(Approx. 1895 words)

## Evolution of Computer Programming

By Dick Maybach, Brookdale Computer User Group

www.bcug.com

n2nd (at) att.net

The first computers were mechanical or a combination of mechanical and electrical components and typically performed only one task. The Norden bombsight and punched card machines used to process the 1890 census data are well-known examples <http://www.columbia.edu/cu/computinghistory/census-tabulator.html>. These were programmed as they were manufactured.



Punched Card Machine from the 1890 Census.

The first programmable electronic computers were essentially a collection of logic elements and were programmed with patch cords, <https://en.wikipedia.org/wiki/ENIAC>. Unfortunately, the first truly programmable machines were nearly as cumbersome. I wrote the first program in machine code, whose commands were numbers, consisting of 8-bit instructions with an 8-bit address <http://www.mirrorservice.org/sites/www.bitsavers.org/pdf/logisticsResearch/Description_of_Operations_ALWAC_III-E_Electronic_Digital_Computer_1954.pdf>. Two commands were combined into an 8-digit (hexadecimal) number, and programmers worked with long lists of these. The operations were primitive (e.g., add the content of a register to that of the accumulator). There were no subroutines; you worked with one big block of code, and there were no provisions for comments; you annotated your list of numbers with hand-written notes.

The code was stored as ones and zeros (holes and no-holes) on a strip of paper tape. To make changes, you put a paper tape in a reader and printed the hex numbers as you punched a new tape until you reached an area you wanted to change. Then you typed and punched new words onto the output tape. If there was a good section, later on, you moved the input tape to it and restarted punching the data on it to the output. The computer had no operating system; you loaded a very short program (using toggle switches) into memory, which could move the data from paper tape into memory and execute the code.

The next step was programming in assembly language, <http://pages.cpsc.ucalgary.ca/~dsb/PDP11/InsSet.html>. Instead of ones and zeros, instructions were short mnemonics (e.g., STO for store, JMP for jump, GOSUB to call a subroutine). Each command still represented one computer instruction. As with machine code, you worked with memory locations instead of variables. (Instead of adding X and Y, you added whatever was stored in location 54432 to what was stored in a register. It was up to the programmer to keep track of where variables were located.) Assembly language did include provisions for subroutines and mechanisms to handle their arguments. In the early days, there was no operating system, and often you worked with paper tape, but although the development process was better, it was still tedious.

* Read an editing program onto the computer from paper tape (punched cards on a mainframe).
* Compose your software (in RAM), and when complete, write it onto paper tape.
* Read a linking program onto the computer.
* Link the separate code modules and write the result (as a memory image) onto paper tape.
* Read the memory image into the computer and run the program.
* If there are errors, repeat.

Assembly-language code was easier to read as the operation code and operands were mnemonics rather than numbers. In addition, there were provisions for comments, and a large program could be split into separately assembled modules. For example, here is a snippet of assembly-language code, where "INIT" and "START" labels and the text following a semicolon is a comment.

|  |  |  |
| --- | --- | --- |
| INIT | MOV #600, R0 | ; set up source address  |
|  | MOV #prtbuf, R1 | ; set up destination address  |
|  | MOV #76, R2 | ; set up loop count  |
| START | MOVB (R0)+, (R1)+ | ; move one character, increment source and destination |
|  | DEC R2 | ; decrement count by one  |
|  | BNE START | ; loop back if decremented counter is not equal to zero  |
|  | HALT |   |

Fortran, the first popular high-level language, appeared in the 1950s. High-level language instructions are independent of the computer on which they run and require a program to translate them to machine instructions. As a result, they can be moved to any machine (perhaps with some minor editing) for which translation software is available. Rather than dealing with registers and memory addresses, high-level languages deal with variables, arrays, objects, complex arithmetic or boolean expressions, subroutines, functions, and loops, focusing on usability over optimal program efficiency, <https://en.wikipedia.org/wiki/High-level_programming_language>. Unlike low-level assembly languages, high-level languages have few, if any, language elements that translate directly into a machine's native opcodes. Other features, such as string handling routines and file input/output, may also be present. One thing to note about high-level programming languages is that these languages allow the programmer to be detached and separated from the machine.

Here is a simple Fortran program. Lines beginning with exclamation points are comments and are not translated into machine code.

program SimpleAdd

! This program adds two numbers.

 implicit none

! Define x, y, and z as decimal numbers.

 real:: x, y, z

! Add x and y and print the result.

 x = 10.0

 y = 20.0

 z = a + b

 print \*, 'The total = ', z

end program SimpleAdd

Except for the command details (e.g., the two colons following "real" and the asterisk following "print"), the basic functioning of this program is obvious. This doesn't mean that becoming proficient in Fortran is trivial, but that programmers can concentrate on the problem they are solving rather than the operation of the computer. Although they do different things, comparing the Fortran program with the assembly-language one that preceded it makes this obvious. There are dozens of high-level languages in use; Fortran is significant because it was the first popular one and remains popular after almost 70 years, although it has evolved significantly. Many PC users are familiar with BASIC, which is related to but simpler than Fortran.

