How to Write Pseudocode

1 Intro

Pseudocode is an informal tool that you can use to plan out your algorithms. As you begin to write more complex code, it can be hard to keep an entire program in your head before coding it. Think of pseudocode as a step-by-step verbal outline of your code that you can later transcribe into a programming language. It is a combination of human language and programming language: it mimics the syntax of actual computer code, but it is more concerned with readability than with technical specificity.

This will discuss the following major ideas:

1. Understanding Pseudocode
2. Example Pseudocode
3. Standard Pseudocode Procedure
4. Practicing Pseudocode
5. Translating Pseudocode to a Programming

2 Understanding Pseudocode

Pseudocode is a step-by-step verbal outline of your code that you can gradually transcribe into programming language. Many programmers use it to plan out the function of an algorithm before setting themselves to the more technical task of coding. Pseudocode serves as an informal guide, a tool for thinking through program problems, and a communication device that can help you explain your ideas to other people.

Pseudocode is used to show how a computing algorithm should and could work. Coders often use pseudocode as an intermediate step in programming, in between the initial planning stage and the stage of writing actual, executable code. Good pseudocode can become comments in the final program, guiding the programmer in the future when debugging the code, or revising it in the future. Pseudocode can also be useful for:

- Describing how an algorithm should work. Pseudocode can illustrate where a particular construct, mechanism, or technique could or must appear in a program. Senior programmers often use the pseudocode to quickly explain the steps their junior programmers need to follow in order to accomplish a required task.

- Explaining a computing process to less technical people. Computers need a very strict input syntax to run a program, but humans (especially non-programmers) may find it easier to understand a more fluid, subjective language that clearly states the purpose of each line of code.
- Designing code in a collaborative development group. High-level software architects will often include pseudocode into their designs to help solve a complex problem they see their programmers running into. If you are developing a program along with other coders, you may find that pseudocode helps make your intentions clear.

2.1 Remember that pseudocode is subjective and nonstandard

There is no set syntax that you absolutely must use for pseudocode, but it is common professional courtesy to use standard pseudocode structures that other programmers can easily understand. If you are coding a project by yourself, then the most important thing is that the pseudocode helps you structure your thoughts and enact your plan. If you are working with others on a project—whether they are your peers, junior programmers, or non-technical collaborators—it is important to use at least some standard structures so that everyone else can easily understand your intent.

- If you are enrolled in a programming course at a university, a coding camp, or a company, you will likely be tested against a taught pseudocode "standard". This standard often varies between institutions and teachers.
- Clarity is a primary goal of pseudocode, and it may help if you work within accepted programming conventions. As you develop your pseudocode into actual code, you will need to transcribe it into a programming language—so it can help to structure your outline with this in mind.

2.2 Understand algorithms

An algorithm is a procedure for solving a problem in terms of the actions that a program will take and the order in which it will take those actions. An algorithm is merely the sequence of steps taken to solve a problem. The steps are normally *sequence*, *selection*, *iteration*, and a *case-type* statement. In C++ and Java, *sequence* statements are imperatives. The *selection* is the *if-then-else* statement. The *iteration* is satisfied by a number of statements, such as the *while*, *do*, and the *for*. The *case-type* statement is satisfied by the *switch* statement.

2.3 Remember the three basic constructs that control algorithm flow

If you can implement a *sequence* function, a *while* (looping) function, and an *if-then-else* (selection) function, then you have the basic tools that you need to write a "proper" algorithm.[1]

- SEQUENCE is a linear progression where one task is performed sequentially after another. For example:

  READ height of rectangle
  READ width of rectangle
  COMPUTE area as height times width

- WHILE is a loop (repetition) with a simple conditional test at its beginning. The loop is indicated by the keyword WHILE. The loop is entered only if the condition is true. For example:

  WHILE Population < Limit
    Compute Population as Population + Births - Deaths

- IF-THEN-ELSE is a decision (selection) in which a choice is made between two alternative courses of action. A binary choice is indicated by these three keywords: IF, THEN, and ELSE. For example:

  IF HoursWorked > NormalMaximum THEN
    Display overtime message
  ELSE
    Display regular time message

3 Example Pseudocode

Imagine that the program must replace all instances of the word "foo" in a text file. The program will read each line in a file, look for a certain word in each line, and then replace that word. You can see that the steps to be repeated are indented with spaces in the pseudocode, just like it would be ideally in real code. A first draft of the pseudocode might look like this:

```
Open the file.
For each line in the file:
   look for the word,
   remove the characters of that word,
   insert the characters of the new word.
Close the file.
```

3.1 Use pseudocode iteratively: write it once, and then revise it later

One of the strengths of pseudocode is that you can lay out the basics and leave the hard stuff for later. Notice that in the word-replacement example above, there is no detail about how to look for the word. You, the programmer, can re-write the pseudocode to include algorithms for removing the characters of the word, and inserting the characters of the new word. A second draft of the pseudocode may look like this:

```
Open the file.
For each line in the file:
   look for the word by doing this:
      read the character in the line,
      if the character matches, then:
         if all of the following characters match then (there is a true match)
            Remove the characters of that word.
            Insert the characters of the new word.
Close the file.
```

