**Elements of Real Programming Languages**

[**https://www.eskimo.com/~scs/cclass/progintro/sx4.html**](https://www.eskimo.com/~scs/cclass/progintro/sx4.html)

There are several elements which programming languages, and programs written in them, typically contain. These elements are found in all languages, not just C. If you understand these elements and what they're for, not only will you understand C better, but you'll also find learning other programming languages, and moving between different programming languages, much easier.

1. There are *variables* or *objects*, in which you can store the pieces of data that a program is working on. Variables are the way we talk about memory locations (data), and are analogous to the ``registers'' in our pocket calculator example. Variables may be *global* (that is, accessible anywhere in a program) or *local* (that is, private to certain parts of a program).

2. There are *expressions*, which compute new values from old ones.

3. There are *assignments* which store values (of expressions, or other variables) into variables. In many languages, assignment is indicated by an equals sign; thus, we might have

b = 3

or

c = d + e + 1

The first sets the variable b to 3; the second sets the variable c to the sum of the variables d plus e plus 1.

The use of an equals sign can be mildly confusing at first. In mathematics, an equals sign indicates equality: two things are stated to be inherently equal, for all time. In programming, there's a time element, and a notion of cause-and-effect: after the assignment, the thing on the left-hand side of the assignment statement is equal to what the stuff on the right-hand side was before. To remind yourself of this meaning, you might want to read the equals sign in an assignment as ``gets'' or ``receives'': a = 3 means ``a gets 3'' or ``a receives 3.''

(A few programming languages use a left arrow for assignment

a <-- 3

to make the ``receives'' relation obvious, but this notation is not too popular, if for no other reason than that few character sets have left arrows in them, and the left arrow key on the keyboard usually moves the cursor rather than typing a left arrow.)

If assignment seems natural and unconfusing so far, consider the line

i = i + 1

What can this mean? In algebra, we'd subtract i from both sides and end up with

0 = 1

which doesn't make much sense. In programming, however, lines like

i = i + 1

are extremely common, and as long as we remember how assignment works, they're not too hard to understand: the variable i receives (its new value is), as always, what we get when we evaluate the expression on the right-hand side. The expression says to fetch i's (old) value, and add 1 to it, and this new value is what will get stored into i. So i = i + 1 adds 1 to i; we say that it *increments* i.

(We'll eventually see that, in C, assignments are just another kind of expression.)

4. There are *conditionals* which can be used to determine whether some condition is true, such as whether one number is greater than another. (In some languages, including C, conditionals are actually expressions which compare two values and compute a ``true'' or ``false'' value.)

5. Variables and expressions may have *types*, indicating the nature of the expected values. For instance, you might declare that one variable is expected to hold a number, and that another is expected to hold a piece of text. In many languages (including C), your *declarations* of the names of the variables you plan to use and what types you expect them to hold must be explicit.

There are all sorts of data types handled by various computer languages. There are single characters, integers, and ``real'' (floating point) numbers. There are text strings (i.e. strings of several characters), and there are *arrays* of integers, reals, or other types. There are types which reference (point at) values of other types. Finally, there may be user-defined data types, such as *structures* or*records*, which allow the programmer to build a more complicated data structure, describing a more complicated object, by accreting together several simpler types (or even other user-defined types).

6. There are *statements* which contain instructions describing what a program actually does. Statements may compute expressions, perform assignments, or call functions (see below).

7. There are *control flow constructs* which determine what order statements are performed in. A certain statement might be performed only if a condition is true. A sequence of several statements might be repeated over and over, until some condition is met; this is called a *loop*.

8. An entire set of statements, declarations, and control flow constructs can be lumped together into a *function* (also called *routine*, *subroutine*, or *procedure*) which another piece of code can then *call* as a unit.

When you call a function, you transfer control to it and wait for it to do its job, after which it *returns* to you; it may also return a value as a result of what it has done. You may also pass values to the function on which it will operate or which otherwise direct its work.

