What Goes Into a Program?

Sam Huffer, 23 April 2017

Firstly, what is a program?

"A program is a file that contains instructions that get the computer to perform a task." (Cain, 2011, p. 6) The computer reads each instruction or statement sequentially, does what it says, then comes back and reads the next instruction, does that, and so on until it reaches the end of the program. However, most programs are a bit more than just a sequence of instructions read and executed one after the other. If that's all a program was, many programs would end up re-using chunks of code again and again, programmers would on occasion have to do a ton of mental acrobatics when accounting for code that could repeat a lot, like when you're navigating back and forth through a menu and each screen has to be redisplayed again and again because the user is indecisive. Purely sequential code to deal with that would give everyone making it a headache, it would be a mess and unnecessarily long, and chances are it would be riddled with problems. To get around that, programmers have created a number of tricks for getting the computer to do exactly what they want without having the program become a nightmare to read and maintain.

Let's talk about data.

Going back to my IT notes from year 10, *data* is raw, unorganised or unprocessed facts. Note that data is not information; that's what you get after you organise and process the data and facts you've got, and maybe throw in a little extra to add to its value. Data is the raw values that you start off with and turn into information later.

Not all data is the same. There are several different *data types* used in programming. You can have *characters* (letters, numbers, and the random symbols used for punctuation and censoring swearing) and *strings* (collections of characters that have been put together; for example, the word "string" is a string). There are also numerical data types such as *integers* (whole numbers; not 4.1, or 3.9, just 4. Keep that decimal point away) and *decimals* (now you can bring that decimal point in; these are all the numbers where you have to have that decimal point if you want to avoid chopping off the end of the number). Then there's *Boolean* data (which deals with values of true and false; no maybe or maybe not, just true or false, yes or no). All of these data types have different uses, and combinations of these are often used in programming.

In addition to these predefined data types, programmers can create additional data types at the start of their code. They tell the computer "hey, here is this new data type you can use in this program, and here are the acceptable values for this data type". For example, you could define a data type "genre" and have listed as acceptable values "pop" (maybe not), "rock", "metal", "classical", etc. Once you've done that, the computer will recognise what these values are and will be able to work with them, and the usage of it throughout your code when appropriate will make that code easier for humans to decipher and interpret, rather than having them argue over which interpretation is right and having an unnecessary number of religions as a consequence.

However, before data can be used in a program, you need to tell the computer to take a chunk of its memory and keep it aside for storing data values. Depending on what you need, you could tell the computer that the chunk of memory is to be used for a single, unchanging value, and thus create a *constant*, or you could tell it that you want to be able to change the value stored at that memory location, and thus create a *variable*. It is important when defining constants and variables in your code to *tell the computer what data type that constant or variable should be*. Otherwise, the program will crash when you try to run it, you'll get a headache and likely shout abuse at the computer for not being psychic.

Most constants and variables only store one data value (e.g. the integer "x" only stores the value "5", or the string "output" only stores the value "hello"). However, programmers can also create and use these magical (not really) things called *arrays* and *records*. These can be used to store multiple related but distinct values in one container.

Think of an array as being like a spreadsheet in Microsoft Excel, sometimes with only a single column, sometimes with multiple columns. Less commonly, you can encounter arrays that are analogous to multiple spreadsheets in a stack, or multiple boxes of stacks of spreadsheets, but let's just worry about single columns or spreadsheets for now. Each *element* in the array is like a single box in a spreadsheet: each box has a different name, or index, and each can hold a separate data value to the previous box or the next box. When assigning values, you can assign a value to one specific element in an array, or a different element, or to every element in the array, overwriting its contents completely. The only limitation is that no matter which element you assign a value to, *all elements in an array take the same type of data*. You can have an array of integers or an array of characters, but you can't have, for example, an array of Boolean values and strings. Again, you'll just make the computer crash, and that's not fun.

Like arrays, records can be used to store multiple values, but they work slightly differently. Instead of viewing records as akin to a spreadsheet, think of them like the forms you have to fill out for the doctor's or school, where they ask your name, date of birth, credit card number, whether you're male or female, etc. You would enter each piece of data into the relevant field, and none of the fields would necessarily ask for the same data type. Records work the same way. *They are a collection of distinct but related fields of (potentially) different data types*. You could even have a field in a record that is an array, or vice versa, an array where each element is a record.

When using constants and variables, there is one thing you must make sure you always do: assign it a value before you use it. If you don't tell the computer "ok, make x equal to 34", it will just look in the memory location set aside for x, and take out whatever junk was left there the last time that memory location was used by another program. For all you know, this could mean x ends up being 89745889878578678623132 42, which, unless you actually need x to equal that exact value, could be very embarrassing when your office starts getting complaints about outrageous invoices due to a computer glitch like that, assuming you can even get the computer to print the invoices without having the program crash on you again.

Controlling a program

Data is all well and good on its own, but it's not particularly helpful unless you use it for something, such as producing information or controlling a program. The latter can be performed through the *conditions* built into code that allows *selection* and *iteration*.

