without any complaints or any slacking-off. That's not the typical behavior of something that's even slightly clever. It will also happily erase its own hard disk (which is a bit like you deleting all your memories then pulling parts of your brain out), so we're clearly not dealing with an intimidatingly intelligent item.

Programmers tell computers what to do. Computers require these instructions to be precise and complete in every way. Humans aren't usually good at giving precise and complete instructions since we have this incredible brain which lets us give vague commands and still get the correct answer. Thus, programmers have to learn to think in an unnatural way: they have to learn to take a description of a task (like "count the number of words in a sentence"), and break it down into the fundamental steps which a computer needs to know in order to perform that task.

You may be feeling slightly uneasy at this point. I've admitted that programming is, in a way, unnatural. I've warned you about the spectacular stupidity of computers, so you're probably getting a small idea of the amount of task-description you'd need to do in order to make your computer do anything even vaguely impressive. But don't worry; there are some excellent reasons to become a programmer:

- Programmers make lots of money.
- Programming really is fun.
- Programming is very intellectually rewarding.
- Programming makes you feel superior to other people.
- Programming gives you complete control over an innocent, vulnerable machine, which will do your evil bidding with a loyalty not even your pet dog can rival.

Who can be a programmer?

You might be wondering just who can become a programmer. The dull thing to do would be to say "anyone!", but I won't do that, because it's just not true. There are a few traits which might indicate that the person would be a good programmer:

- Logical
- Patient
- Perceptive

- At least moderately intelligent
- Enjoys an intellectual challenge
- Star Trek fan
- Female *

* Actually, males and females make equally good programmers. I just said that to address the gender disparity which exists in the programming world; it's true that there are currently more male programmers than female, which is strange given that one of the first ever programmers (Ada Lovelace) was female. Besides, male programmers are especially willing to encourage more females to take up the profession, since they (the male programmers) traditionally have no social lives whatsoever.

What's actually involved in programming?

In this section I'm going to introduce some technical terms. It's unavoidable. The jargon will always get you in the end (if you choose to become a programmer, soon you'll actually be **talking** in jargon). However, I'll explain each term as we come to it, and there's a list of jargon in the following section.
(With explanations, of course).

So, what's actually involved in programming - the actual process of writing programs? I'll answer that myself even though I asked the question, since I'm assuming that you don't know yet. Here's a quick overview of the process:

1. *Write a program.*
2. *Compile the program.*
3. *Run the program.*
4. *Debug the program.*
5. *Repeat the whole process until the program is finished.*

Let's discuss those steps one by one, shall we? Yes, we shall. And here goes.

1. Write a program.

I have a small amount of bad news for you: you can't write programs in English. It would be nice indeed to be able to type "count the number of words in a sentence" into your computer and have it actually understand, but that's not going to happen for a while (unless someone writes a program to make a computer do that, of course). Instead, you have to learn a **programming language**.

Thankfully, learning a programming language is nothing like that - for one thing, much of a programming language is indeed in English. Programming languages commonly use words like "if", "repeat", "end" and such. Also, they use the familiar mathematical operators like "+" and "=". It's just a matter of learning the "grammar" of the language; how to say things properly. Since there are many possible languages you could choose to learn I won't go into any specific one here. I just wanted to make you aware of the fact that you won't be typing your programs in English.

So, we said "Write a program". This means: write the steps needed to perform the task, using the

programming language you know. You'll do the typing in a **programming environment** (an application program which lets you write programs, which is an interesting thought in itself).

Incidentally, the stuff you type to create a program is usually called **source code**, or just **code**. Programmers also sometimes call programming **coding**.

2. Compile the program.

In order to use a program, you usually have to **compile** it first. When you write a program (in a programming language, using a programming environment, as we mentioned a moment ago), it's not yet in a form that the computer can use. This isn't hard to understand, given that computers actually only understand lots of 1s and 0s in long streams. You can't very well write programs using only vast amounts of 1s and 0s, so you write it in a more easily-understood form (a programming language), then you convert it to a form that the computer can actually use. This conversion process is called **compiling**, or **compilation**. Not surprisingly, a program called a **compiler** does the compiling.