Placing code into functions not only avoids repetition if the same sequence of actions must be performed at several places within a program, but it also makes programs easy to understand, because you can see that some function is being called, and performing some (presumably) well-defined subtask, without always concerning yourself with the details of how that function does its job. (If you've ever done any knitting, you know that knitting instructions are often written with little sub-instructions or patterns which describe a sequence of stitches which is to be performed multiple times during the course of the main piece. These sub-instructions are very much like function calls in programming.)

9. A set of functions, global variables, and other elements makes up a *program*. An additional wrinkle is that the source code for a program may be distributed among one or more *source files*. (In the other direction, it is also common for a suite of related programs to work closely together to perform some larger task, but we'll not worry about that ``large scale integration'' for now.)

10. In the process of specifying a program in a form suitable for a compiler, there are usually a few logistical details to keep track of. These details may involve the specification of compiler parameters or interdependencies between different functions and other parts of the program. Specifying these details often involves miscellaneous syntax which doesn't fall into any of the other categories listed here, and which we might lump together as ``boilerplate.''

Many of these elements exist in a hierarchy. A program typically consists of functions and global variables; a function is made up of statements; statements usually contain expressions; expressions operate on objects. (It is also possible to extend the hierarchy in the other direction; for instance, sometimes several interrelated but distinct programs are assembled into a suite, and used in concert to perform complex tasks. The various ``office'' packages--integrated word processor, spreadsheet, etc.--are an example.)

As we mentioned, many of the concepts in programming are somewhat arbitrary. This is particularly so for the terms *expression*, *statement*, and *function*. All of these could be defined as ``an element of a program that actually does something.'' The differences are mainly in the level at which the ``something'' is done, and it's not necessary at this point to define those ``levels.'' We'll come to understand them as we begin to write programs.

An analogy may help: Just as a book is composed of chapters which are composed of sections which are composed of paragraphs which are composed of sentences which are composed of words (which are composed of letters), so is a program composed of functions which are composed of statements which are composed of expressions (which are in fact composed of smaller elements which we won't bother to define). Analogies are never perfect, though, and this one is weaker than most; it still doesn't tell us anything about what expressions, statements, and functions really*are*. If ``expression'' and ``statement'' and ``function'' seem like totally arbitrary words to you, use the analogy to understand that what they are is arbitrary words describing arbitrary levels in the hierarchical composition of a program, just as ``sentence,'' ``paragraph,'' and ``chapter'' are different levels of structure within a book.

The preceding discussion has been in very general terms, describing features common to most ``conventional'' computer languages. If you understand these elements at a relatively abstract level, then learning a new computer language becomes a relatively simple matter of finding out how that language implements each of the elements. (Of course, you can't understand these abstract elements in isolation; it helps to have concrete examples to map them to. If you've never programmed before, most of this section has probably seemed like words without meaning. Don't spend too much time trying to glean all the meaning, but do come back and reread this handout after you've started to learn the details of a particular programming language such as C.)

Finally, there's no need to overdo the abstraction. For the simple programs we'll be writing, in a language like C, the series of calculations and other operations that actually takes place as our program runs is a simpleminded translation (into terms the computer can understand) of the expressions, statements, functions, and other elements of the program. Expressions are evaluated and their results assigned to variables. Statements are executed one after the other, except when the control flow is modified by if/then conditionals and loops. Functions are called to perform subtasks, and return values to their callers, which have been waiting for them.

### Brace Styles in C

[Ron House](http://www.csd.uwo.ca/~jamie/.Refs/.Footnotes/prog-advice.html" \l "house) had this to say about brace styles in C:

vjpachiano@ccgate.dp.beckman.com (Vincent J. Pachiano Jr.) writes:   
<My all time favorite !!!

<for (i = 0; i < 99; i++);

<{

< some\_reptitious\_code\_1;

< some\_reptitious\_code\_2;

<}

<Now scratch your head and ask why the loop doesn't   
<execute 99 times!! Stupid error (Yes), but one of my   
<top 10 favorites!!

So stop using an objectively defective program layout style! Write it as:

for (i = 0; i < 99; i++) {

some\_reptitious\_code\_1;

some\_reptitious\_code\_2;

}