High-level languages greatly simplified program development, but the result was much more complex software that was difficult to develop and maintain. For example, it was now easy to code mathematical formulas (the name is an abbreviation of "formula translator"), but coding decisions remained problematic. For example, almost every program contains such constructs as "do something and what you do next depends on the result." The code for this was often something like the following.

 IF X < 0, GO TO A

Where A is a label to another program statement (or in BASIC, a line number). As the size of programs and the number of coders working on them grew, this produced chaos. Programmers would try to make sense of their work by drawing arrows from the GOTOs to their destinations and soon began calling the result "spaghetti code." Edgar Dijkstra stated the problem in his classic paper, Go To Statement Considered Harmful, <https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf>.

As a result, most modern languages have eliminated the GOTO statement and statement labels. Instead, you will see variations of such control statements as the following.

 if (condition is true) then {

 lines of code

 {

 else {

 more lines of code

 }

Example conditions are “x > 0”, “x = 4 and y = 6”, and the like. If the condition is true, then all the code between the following curly braces is executed; if the condition is not true, the code between the curly braces runs. There are also loop statements, such as the following.

 While (condition is true) {

 lines of code

 }

When these were used, the results were called "structured programs." The advantages were so great that almost all modern languages use this method, but it has been retrofitted into older ones, such as Fortran and assembly language. Indeed, the technique is more an approach to programming than the particular language or statements used. You can write well-structured software using gotos, but it becomes easier with tools that facilitate it.

The structured design allows a better understanding of how the problem is solved and makes it simpler for a designer to concentrate on it more accurately. It's based on a divide-and-conquer strategy, where a problem is broken into several small problems, which are solved individually. A well-structured design always follows some rules for communication among multiple modules and features cohesion (grouping of functionally related elements) and coupling (communication between different modules).

A later development was Object-oriented programming (OOP) <https://dev.to/charanrajgolla/beginners-guide---object-oriented-programming>, which doesn't replace structured programming but adds to it. It's most easily understood through examples. For example, suppose you were writing software for a car repair shop. Since the objects they deal with are cars, you define a car as the basic object in your program. In OOP jargon, this takes the form of Car class, and objects of this class have properties, such as VIN, owner, license plate, make, model, and year. So, for example, the following statement would create an instance of a Car object with the properties just mentioned.

 a\_car = Car(WP0AB2A83EK194587, Joe Richguy, VROOM, Porsche, 911, 2014)

The class also defines functions that apply only to cars, such as:

* change\_oil(3-11-21, SAE 30),
* update\_odometer(34267), and
* new\_plate(FTP 407).

Since these apply only to the car class, we could reuse the names of functions that performed different actions on different objects, such as integers. (It may help to think of OOP functions as being local to a class, in the same way, that local variables exist only within a subroutine.) So, for example, in Python, you might update the odometer with the following statement.

 a\_car().update\_odometer(34267)

Similarly, you might find the current mileage with this Python command.

 Mileage = a\_car().odometer()

This format makes it clear that these actions apply only to a particular object.

The definitions of these functions are stored in the Car class, while the actual data are stored in each car object. You can view classes as factories that produce objects. Once a programmer has designed the car Class, it generates a car object each time it's called (as in the a\_car example above). There is, of course, more to setting up the software; in particular, we haven't said how we will store and access the car objects. Objects can correspond to real-world items. For example, an online shopping system might have objects, such as a shopping cart, customer, and product. But they can also represent more abstract entities, such as one that represents an open file.

In OOP, functions can behave differently when applied to different objects. You are probably already familiar with this; for example, look at these two Python examples.

A = “abc”

b = “def”

print(a + b) produces “abcdef”

Here, the "+" function concatenates two string objects.

c = 1

d = 2

print(c + d) produces 3

But here, the "+" function adds two integer objects.

The same function (" +" in this case) produces dramatically different results depending on the class of the objects on which it operates. You can appreciate that on a large project with many programmers; this would simplify their work and reduce errors because they don't have to coordinate the naming of functions, although they will have to coordinate the naming of classes.

OOP continues the organizing of code that was begun by structured programming. In the latter, program flow changed from a stream-of-consciousness approach to one that completed each task before undertaking the next one. Thus, for example, in OOP, a class defines the information associated with an object and the functions that manipulate it.

These techniques have accumulated as languages have evolved, with new techniques retrofitted into old ones, either as a revision or an add-on package. Thus, if you revisit a language you learned in the computer dark ages, you may find that it now has relatively modern features.