IMPERATIVE PROGRAMMING

Gordon Pace, Walid Taha
Chalmers Technical University, Department of Computing Science,Sweden.

Keywords: imperative programming, stateful programming, computational effects

Contents

Summary

The vast majority of programs running in the world are written in what is generally known as imperative programming languages. A program written in an imperative language is a sequence of instructions that alter the state of a computer. Affecting the state of a computer is often used to affect the state of the world. For example, changing the state of a particular memory cell in a computer can be made to actuate a motor on a robot's arm.

This article is an introduction to the basics of imperative programming, including the notions of imperative variables, basic types, static data structures and dynamic data structures. The presentation gives examples in various imperative languages, including C, BASIC, Pascal and Ada. In addition to introducing the various features of imperative

programming languages, we touch upon important concepts such as basic software development and engineering concepts, and formal methods for imperative languages. We conclude by briefly outlining some important research directions that are already making their way into the mainstream of imperative programming.

1. Effect: The Essence of Imperative Programming

Today, imperative programming is the most widely used paradigm for building software for digital computers. At the heart of this paradigm is the view that programs are a sequence of instructions or commands that change the state of a digital computer. By employing situation-specific techniques, altering the state of the digital computer can also be made to initiate change in the surroundings of the computer. Examples of such actions include:

- printing a character on a screen (or monitor),
- actuating the arm of a robot,
- turning on a light bulb, or
- correcting the speed of an aeroplane.

Regulating the correspondence between the state of the computer and the surroundings is usually done through a combination of hardware and software (often called device drivers). While detailing the hardware and device driver techniques is beyond the scope of this article, they are generally considered advanced topics, and are not necessary for understanding, using, or writing imperative programs.

1.1 The Simplest Effects: Input/Output

The simplest effects that a program can demonstrate to a user is input/output (IO). The following FORTRAN program (entitled Simple) reads an integer N from user and writes it back to the screen:

```
PROGRAM Simple
READ (*,*) N
M = N*2
WRITE (*,*) M
STOP
END
```

When this program is executed, it first waits for the user to enter an integer. When this integer is entered, it is stored in a memory cell called N. Next, N is multiplied by 2. The result is stored in a memory cell called M. Finally, the contents of M are printed back on the screen.

2. Variables and Assignment

Digital computers contain a fixed number of memory cells (see also the article on hardware). In general, the state of each of these memory cells is exactly one of two states, often called “high” and “low”, or alternatively, simply “1” and “0”. Single memory cells can only be used to represent simple values, such as booleans (“true” and “false”). At a higher level, small aggregates of memory cells are called words. At an even higher level, aggregates of words are used to represent various kinds of data such as integers, floating point numbers, characters, and strings. The state of the computer is the aggregate of the states of all memory cell.

Each storage unit has a unique address. In low level programming languages, these addresses can be used explicitly in programs. There are however many disadvantages to using such locations explicitly (especially that there can be a huge number of storage units on a computer. For example, a PC with a 32MB memory has approximately 32*10^6 different addresses!). For this and other reasons, imperative programming languages generally offer programmers a notion of variables, which makes it possible to name memory locations in a more abstract manner.

2.1 Declaration

Unless a programming language provides means for implicitly declaring variables (as in the case of FORTRAN), variables have to be declared before they are used. Variable declarations usually indicate the type of the information that will be stored in each variable. The type determines how much memory space to reserve for the variable and how the data in that memory region will be interpreted. For example, in the programming language Ada, variables can be declared as follows:

```
Age: Integer;
Name: String (1..10) := “John”;
```

Here Age is declared to be an integer, whereas Name is a string (not exceeding 10 characters). In most languages, declared variables can be given an initial value, as shown in the above example with the variable Name being initialised to the string “John”.

Sometimes it is convenient to name values which may be used but not changed during the execution of the program. Stating that it will have a constant value allows the compiler to create more efficient code. Note that constants only make sense if initialised (to their constant value):

```
DrivingAge: constant Integer := 18;
Pi: constant Float := 3.142;
```

It is generally considered good programming practice to give variables meaningful names that can be understood by a wide audience of readers. In particular, it can be hard to read another person’s code if the name DA was used in the above declaration instead of DrivingAge. Having suggestive variable names helps make code more readable.

In most programming languages, declared variables start off either with a standard initial value (such as 0 for integers) or could have any arbitrary value (possibly left over by previous programs!). In order to store values to variables, and to alter the contents of a variable in general, an assigment statement is used.

2.2 Assignment

Assignments are the principal means by which the computer store is updated., and since imperative programming is all about side-effects on the store, assignments are central to these languages. The following three Ada commands illustrate different assignments that can be made the variable Age:

```
Age := 25;
Age := CurrentYear – 1971;
Age := Age+1;
```

The left hand side of an assignment states the variable whose value will be updated, while the right hand side gives an expression which will be evaluated before the variable is assigned its new value. In the first statement, the value 25 is simply stored in Age. In the second statement, the contents of CurrentYear are first looked up, and then 1971 is subtracted from that value. Finally, the result is stored in Age. In the third line, the content of Age is first looked up, then 1 is added to that content, and it is finally written back into Age. This means that if the value of Age was 42 before the third statement, it will be 43 after that statement.

As we have seen in the first example in this article, there are other ways of altering the contents of variables, such as input commands.

