Structured Programming Guidelines

David Smallberg Luis Ángel Larios Cárdenas

April 16, 2015

1 Flow Charts

A *Flow Chart* is a graphical interpretation to an **algorithm**. It graphically shows the steps or processes we need in order to solve a problem. A flow chart is very important because we can write a program in any programming language by departing form a well-constructed flow chart. If the flow chart is complete and correct, moving on to programming is relatively straightforward.

Flow chart symbols are shown in table 1. They satisfied the "International Organization for Standardization" (ISO) and the "American National Standards Institute" (ANSI) recommendations.

1.1 Rules for Building Flow Charts

Figure 1 represents the stages we must follow during the construction of a flow chart. Our presented symbols, appropriately laid out, allow us to create a flexible, graphical structure to illustrate the steps necessary to reach a specific result.

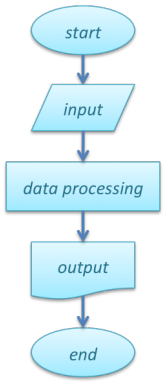


Figure 1: Stages for building a flow chart.

The following is a series of rules to construct flow charts:

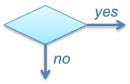
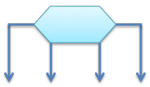
Symbol	Description
	Defines *start* and *end* of the flow chart.
	Data input. Indicates *reading*.
	Represents a *process*. It is used to indicate assignments, arithmetic operations, etc.
	Represents a *decision*. It is used to indicate a condition, and upon its evaluation we follow the appropriate branch.
	Represents a *multiple decision*. It is used to indicate a *selector*, and depending on the selector's value, we follow one of the alternative paths.
	Indicates *printing*, either to screen or to an external device. Represents *writing*.
	Express the flow direction in a flow chart.
	Represents *connection* within the same page.
	Represents *connection* among different pages.
	Indicates a call to a *module* or *sub-routine* within the program. Actually, it represents a sub-problem we must solve in order to continue with the execution of the flow chart.

Table 1: Flow chart symbols

(a) Any flow chart must have a *start* and *end* (see figure 2).

(b) The lines you use to indicate flow direction must be straight, and vertical or horizontal.

(c) All lines used to indicate action flow must be connected. The connection must arrive to symbols such as *input*, *output*, *process*, *decision*, *connection*, or *program end*.

(d) A flow chart must be built from top to bottom, or from left to right.

(e) The notation you use in a flow chart must be independent of any programming language.

Figure 2: Starting and ending a flow chart.

(f) You may add "comments" or "annotations" to complex tasks within the flow chart.

(g) If the flow chart requires more than one page for its construction, use connectors appropriately.

1.2 Flow Chart Examples

Next, we present a few examples so that the reader can familiarize and develop skills to solve problems by using flow charts. Later, we will show how to translate flow charts into `C++`.

Example 1: Simple Sequence of Actions

Write a program where the user inputs two integers A and B, and it computes and writes the result of the following expression:

$$\frac{(A+B)^2}{3}$$

Solution

The flow chart that represents the solution for this problem is shown in figure 3. We can notice the *sequential* nature of the flow chart, indicating how one step takes us to another, until we finally reach the `end` node. Moreover, it is possible to add comments, thus making a flow chart more readable and easier to translate into `C++` in a later task.

Notice, too, that we do not have to stick to any programming language syntax. A flow chart is independent, flexible, and easy to adapt to the programmer's style. For instance, we have written a mathematical expression as `result = ( ( A + B )^2 ) / 3`, which does not rely on any standard convention.

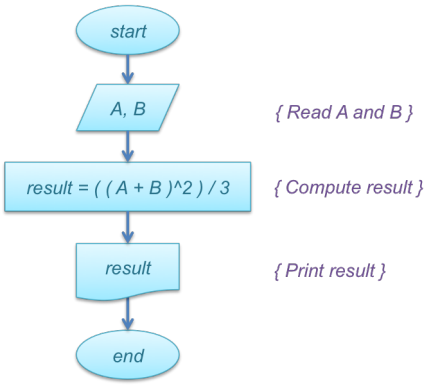


Figure 3: Solution to example 1.

Example 2: Simple Selective Structure

Write a program such that, given a worker's salary, adds 15% to it if his salary is less than $1,000. Print, on this case, the new worker's salary.