Programmers will often want their code to do different things depending on the value of a variable, like how if it looks like it's going to be sunny, you'll wear a shirt and shorts, while if it's going to rain and be cold, you'll bundle up like an Eskimo. Programmers can replicate these sorts of decisions inside code using special pieces of code that facilitate selection: if statements and case statements.

If statements are pretty straightforward. They follow the logic "if *condition* is true, then do this". For example, if the value of variable x is "5", the computer displays "hello" on your screen. If statements can also be built to be more complex than this, using multiple conditions (e.g. "if x = 5 or x = 6" or "if x = 5 *or* y = 6"), or by getting the computer to execute different chunks of code depending on which conditions hold true (e.g. "if *condition a* is true, do action a, else do action b" or "if *condition a* is true, do action a, else if *condition b* is true, do action b, if *condition c…*").

However, this can get tedious and should be avoided unless necessary. Case statements can help with this. Using a case statement, you can make the computer execute different code based on the value of a single variable (e.g. if x is 1, the computer does one action; if x is 2, the computer does

something else; if x is 3, the computer does something different again, etc.). This can be used to simplify code and make it much more readable and concise than by going "if then else if then else if then else if then else…."

Frequently, you'll want to repeat the use of these selection statements, or chunks of code in general, until a condition is met. The tool for this is a *loop*. A loop allows you to reiterate through a section of code until a condition is met (a *repeat-until* loop), while a condition is true (a *while-do* loop), or a set number of times (a *for-do* loop). Each is more useful than the others under particular circumstances. For example, when you want a section of code to be executed at least once and possibly more times, you would use a repeat-until loop because it checks whether the condition holds true at the end of each iteration. When there are times where you might not want a section of code to be executed even once, but it might need to be executed multiple times, you would use a while-do loop as this checks the condition at the start of the loop before it has a chance to do the wrong thing and make your day bad. When you want to use a while-do loop but can determine in advance how many times you want to go through the loop, such as when you want to change each value in an array and you know the length of the array in advance, you would use a for-do loop and get it to loop through x number of times. The only problem with loops is that, if you're careless, you might get stuck in an infinite loop that will crash the program because just keeps repeating the same instructions again and again and again and again and again and again and again and again and again and again…

Dividing code up

When used well, both iteration and selection-type instructions can improve the efficiency, effectiveness and quality of code and a program immensely. But even when you use both, you can still end up having to reuse chunks of code throughout your program needlessly. However, the good news is that the programmer's toolbox has room for even more tricks than have been discussed thus far…

Meet *procedures*: sections of code that you can call from anywhere else within your program. Think of it like taking a chunk of code, putting it in a box and labelling it, and then when you need your program to use it, you just tell it to go to that box and use what's in it. Procedures are very useful for breaking up your code into small chunks that just do a specific task, making it more readable and easier to maintain, and are helpful when you want to reuse sections of code. Using procedures, you don't have to cut and paste the code for that procedure throughout your program. You just tell the computer "go look at that box over there." The more you can break code up like this, the better, so long as you don't do it to the point of idiocy and end up having to needlessly scroll up and down and up and down again and again to understand what your code is doing. In this process, you can even take some of the code in your program, and put it in another program file that your main program can call and use. This process of *modularisation* is useful when you want to reuse sections of code across different programs. That way, you don't have to reinvent the wheel, you can just get one you prepared earlier out of the shed.

Sometimes your procedure will need to know certain things before it can do what you want it to. For this, you build in some *parameters*, which are pieces of data that you put into the procedure when you call it in your program ("go use what's in that box, and here's some data that you'll need for that"). When calling a procedure that needs data to operate properly, make sure that you do give it to the procedure. Otherwise, it'll just think to itself "REBELLION!!!" and crash on you. But if you make sure to give it all the data it needs, that procedure will be your best friend. Maybe. No? Ok, moving on.

Want to know what's better than a procedure that does something for you? One that can give you something back for your trouble. Procedures that can do something for you and then give you back data are called *functions*. Depending on what it does, it may or may not need some parameters,

but they will always give you something back, like a vending machine that always gives you some snacks once you give it your life's savings of $2.50. Assuming it's not broken though.

And that's it.

Now you know what goes into a program. You know now that a program is comprised of chunks of instructions that get selected for and/or repeated, using data to determine what should happen and what the output of the program should be. You know how programmers break up their code to make it more readable, more manageable, and less error prone, as well as how these problems can arise if the programmer is careless or misses something. To be fair, computers are stupid and will not do what you want them to unless you format your instruction exactly right, with two sugars and no milk. Though I suppose that might come in handy if someone decides to make a killer AI and start the apocalypse, or lock everyone inside of a virtual world. With any luck, their program will be riddled with so many bugs that it fries their computer, while everyone else happily codes their own programs to their heart's content, blissfully ignorant of how close to the end of the world they may or may not be.

References

Cain, A, 2011, 'Programming Arcana', Swinburne University of Technology, viewed 23 April 2017, <https://ilearn.swin.edu.au/bbcswebdav/pid-6305805-dt-content-rid-34412896_2/courses/2017-HS1-COS10009-220415/Content/Programming-Arcana.pdf>