3. Control Structures

As noted earlier, a program is a sequence of instructions. The default order of execution for a sequence of commands is known as sequential execution. To illustrate, if our program consists entirely of assignment and input/output commands, the computer will simply execute these commands in order. However, it is often the case that more control is imposed on the order in which various sequences of commands are executed. Imperative programming languages provide a variety of what is known as control structures so that programmers can write programs where the order of executing the various instructions can depend on the state of the machine.

3.1 Conditionals

Programs frequently have to take actions depending on a condition. This is how the state of the machine will affect the actions that the program will perform. The basic construct to do this is the if-then-else statement. In Ada, this statement has the following general form:

if *condition*
then *true-branch*
else *false-branch*
end if

For example, the following Ada fragement reads an integer, and prints either even or odd depending on the parity of the number:

```
Get(I);
if (I mod 2 = 0)
then Parity := “even”;
else Parity := “odd”;
end if;

Text_IO.Put_Line(“The number you gave me is “& Parity);
```

Another conditional construct is the case statement. Rather than restricting ourselves to two alternatives, a case statement allows us to have as many alternatives as we would like:

```
case expression is
when pattern1 => action1
when pattern2 => action2
…
when patternN => actionN
end case;
```

When a case statement is executed, the expression is evaluated and compared to the list of patterns. The action associated to the first succesfuly matched pattern is executed. The values usually also have to be disjoint, that is, an expression may not match more than one value. However, they need not be a single value as shown in the following example:

```
case FavouriteNumber is
when 13         => Text_IO.Put_Line(“Not superstitious”);
when 0 | 100    => Text_IO.Put_Line(“Interesting choice”);
when 1..12      => Text_IO.Put_Line(“Small number”);
when others     => Text_IO.Put_Line(“That’s a large number”);
end case;
```

Assuming that FavouriteNumber is not less that 0, the above program will output a humorous comment depending on the user’s favourite number. The second value indicates that it will match either the value 0 or 100. The third line matches any number in the range from 1 to 12. The last line matches any value not falling into any of the other categories.

Note that case statement in general cannot be used to replace a nested sequence of if-then-else statements. A case statement can only match values to a simple pattern, and not a collection of conditions.

3.2 Iteration

Repeating the same code does not necessarily have the same effect since the initial state of the machine may be different. Various problems can be solved by repeatedly executing a piece of code until a certain condition is met.

The most common iteration construct is the while loop. In Ada, it has the following general form:

```
while condition loop
loop-body
end loop;
```

This control structure repeats *loop-body* as long as *condition* is true. The *condition* is checked when the while statements start executing, and after completing each pass through the *loop-body*. Note that if the *condition* in a while statement above is never satisfied, the program fails to terminate (often described as "the program goes into an infinite loop"). Accidental non-termination can be a serious problem in applications in safety-critical applications (See also section about non-termination.)

The following Ada program computes the smallest non-negative integer x which satisfies the inequality x*x > x+100:

```
Num := 0;
while (Num * Num <= Num + 100) loop
Num := Num + 1;
end loop;
```

An important special case of while loops is when we simply wish to repeat a sequence of commands a specific number of times. For example, the following code computes 5 * 6 in a naive manner:

```
Counter := 1;
Sum := 0;
While (Counter <= 6) loop
Sum := Sum + 5;
Counter := Counter + 1;
end loop;
```

This occurs frequently enough to warrant having a separate construct called a for statement. In Ada, this kind of statement has the following general form:

```
for counter-variable in startValue .. endValue loop
loop-body;
end loop;
```

Rewriting the simple program above using for statement, we get:

```
Sum := 0;
for Counter in 1..6 loop
Sum := Sum + 5;
end loop;
```

An advantage gained through the use of for-loops is that can often guarantee the termination of the loop.

4. Procedures and Functions

It is desirable to name pieces of code and invoke them required. Structured code is considered to be essential when writing large programs and is discussed in more detail in (section about modular code).
The most basic procedure definition and use mechanism was the GOSUB (go to sub-routine) and RETURN commands (eg in most old variants of BASIC).

```
1000 REM Swap values of x and y
1010 LET tmp=x
1020 LET x=y
1030 LET y=tmp
1040 RETURN
…
7000 GOSUB 1000
```

In the above example, the sub-routine starting at line 1000 (most older versions of BASIC used line numbers to refer to locations in the code) swapped the values of variables x and y and returned control back to where the sub-routine was called from. Note that this mechanism is very limited for a number of reasons:

- There is no way we can call the sub-routine to swap the values of another two variables, despite the fact that the sequence of instructions would be identical (except for the names of the variables)
- It is essential to know which variables the sub-routine uses. Without our remark in line 1000, we would have to read through the code to work this out.
- If our program uses variable tmp, then its value may be corrupted by calling this sub-routine. It is thus essential to know, not only the names of variables read by the sub-routine, but also the ones used locally.

These, together with other drawbacks (that will become evident as we discuss the issue further), make this solution an unattractive one, and most languages now have a more modular way of defining sub-routines – usually called procedures, or functions if they return a value.

-
-
-

Bibliography

Cousot P. (1990). *Methods and logics for proving programs*, Handbook of theoretical computer science: volume B, pages 841--994. MIT Press. [This is a chapter that gives an indepth treatment of the mathematics available today for proving the correctness of programs, especially imperative ones.]

Kernighan, B. W., and Ritchie, D. M. (1988). *The C Programming Language*. Second, Edition, Prentice Hall. [A standard reference on the C programming language.]

Cooper, D., and Clancy, M. (1985). *Oh! Pascal!* Second Edition, Norton. [An introduction to programming in the Pascal programming language]