Solution

Figure 4 shows the flow chart solution to this problem. Here, we introduce a selective structure: `if - then`.

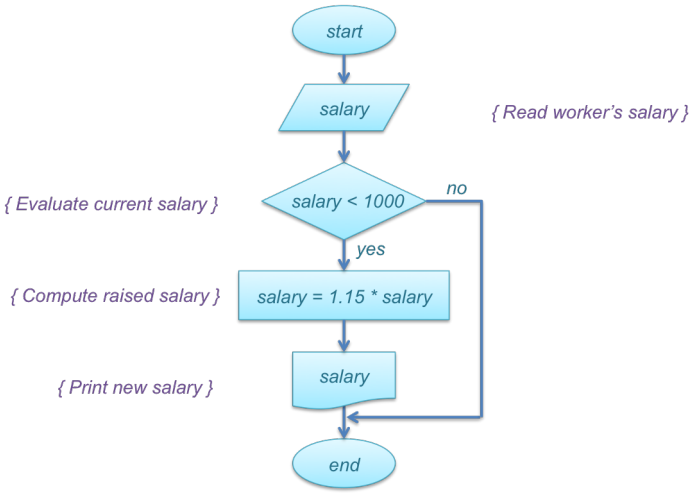


Figure 4: Solution to example 2.

If the worker's salary is less than $1,000, we follow the branch labeled with `yes`, and then we proceed to increase the salary and print it. If the salary is

at least $1,000, we do not have to do anything to the salary, and we simply terminate the program. The latter is indicated with a "bent" arrow labeled `no`, which departs from the condition and connects to the flow exactly before reaching the `end` of the program.

Example 3: Double Selective Structure

Write a program such that given as input a worker's salary, it applies a 15% increment if his salary is less than $1,000 and 12% otherwise. Print the new worker's salary.

Solution

Figure 5 shows a flow chart to solve this problem. We have introduced an `if - then - else` structure. Unlike the solution in figure 4, we now have two branches with *actions* or *processes* that are executed depending on the evaluation of the condition. Observe, however, that both `yes` and `no` paths join in a common point exactly before printing the new salary. Furthermore, we took out the *outputting* of the new salary from the individual `yes` and `no` branches with the purpose of avoiding duplication of the same action. By eliminating redundancy like the one pointed here, our program will be less prone to errors during the translation stage.

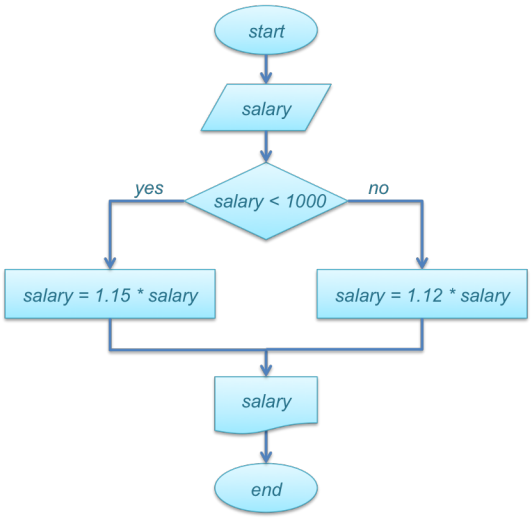


Figure 5: Solution to example 3.

Example 4: Looping – *While*

Write a program that collects (gets as input) the salary of 10 workers and prints the average salary in the group.

Solution

We now need to resort to a different type of structure – a repetitive statement or *loop*. Our first choice of loop is a `while`, which executes a sequence of *actions* as long as a leading *condition* holds. Figure 6 shows the solution to our problem.

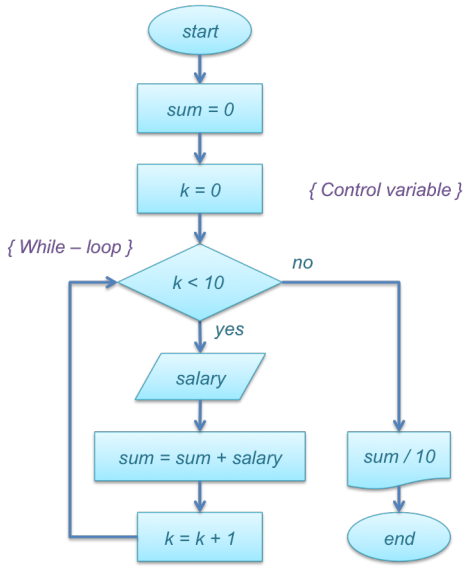


Figure 6: Solution to example 4.

