Creating Programs
C Programming Edition

What is programming?[1]

Given the general nature of that question, and the fact that programming is seen as a hideously complex subject, you're probably expecting a highly convoluted 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. It also has the added benefit of being concise, and sounding very much like something you'd read in an official book on the topic, thus adding to my credibility.

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. Let me start by giving you a fact about programmers that you'll find very easy to believe:

Programmers think in an unnatural way.

This refers just to the fact that programming is breaking a task into small steps, and that's not the usual way that your mind works. 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:

- Look through the words for one beginning with "A".
- If you found a word beginning with "A", put that word at the beginning of the list (in your mind).
- Look for another word beginning with "A".
- If there's another word beginning with "A", compare its second letter with the second letter of our first "A" word.

[1] https://scotlandsoftware.com/introduction-to-programming/ Scottland Software – Matt Gemmell

- 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.
- 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.

To some people, that statement is almost sacrilegious. You can understand that, because computers are really expensive. If you've just bought a Ferrari, you probably don't want your neighbor to come along and say that it's ugly and slow. 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 it's 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.

In fact, computers are so painfully stupid that they require to be told, in minute detail, how to do even the most laughably simple of tasks. It's quite pathetic, when you think about it (or perhaps we're pathetic, since we're willing to pay ridiculous amounts of money to own them). In fact, just about the only positive thing about computers is that they're completely obedient. No matter how crazed your instructions might be, your computer will carry them out precisely.

By now, hopefully you can see how this is all tying together. 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.

Software Development Cycle

To overcome this "unnatural way of thinking" and to develop a well written, working program, certain steps should be followed. These steps are known as the **Program Development Cycle**.

The first step is to **define the problem**. If you are incapable of stating what the program should do, how it will do it and what the program should output, then it is impossible to write the program.

If you cannot do the task yourself, you cannot be expected to program a computer to do it correctly.

The next step is to **plan the solution** by developing the algorithm to solve the problem. An algorithm is defined as "*A formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point.*"[2] As stated in the introduction, computers are stupid, so you must break down the problem into the smallest steps possible.

If the task the program is to perform is complex, the task will be broken up into smaller tasks and each is programmed separately. These smaller tasks are called functions, sub routines, or procedures. These sub routines are then tied together using a MAIN LINE PROGRAM that calls each sub routine in turn. Sub routines can be reused many times within the main line program.

Only when the first two steps are complete should you sit down and start writing **code** by converting your flowchart to C Code. Well planned code will have fewer mistakes and take less time to debug.

Once the code is written, **type it into** the computer, while commenting along the way.

Once you are finished, **compile the code**. At this point, the compiler will report any errors in the syntax. Think of syntax as the grammar and spelling of the particular programming language you are working in.

If error(s) occurred, go back and correct them and recompile. This process usually takes a number of tries.

Once you get compiled code, **run the code and check the output** the program gives you is correct for several test sets of inputs. If the program gives an incorrect output, a LOGIC ERROR has occurred and you must go back and find the error in the code and correct it.

---

[2] http://www.webopedia.com/TERM/a/algorithm.html

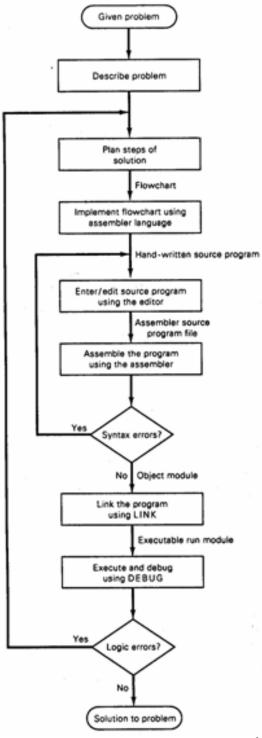The programming development cycle is summarized in the flowchart in Figure 1.

Figure 1 - Programming Development Cycle[3]

Programming Style[4]

White Space

Write code that is as easy as possible to read and maintain. Adding white space in the form of blank lines, spaces, and indentations will significantly improve the readability of your code.

Blank Lines

A careful use of blank lines between code "paragraphs" can greatly enhance readability by making the logical structure of a sequence of lines more obvious.

Indentation

Use indentations to show the logical structure of your code. Typically this is done using the TAB key:

```
main()
{
        int c;

        c = getchar();

        while (c!= EOF)
        {
                putchar(c);
                c=getchar();
        }
}
```

Names

Use descriptive variable and function names where possible.

For microcontroller setup registers and bits, use the name in the processor documentation wherever possible.

This will help document the variable and functions without the need for comments.

---

[4] Most of this section is from C Style Guide, NASA SEL-94-003
http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf

Comments

Judiciously placed comments in the code can provide information that a person could not discern simply by reading the code. Comments can be added at many different levels

- At the program level, you can include a README file that provides a general description of the program and explains its organization
- At the file level, it is good practice to include a file prolog that explains the purpose of the file and provides other information (discussed in more detail below)
- At the function level, a comment can serve as a funtion prolog.
- Throughout the file, where data are being declared or defined, it is helpful to add comments to explain the purpose of the variable.

Comments can be written in several styles depending on their purpose and length. Use comments to **add information** for the reader or to **highlight sections** of code. Do not paraphrase the code or repeat information contained in the program.