3.2 Use pseudocode to add features

Pseudocode helps programmers think their way through a problem, just like intermediate steps in a math problem. When used properly, pseudocode can make a complicated programming challenge seem simple. You can improve the pseudocode little by little, one step at a time:

Open the file.
Ask the user for the word to be replaced.
Ask the user for the word with which to replace it.
For each line in the file:
  look for the word by doing this:
    read the character in the line,
    if the character matches, then:
      if all of the following characters match then (there is a true match)
        Count the occurrence of that word.
        Remove the characters of that word.
        Insert the characters of the new word.

Display the number of occurrences of the word.
Close the file.

4 Standard Pseudocode Procedures

4.1 Write only one statement per line

Each statement in your pseudocode should express just one action for the computer. In most cases, if the task list is properly drawn, then each task will correspond to one line of pseudocode. Consider writing out your task list, then translating that list into pseudocode, then gradually developing that pseudocode into actual, computer-readable code. [2]

Task list:

```
Read name, hourly rate, hours worked, deduction rate
Perform calculations
gross = hourly rate * hours worked
deduction = gross pay * deduction rate
net pay = gross pay -- deduction
Write name, gross, deduction, net pay
```

Pseudocode:

```
READ name, hourlyRate, hoursWorked, deductionRate
grossPay = hourlyRate * hoursWorked
deduction = grossPay * deductionRate
netPay = grossPay – deduction
WRITE name, grossPay, deduction, netPay
```

4.2 Capitalize the initial keyword of each main direction

In the above example, READ and WRITE are capitalized to indicate that they are the primary functions of the program. Relevant keywords might include: READ, WRITE, IF, ELSE, WHILE, DO. If you want to use ENDIF and ENDWHILE and ENDIF to make sure you can distinguish the ends of those constructs, that is fine.

4.3 Write what you mean, not how to program it

Some programmers write pseudocode like a computer program: they write something like "if a % 2 == 1 then". However, most readers must stop to reason through lines that are so abstractly symbolic. It is easier to understand a verbal line like "if a is odd then". The clearer you are, the more easily people will be able to understand what you mean. [3] You will never use C++ code in place of text.

4.4 Leave nothing to the imagination

Everything that is happening in the process must be described completely. Pseudocode statements are close to simple English statements. Pseudocode does not typically use variables, but instead describes what the program should do with close-to-real-world objects such as account numbers, names, or transaction amounts.

Some examples of valid pseudocode are:

- If the account number and password are valid then display the basic account information.
- Prorate the total cost proportional to the invoice amount for each shipment.

Some examples of invalid pseudocode are:

- let g=54/r (Why: Do not use variables. Instead, you should describe the real-world meaning.)
- do the main processing until it is done (Why: You should be specific about what 'main processing' means, and what qualifies the process as 'done'.)

4.5 Use standard programming structures

Even if there is no standard for pseudocode, it will be easier for other programmers to understand your steps if you use structures from existing (sequential) programming languages. Use terms like *if*, *then*, *while*, *else*, and *for* the same way that you would in your preferred programming language. Consider the following structures:

- if CONDITION then INSTRUCTION – This means that a given instruction will only conducted if a given condition is true. "Instruction", in this case, means a step that the program will perform. "Condition" means that the data must meet a certain set of criteria before the program takes action.
- while CONDITION do INSTRUCTION – This means that the instruction should be repeated again and again until the condition is no longer true.
- do INSTRUCTION while CONDITION – This is very similar to "while CONDITION do INSTRUCTION". In the first case, the condition is checked before the instruction is conducted, but in the second case the instruction will be conducted first. So in the second case, INSTRUCTION will be conducted at least one time.
- for count = NUMBER1 to NUMBER2 do INSTRUCTION – This means that "count", a variable, will be automatically set to NUMBER1. "count" will be increased by one in every step until the value of the variable reaches NUMBER2. You can use any name for the variable that you think fits better than "count".
- function NAME (ARGUMENTS): INSTRUCTION – This means that every time a certain name is used in the code, it is an abbreviation for a certain instruction. "Arguments" are lists of variables that you can use to clarify the instruction.