As we can see, when the condition `k < 10` is `true`, we proceed to collect a salary and add it to a cumulative variable `sum` (which was previously initialized to zero). After processing the input `salary` and updating `sum`, a *control* variable is incremented, indicating we are done with the current worker's salary. Thus, the flow goes back to the *loop condition*, we check if indeed `k < 10`, and determine whether we should continue with yet another *iteration* or finish the *while*-loop by following the `no` branch. Finally, upon exhausting the loop, we print the average, and finish the program by reaching the `end` node.

Example 5: Looping – *Do-While*

Write a program that *keeps* reading an integer from the user, until he inputs a number that is multiple of 2.

Solution

Figure 7 presents the solution to this problem. One particular feature in our example is that we are required to keep asking the user for a number, *until* he enters an even integer. In this case, we do not know how many times the user will fail to give a valid input. This setting suggests a loop, where the condition

to break it is a user's input corresponding to a multiple of 2. Moreover, the only certainty we have is that we must read the user's input *at least once*.

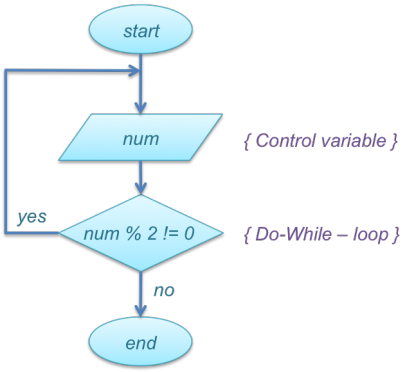


Figure 7: Solution to example 5.

A variation of a loop that fits our requirements is the `do-while` structure. As we can see in figure 7, the body of the loop is executed one or more times, depending on the *control variable*, which, in this case, corresponds to `num` (e.g. an unpredictable user's input). Notice, though, that we have used `C++` notation to represent the *modulo* operation and the *difference* operator with the `%` and `!=` symbols, respectively.

Example 6: Looping – *For*

Write a program that computes and prints the product of the odd integers between 1 and 15, inclusive.

Solution

A `for`-loop is another iterative structure that we can use in problems that require repetition. Figure 8 shows our approach for solving example 6. There, we have introduced a new flow chart symbol that, more or less, summarizes the compact structure of a *for*-loop. In other words, a *while* structure can easily supersede a *for*-loop, but the latter, by nature, is a simplification that moves all looping-control to one location: a statement (one box) with the *initialization*, the *stay-in-loop-condition*, and the *prepare-for-next-iteration* controllers in the same place.

The *for*-loop symbol is a box divided in three areas:

- The **upper left triangle** contains the *initialization* of the control variable (e.g. `k = 1`).
- The **right center triangle** holds the *stay-in-loop-condition* (e.g. `k <= 15`).

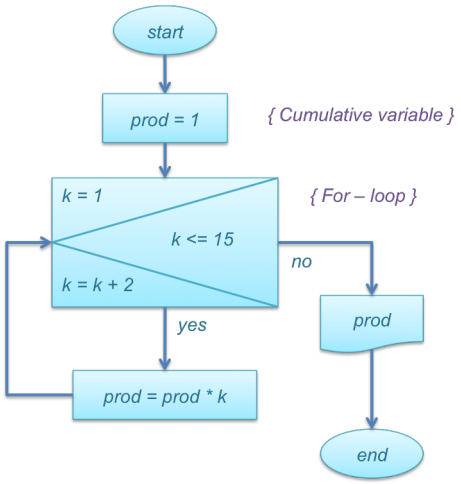


Figure 8: Solution to example 6.

- The **bottom left triangle** bears the *prepare-to-next-iteration* action (e.g. `k = k + 2`).

The *initialization* action is executed *only once*, when we enter the loop. Then, the condition is evaluated. If the condition turns out being `true`, we move on to the actions following the `yes` branch, otherwise, we leave the loop via the `no` path. On the other hand, at the end of each iteration the *prepare-to-next-iteration* statement is executed, and, then, the loop condition is evaluated. If the latter holds, a new iteration is performed, otherwise we finish the *for*-loop by leaving through the `no` branch.

