# Ch 1 - The way of the program

The goal of this book, and this class, is to teach you to think like a computer scientist. I like the way computer scientists think because they combine some of the best features of Mathematics, Engineering, and Natural Science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem-solving**. By that I mean the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The way of the program."

## What is a programming language?

The programming language you will be learning is C, which was developed in the early 1970s by Dennis M. Ritchie at the Bell Laboratories. C is an example of a **high-level language**; other high-level languages you might have heard of are Pascal, C++ and Java.

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as machine language or assembly language. Loosely-speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is much easier to program in a high-level language; by "easier" I mean that the program takes less time to write, it's shorter and easier to read, and it's more likely to be correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.

A compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

As an example, suppose you write a program in C. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named **program.c**, where "program" is an arbitrary name you make up, and the suffix **.c** is a convention that indicates that the file contains C source code.

## What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The instructions, which we will call **statements**, look different in different programming languages, but there are a few basic operations most languages can perform:

* **input:** Get data from the keyboard, or a file, or some other device.
* **output:** Display data on the screen or send data to a file or other device.
* **math:** Perform basic mathematical operations like addition and multiplication.
* **testing:** Check for certain conditions and execute the appropriate sequence of statements.
* **repetition:** Perform some action repeatedly, usually with some variation.

That's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of statements that perform these operations. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these basic operations.

## What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

## Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. For most readers, a few syntax errors are not a significant problem, but compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

## Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program. Programming in C allows you to get very close to the actual computing hardware. Most run-time errors C occur because the language provides no protection against the accessing or overwriting of data in memory. For the simple sorts of programs we will be writing for the next few weeks, run-time errors are rare, so it might be a little while before you encounter one.

## Logic errors and semantics

The third type of error is the logical or semantic error. If there is a logical error in your program, it will compile and run successfully, but it will not do the right thing. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

## Experimental debugging

One of the most important skills you will acquire in this class is debugging. Debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

Debugging is like detective work, where you are confronted with clues and you have to infer the processes and events that lead to the results you see. Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one.

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does something, and make small modifications, debugging them as you go, so that you always have a working program.

In later chapters, more suggestions about debugging and other programming practices will be made.

## Formal and natural languages

**Natural languages** are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

**Formal languages** are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

**Programming languages are formal languages that have been designed to express computations.**

As I mentioned before, formal languages tend to have strict rules about syntax. For example, $3+3=6$ is a syntactically correct mathematical statement, but $3=+6\$$ is not. Also, $H\_2O$ is a syntactically correct chemical name, but $\_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with 3=+6$ is that $ is not a legal token in mathematics (at least as far as I know). Similarly, \_2Zz is not legal because there is no element with the abbreviation Zz.

The second type of syntax rule pertains to the structure of a statement; that is, the way the tokens are arranged. The statement 3=+6$ is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common---tokens, structure, syntax and semantics---there are many differences.

ambiguity:

Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy:

In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness:

Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry:

Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose:

The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.

Programs:

The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

## The first program

**Traditionally** the first program people write in a new language is called "Hello, World." because all it does is display the words "Hello, World." In C, this program looks like this:

#include <stdio.h>

#include <stdlib.h>

/\* main: generate some simple output \*/

int main(void)

{

printf("Hello, World.\n");

return(EXIT\_SUCCESS);

}

Some people judge the quality of a programming language by the simplicity of the "Hello, World." program. By this standard, C does reasonably well. Even so, this simple program contains several features that are hard to explain to beginning programmers. For now, we will ignore some of them, like the first two lines.

## Comment

The third line begins with /\* and ends with \*/, which indicates that it is a **comment**. A comment is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a /\*, it ignores everything from there until it finds the corresponding \*/.

## Main Function

In the forth line, you notice the word main. main is a special name that indicates the place in the program where execution begins. When the program runs, it starts by executing the first **statement** in main() and it continues, in order, until it gets to the last statement, and then it quits.

## Output Statement

There is no limit to the number of statements that can be in main(), but the example contains only two. The first is an **output** statement, meaning that it displays or prints a message on the screen. The statement that prints things on the screen is printf(), and the characters between the quotation marks will get printed. Notice the \n after the last character. This is a special character called newline that is appended at the end of a line of text and causes the cursor to move to the next line of the display. The next time you output something, the new text appears on the next line. At the end of the statement there is a semicolon (;), which is required at the end of every statement.

## Syntax

There are a few other things you should notice about the syntax of this program. First, C uses curly-brackets ({ and }) to group things together. In this case, the output statement is enclosed in curly-brackets, indicating that it is inside the definition of main(). Also, notice that the statement is indented, which helps to show visually which lines are inside the definition.

At this point it would be a good idea to sit down in front of a computer and compile and run this program. The details of how to do that depend on your programming environment, this book assumes that you know how to do it.

As I mentioned, the C compiler is very pedantic with syntax. If you make any errors when you type in the program, chances are that it will not compile successfully. For example, if you misspell stdio.h, you might get an error message like the following:

hello\_world.c:1:19: error: sdtio.h: No such file or directory

There is a lot of information on this line, but it is presented in a dense format that is not easy to interpret. A more friendly compiler might say something like:

“On line 1 of the source code file named hello\_world.c, you tried to include a header file named sdtio.h. I didn't find anything with that name, but I did find something named stdio.h. Is that what you meant, by any chance?”

Unfortunately, few compilers are so accommodating. The compiler is not really very smart, and in most cases the error message you get will be only a hint about what is wrong. It will take some time for you to learn to interpret different compiler messages.

Nevertheless, the compiler can be a useful tool for learning the syntax rules of a language. Starting with a working program (like hello\_world.c), modify it in various ways and see what happens. If you get an error message, try to remember what the message says and what caused it, so if you see it again in the future you will know what it means.

## Glossary

problem-solving:

The process of formulating a problem, finding a solution, and expressing the solution.

high-level language:

A programming language like C that is designed to be easy for humans to read and write.

low-level language:

A programming language that is designed to be easy for a computer to execute. Also called "machine language" or "assembly language."

formal language:

Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language:

Any of the languages people speak that have evolved naturally.

portability:

A property of a program that can run on more than one kind of computer.

interpret:

To execute a program in a high-level language by translating it one line at a time.

compile:

To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code:

A program in a high-level language, before being compiled.

object code:

The output of the compiler, after translating the program.

executable:

Another name for object code that is ready to be executed.

statement:

A part of a program that specifies an action that will be performed when the program runs. A print statement causes output to be displayed on the screen.

comment:

A part of a program that contains information about the program, but that has no effect when the program runs.

algorithm:

A general process for solving a category of problems.

bug:

An error in a program.

syntax:

The structure of a program.

semantics:

The meaning of a program.

parse:

To examine a program and analyze the syntactic structure.

syntax error:

An error in a program that makes it impossible to parse (and therefore impossible to compile).

logical error:

An error in a program that makes it do something other than what the programmer intended.

debugging:

The process of finding and removing any of the three kinds of errors.

Permission is granted to copy, distribute, transmit and adapt this work un- der the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Inter- national License: https://creativecommons.org/licenses/by-nc/4.0/. If you are interested in distributing a commercial version of this work, please contact the author(s). The LATEX source and code for this book is available from: https://github.com/mkpenta/ThinkC