4.6 Use blocks to structure steps

Blocks are syntactic tools that tie several instructions together into one instruction. You can use blocks to order information (say, steps in Block 1 always come before steps in Block 2)

or simply to wrap information (say, Instruction1 and Instruction2 are thematically related). In general, indent all statements that show "dependency" on another statement.[4]

When using spaces, every instruction of the same block has to start at the same position. Blocks within blocks have more spaces than the parent block. An instruction from a parent block ends the child block, even if there is an instruction later with the same amount of spaces in front.

```
BLOCK1
BLOCK1
  BLOCK2
  BLOCK2
    BLOCK3
  BLOCK2
    BLOCK4
BLOCK1
```

5 Practicing Pseudocode

Start by writing down the purpose of the process. This gives you a way to judge whether the pseudocode is complete: in short, if the pseudocode accomplishes the purpose, it is complete. Continue by writing down the process. If the process is simple, this may take only one pass. Look at what you have written down and ask the following questions:

- Would this pseudocode be understood by someone who is at least somewhat familiar with the process?
- Is the pseudocode written in such a way that it will be easy to translated it into a computing language?
- Does the pseudocode describe the complete process without leaving anything out?
- Is every object name used in the pseudocode clearly understood by the target audience?

5.1 Write initial steps of pseudocode that set the stage for functions

The first parts of a code typically define the variables and other elements that you will use to make the algorithm functional.

- Include dimension variables. Write code that shows how each variable or data element will be used. [5]
- Set up controls. You will need to define your controls in pseudocode–from text and image boxes in object-oriented programming (OOP) languages to basic controls in simpler codes–much as you would in a regular project. [6]

5.2 Write functional pseudocode

Use pseudocode principles to add specific event-driven or object-driven code when you have completed the "setting" for your project. Each line of your code should describe a sequence function, a looping function, a selection function, or another distinct action.

5.3 Add comments, if necessary

In actual computer code, comments serve the role of identifying tasks and parts of the code to the reader. Pseudocode should clearly explain these steps in nearly-plain English, so you will probably not need to use comments until you translate your pseudocode into a programming language.

Many programmers choose to work their pseudocode into the final code in the form of comments. This helps other programmers–who might be collaborating, reviewing, or learning from the code–understand the intention behind each line.

5.4 Read over the finished project for logic and syntax errors

Again, syntax does not have to be exact, but your pseudocode should still make sense. Try to put yourself into the shoes of another person reading this code, and consider whether your directions are as clear as they can be.

- Assess your code modules according to the various elements that they comprise. For example: core operations for the computer include reading or getting information from a file, writing to a file or display on the screen, doing mathematical calculations, valuing data variables, and comparing one or more elements. Each of these has its own place in a computer code project, as well as in the pseudocode that you write to support that project.
- Work specific tasks into the pseudocode. When you have identified each task, represent it with readable pseudocode that, while mimicking the actual code language that you will use, need not exactly follow the computer programming language.

- Make sure all of the applicable elements are in the pseudocode. Even though you may not need some of the more technical items, like variable declarations, you will still need to represent every task and element clearly in the text.

5.5 Review the pseudocode

Once the pseudocode describes the process reasonably well, without gross errors, review it with any other stakeholders in the project. Ask your team members to give you feedback on which parts of the pseudocode need improvement. Descriptions of processes are often incomplete, so this will help fill in the details of the process. If you are working on the code all by yourself, read through what you have written and consider asking someone else to proofread it.

If your team does not approve the pseudocode, rewrite it for clarity. Ask your team what went wrong: are your steps just unclear, or did you forget to include an important piece of the process?

5.6 Save your pseudocode

Once you and your team have approved the pseudocode, save it in an archive. When you write the actual computer-readable code, be sure to include the pseudocode as a comment in the computer code file.

6 Translating Pseudocode to a Programming Language

Trace the pseudocode and understand how it works. The pseudocode presents you an algorithm. For instance, the code might sort an array list in alphabetical order. The basic, "genetic" code will guide you as you build out the algorithm in your preferred programming language.

Write the pseudocode neatly, simply and efficiently. A well-written pseudocode could make the entire algorithm more efficient and error-free when you launch the program.

6.1 Use coding elements that fit your programming language

These elements might include declaration of variables, if statements, and loop statements. Each procedure line can be implemented in many different ways. Your options depend on the level of your programming language.

For example, try to display data for the user to see. You can use a notification window to display the data, or you can use the existing graphical interface on which you are designing.

6.2 Implement then re-trace and compare the actual code to the pseudocode

Make sure that your actual, implemented code follows the instructions laid out by the pseudocode. For example: if the Pseudocode accepts input and output, try all possible inputs, and compare the output of your implemented code to the output computed from the pseudocode. You can ask a fellow programmer to do the tracing or recommend a better method to test your code.

7 Sources and Citations

1. http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html

2. http://faculty.ccri.edu/mkelly/COMI1150/PseudocodeBasics.pdf

3. http://www.bfoit.org/itp/Pseudocode.html

4. http://www.unf.edu/~broggio/cop2221/2221pseu.htm

5. http://www.g-wlearning.com/cad/9781605251615/student/resourceCenter/PDF/DimVar.pdf

6. http://searchsoa.techtarget.com/definition/object-oriented-programming

This document is based on the page found at http://www.wikihow.com/Write-Pseudocode.