1.3 Exercises

1. The number of sounds that a cricket produces is proportional to the environment temperature. Accordingly, we can determine the environmental temperature by using the cricket as our thermometer. We have approximated the following mathematical relation:

$$T = \frac{N}{4} + 40$$

   where T is the temperature in Fahrenheit degrees, and N is the number of sounds emitted per minute.

   Write a program that computes and prints T after reading N. Your program should not output anything if N is negative (since a cricket cannot produce a *negative* amount of sounds).

2. Given a quadratic equation $ax^2 + bx + c = 0$, write a program to find the real roots by using the expression:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Hint: Real roots are the two x values for which the equation holds. They are real because the argument $(b^2 - 4ac)$ is nonnegative.

3. Write a program that determines and prints whether an input number is positive, negative or zero.

4. Write a program such that, after reading three numbers (in sequence), determines if they are in ascending order (i.e. after reading A, then B, then C, the program should print "Yes" if $A < B < C$). *Hint:* Check the `AND` operator (`&&` in `C++`).

5. Write a program that allows to compute the payment to a worker, taking into account his salary, overtime hours, and category. In order to pay overtime, we consider table 2:

Category	**Payment per overtime hour**
1	\$30
2	\$38
3	\$50
4	\$70

Table 2: Overtime payment for each category

Each worker can have at most 30 overtime hours; if they have more than that, we will only pay them 30 hours. Workers whose category is any other than 1, 2, 3, or 4, are not paid overtime.

6. Write a program such that given 270 integer numbers, it computes the sum of the odd numbers and the average of the even numbers.

7. Write a program that obtains and prints the sum of the terms in the following series:

$$2, 5, 7, 10, 12, 15, 17, ..., 1800$$

8. Write a program that computes and prints the average of several integers. Assume that the last input value is the *sentinel* 9999. A typical input sequence could be:

10 8 11 7 9 9999

9. Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + ...$$

Print a table that shows your approximations to 2, 3, ... 20 terms of the previous series. How many terms did you need to get 3.14159?

2 From Flow Charts to C++

Once that you find yourself comfortable solving problems with flow charts, it is not difficult to move on to coding them in any language. Particularly, we will show a couple of examples using `C++` from problems we described in the previous section.

Since all variables you write in a flow chart are already necessary, they pass directly to `C++` where you only have to take care of declaring them before they are used, and giving them appropriate types.

2.1 Examples

Example 7

Recall figure 6 from example 4. Since we have solved the problem graphically, we do not have to check the problem statement again – everything has been summarized in the flow chart.

```
#include <iostream>
using namespace std;

int main()
{
   double sum = 0.0;          // Accumulates workers' salaries.
   int k = 0;                 // Control variable.

   while( k < 10 )            // While loop.
   {
      cout << "Provide a salary: $";
      double salary;
      cin >> salary;          // Read in salary.
      sum += salary;          // Accumulate salary.

      k++;                    // Update the control variable.
   }

   cout << "Average salary is: $" << sum/10 << endl;
```

```
   return 0;
}
```

Example 8

In example 6 (figure 8) we used the *for*-loop as an alternative to the *while*-loop. The flow chart presented there accurately displayed how we should frame the solution to that problem. Now, it will be easier to translate such diagram into C++ code.

```
#include <iostream>
using namespace std;

int main()
{
   int prod = 1;                        // Variable to hold product.

   for( int k = 1; k <= 15; k += 2 )    // A for loop.
      prod *= k;

   cout << "The product of odd numbers between 1 and 15 is: ";
   cout << prod << endl;

   return 0;
}
```

2.2 Exercises

Translate all of the flow charts you drew in section 1 exercises into C++ code. Check that the logic you developed there is correct by testing your inputs with different values (where possible).

References

[1] H. M. Deitel, and P. J. Deitel. (1995) *Cómo Programar en C/C++*. Segunda Edición. Prentice Hall.

[2] O. Cairó. (2005) *Metodología de la Programación: Algoritmos, Diagramas de Flujo y Programas*. Tercera Edición. Alfaomega Grupo Editor.