It's worth mentioning that if your program has problems which the compiler can't deal with, it won't be able to compile your program.

You'll be pleased to hear that your programming environment will include a suitable compiler (or maybe more than one compiler: each different programming language your programming environment allows you to use requires its own compiler). Compilers are just fancy programs, so they too are written by programmers. Programmers who write compilers are a bit like gods; they make it possible for everyone else to program.

3. Run the program.

Now that you've compiled the program into a form that the computer can use, you want to see if it works: you want to make the computer perform the steps that you specified. This is called **running** the program, or sometimes **executing** it. Just the same as how a car isn't much use if you don't drive it, a program isn't much use if you don't run it. Your programming environment will allow you to run your program too (as you can see, programming environments do rather a lot for you).

4. Debug the program.

You've probably heard the term "**debug**" before (it's pronounced just as you might expect: "Dee- bug"). It refers to fixing errors and problems with your program. The term became used to describe any kind of problem-solving process in relation to computers, and we use it today to refer purely to fixing errors in our code.

You may also have heard the phrase "it's not a bug, it's a feature". Programmers sometimes say this when someone points out a problem with their programs; they're saying that it's not a bug, but rather a deliberate design choice (which is almost always a lie).

Once again, your programming environment will help you to debug your programs. You usually debug your program by **stepping** through it. This means just what it sounds like: you go through your program one step at a time, watching how things are going and what's happening. Sooner or later (usually later),

you'll see what's going wrong, and slap yourself upside the head at the extremely obvious error you've made. Ahem.

5. Repeat the whole process until the program is finished.

And then you repeat the whole process until you're happy with the program. This is trickier than it might sound, since programmers are never happy with their programs. You see, programmers are perfectionists - never satisfied until absolutely everything is complete and elegant and powerful and just gorgeous. Programmers will commonly release a new version of their program every day for a couple of weeks after the initial.

And that's the basic process of programming. Note that most programming environments will make a lot of it much easier for you, by doing such things as:

- Warning you about common errors
- Taking you to the specific bit of code which is causing the compiler to stop
- Letting you quickly look up documentation on the programming language you're using
- Letting you just choose to run the program, and compiling it automatically first
- Coloring parts of your code to make it easier to read (for example, making numbers a different color from other text)
- And many other things

So, don't worry too much about the specifics of compiling then running then debugging or whatever. The purpose of this section was mostly to make you aware of the cyclical nature of programming: you write code, test it, fix it, write more, test it, fix, and so on.

Complete Process of Program execution

You use a text editor to create a C++ program following the rules, or syntax, of the high-level language. This program is called the source code, or source program. The program must be saved in a text file that has the extension .cpp. For example, if you saved the preceding program in the file named FirstCPPProgram, then its complete name is FirstCPPProgram.cpp.

Source program: A program written in a high-level language.

2. The C++ program may contain the statement #include <iostream>. In a C++ program, statements that begin with the symbol # are called preprocessor directives. These statements are processed by a program called preprocessor.

3. After processing preprocessor directives, the next step is to verify that the program obeys the rules of the

programming language—that is, the program is syntactically correct—and translate the program into the equivalent machine language. The compiler checks the source program for syntax errors and, if no error is found, translates the program into the equivalent machine language. The equivalent machine language program is called an object program.

Object program: The machine language version of the high-level language program.

4. The programs that you write in a high-level language are developed using an integrated development environment (IDE). The IDE contains many programs that are useful in creating your program. For example, it contains the necessary code (program) to display the results of the program and several mathematical functions to make the programmer's job somewhat easier. Therefore, if certain code is already available, you can use this code rather than writing your own code. Once the program is developed and successfully compiled, you must still bring the code for the resources used from the IDE into your program to produce a final program that the computer can execute. This prewritten code (program) resides in a place called the library. A program called a linker combines the object program with the programs from libraries.
Linker: A program that combines the object program with other programs in the library and is used in the program to create the executable code.