This section describes the use of comments and provides examples:
- Boxed comments – Used for prologs or as section separators
- Block comments – Use at the beginning of each major section of the code as a narrative description of that portion of the code (main and functions)
- Short comments – Write on the same line as the code or data definition they describe
- Inline comments – Write at the same level of indentation as the code they describe:

Examples: Boxed Comment

```
/*************************************************
 *                                               *
 *                                               *
 *                                               *
 *                                               *
 *                                               *
 *************************************************/
```

Or

```
/*************************************************



*************************************************/
```

Example: Section Separator

```
/************************************************/
```

Or

```
/**************** some text here ****************/
```

Example: Short Comments

```
int f[];        /* array temperature readings */
char a;         /* hold single character input */
int count;      /* loop counter */
```

- Tab comment over far enough to separate it from code statements
- If more than one short comment appears in a block of code or data definition, start all of them at the same tab position and end all at the same position.

Example: inline comment

```
switch (ref_type)
{

    /* perform case for either s/c position or velocity
     *vector request using the RSL routine c_calpvs */

    case 1:

    case 2:

    ...

    case n:
}
```

In general, use short comments to document variable definitions and block comments to describe computations processes

Example: block comment vs. short comment

preferred style:

```
/* Main sequence: get and process all user requests */

while (!finish())
{
    inquire();
    process();
}
```

8

not recommended:

```
while (!finish())       /* Main sequence:    */
{                       /*                   */
      inquire();        /* Get user request  */
      process();        /* And carry it out  */
}                       /* As long as possible */
```

COMMENTS ON COMMENTS[5]

Writing comments is an art form. In my experience, seasoned programmers often do it badly and beginners are much worse. There are no good hard-and-fast rules for commenting, but the following are some rules of thumb that I consider important:

- You are the main user of your comments. Comment as though you expect to come back in six months and modify or debug the program, remembering nothing about how it works. In particular, your comments are intended for someone who knows how individual instructions work.

    Comments such as

    ```
    a=a+1;              // Add 1 to ax <== VERY BAD STYLE
    ```

    are almost always worthless!

    Comments should answer the question "WHY DID I DO THIS?" Anyone with a little ASM knowledge knows that the above add's one to 'a', but they will not know why you incremented the 'a' register. For Example:

    ```
    a=a+1;          // count number of times loop executes
    ```

    tells that 'a' is a loop counter and that it is incremented every time the loop executes.

- The next main user of your comments is the poor sap who must maintain your code when you have been promoted to a more exalted position. When in doubt, over-comment.

- Do comments as you write the code. The usual student practice of writing a "bare" program first and then adding the comments when it's debugged is counterproductive. In the first place, you lose the main utility of the comments: debugging your code. In the second, adding the comments often introduces bugs (such as omitting the ';').

[5] Most of this section is from "Assembly Language Programming for the IBM PC Family 3rd ed by William B Jones, Scott Jones Publishers, 2001

- Use blank lines to separate code. This will make the code easier to read and allow you to comment blocks of code with a single comment (see below).

- Keep your code in columns (as shown in the previous section). This will make the code much easier to read and help in the debugging process.

- Do not try to comment every line. Many textbook authors make a comment-per-line rule in assembly language, but I don't believe in it. Many program lines are so straightforward that a comment would hide more than it reveals, and other lines require several lines of comments to fully explain them. The comment-per-line rule is a way of avoiding consideration of what really constitutes useful commenting.

- On the other hand, one (excellent) text I have read advocates *never* having a comment that applied only to a single line of code. This rule is worth keeping in mind but not following slavishly.

- When you change your program, change the comments. If there is anything worse than a program without comments, it is a program in which the comments don't agree with the code! When you add new code, add appropriate comments with it.

- If a section of code works together to perform a function, comment it as a group and leave a blank line before and after the code segment to separate it from the rest of the code.

- Start programs and sub procedures with general comments concerning name, programmer, copyright notice, if any, etc. Most shops have a standard heading. You should also include a description of how the program is used (e.g., calling sequence if it is a sub procedure)

For this course we will use the following standard header:

```
/*****************************************************************

  Name:      Program name (aka Label used to call the
             sub procedure)

  Author:    Your Name

  Date:             The date the code was started

  Revisions:        (only when required - see below)

  Function: A description of what the program (or sub procedure)
             does.

  Procedure How the code does what it does (for more complex
             programs and sub procedures. Most of the time the
             function provides enough detail so this is not
             required)

  --------------- rest are for Sub Procedures ONLY -------------

  On call:  What needs to be where when the function is called

  Returns:  What will be where when the function is completed

  *****************************************************************/
```

- Programs that are going to be used for any length of time should have a revision history somewhere in their standard heading. After the preliminary debugging is completed (i.e., when the original programmer thinks he or she is done), further changes should be described at the beginning of the pro*gram* with date of change and the name or initials of the programmer making the change. Each programming shop has its own format, but one possible form is

```
/**********************************************

                REVISION HISTORY

  2/14/96 Handle negative argument correctly WBJ
  3/24/96 Fix bug introduced by neg arg fix LIV
  5/9/96 REALLY fix neg arg problem this time WBJ

  /**********************************************
```