5. You must next load the executable program into main memory for execution. A program called a loader accomplishes this task.

Loader: A program that loads an executable program into main memory.

6. The final step is to execute the program.

As a programmer, you need to be concerned only with Step 1. That is, you must learn,
Understand, and master the rules of the programming language to create source programs.

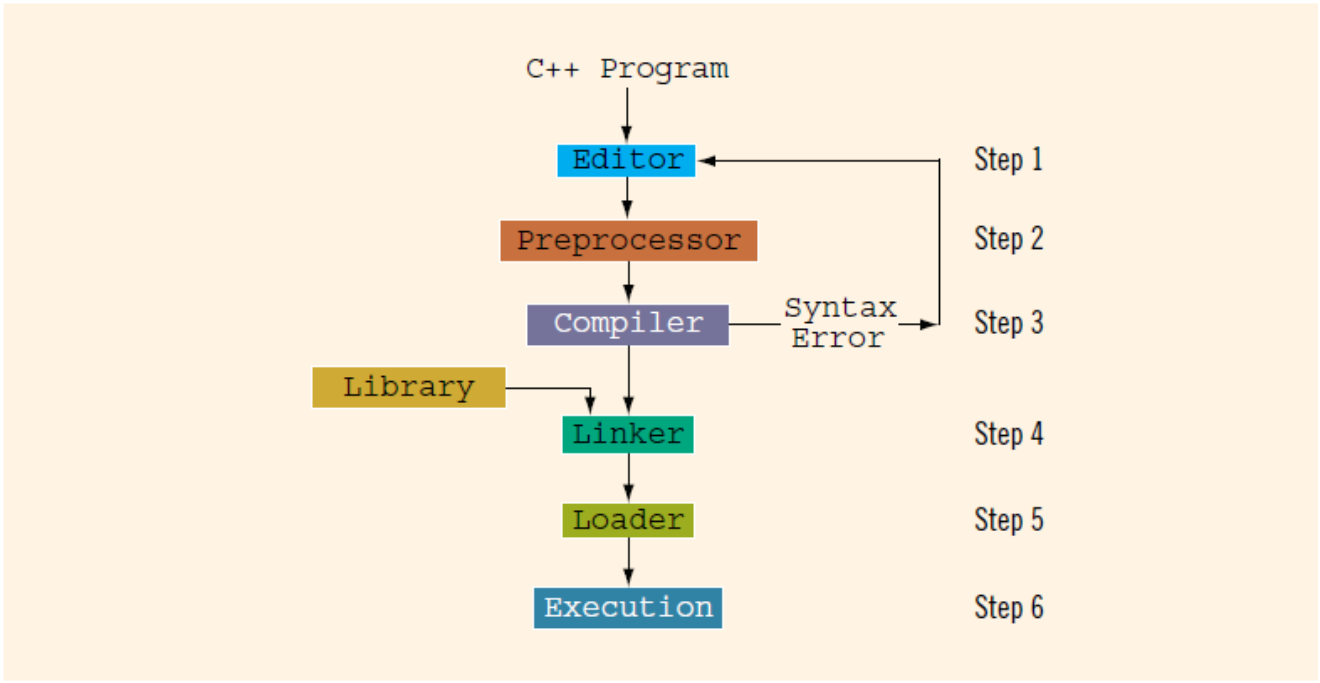

Figure 1: Program execution process

Algorithm: A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

In a programming environment, the problem-solving process requires the following three steps:

1. Analyze the problem, outline the problem and its solution requirements, and design an algorithm to solve the problem.

2. Implement the algorithm in a programming language, such as C++, and verify that the algorithm works.

3. Maintain the program by using and modifying it if the problem domain changes.

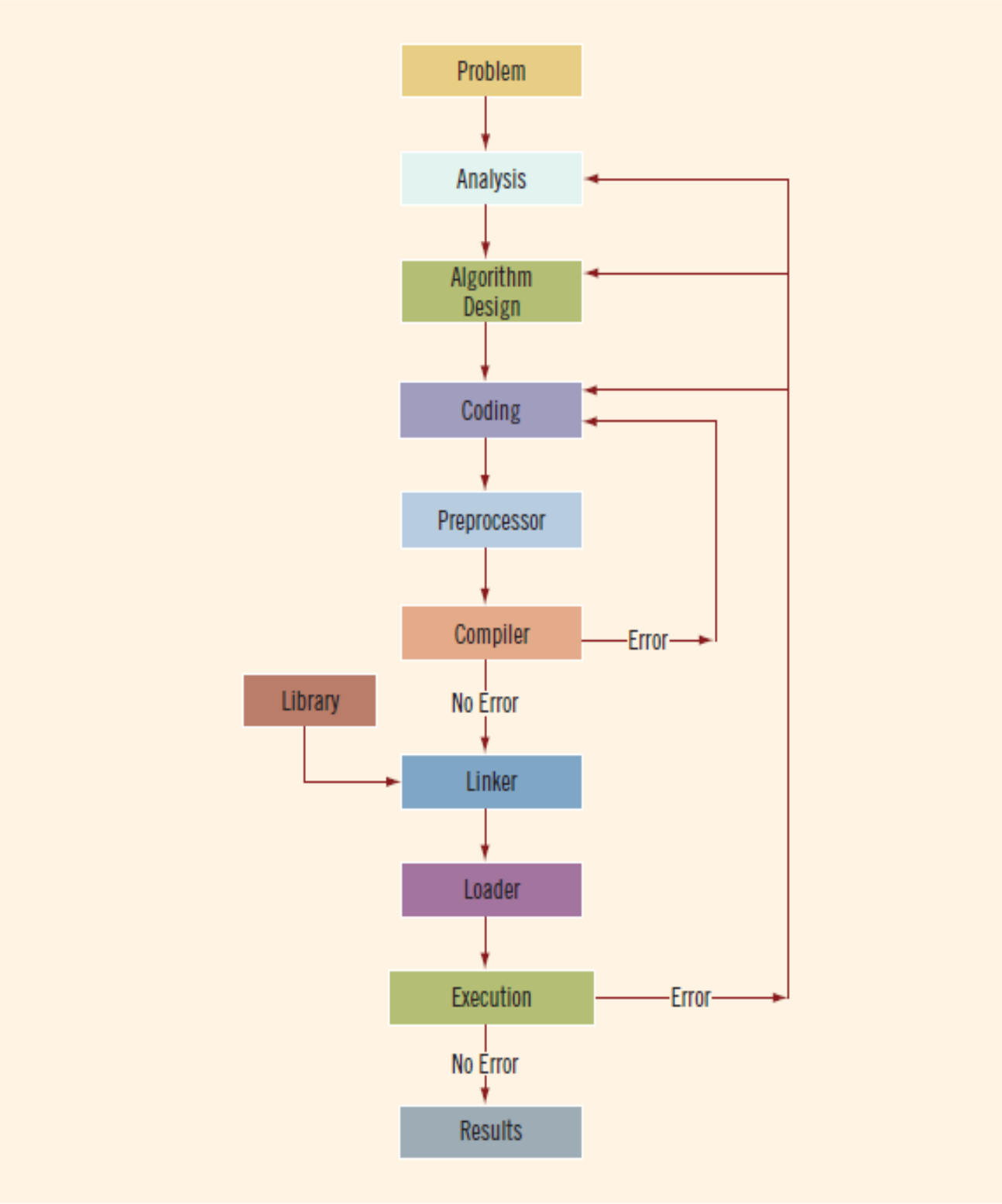

Figure 2: Program start to end Execution Process

Difference between A library and Header file?

In simple terms, library contains function body whereas header file contains function prototype. Example: math.h is a header file which includes function prototype for function calls like sqrt(),pow() etc.{ libm.so is the library file used for these type of functions.} libc.so is a library file which includes function body for function calls like printf(),scanf() etc. {stdio.h is the header file for these functions.}

Namespace?

Environment Variables and path??