PSEUDOCODE and IF/ELSE

CSC100 Introduction to programming in C/C++ (Spring 2025)

Marcus Birkenkrahe

March 10, 2025

README

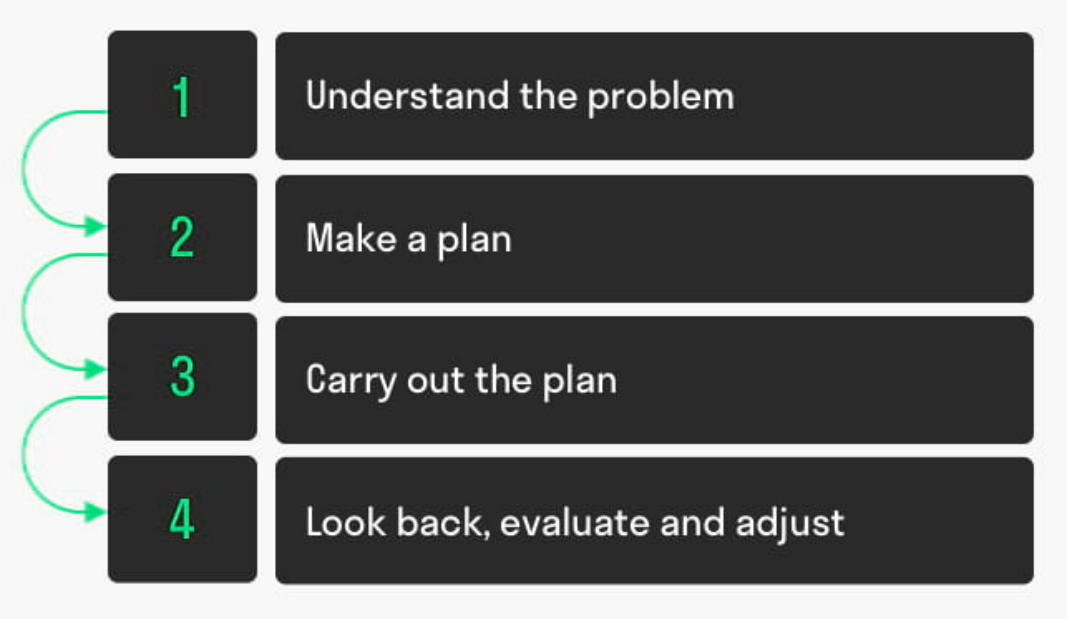
- In this section of the course, we go beyond simple statements and turn to program flow and evaluation of logical conditions
- This section follows chapter 3 in Davenport/Vine (2015) and chapters 4 and 5 in King (2008)
- Practice workbooks, input files and PDF solution files in GitHub

Overview

- **Pseudocode** is a method to write down/analyze an *algorithm* or a *heuristic* without having to bother with *syntax* (like `&i` vs. `i`)
- The prefix pseudo- comes from Ancient Greek , meaning "lying", "false" or "untrue", as in "pseudoscience" or "pseudonym"
- Pseudocode does not need to compile or run so it is closer to a heuristic than to an exact algorithm.
- Code however needs to be exact and is always algorithmic.
- **Always start with pseudocode** before coding, and when you're stuck (not because of syntax ignorance) go back to pseudocode.

Algorithms vs. heuristic - Pólya's problem solving method

- The heuristic method à la George Pólya in four steps:



- See also "How To Solve It" in the chat for more details:

    1. **Understanding the problem:**
        - What is the unknown? What are the data? What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?
        - Draw a figure. Introduce suitable notation
        - Separate the various parts of the condition. Can you write them down?
    2. **Devising a plan:**
        - Have you seen it before? Have you seen the same problem in a slightly different form? Do you know a related problem? Do you know a theorem or a law or a formula that could be useful?

- Look at the unknown: Try to think of a familiar problem having the same or a similar unknown.
- If you have a problem related to yours and solved before: could you use it? Could you use its result? Could you use its method? Should you introduce some auxiliary element in order to make its use possible?
- Could you restate the problem? Could you restate it still differently? Go back to the definitions.
- If you cannot solve the proposed problem try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? Keep only part of the condition, drop the other part: how far is the unknown then determined, how can it vary? Could you derive something useful from the data? Could you think of other data appropriate to determine the unknown? Could you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other?
- Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

3. **Carrying out the plan:** Carrying out your plan of the solution, check each step. Make sure that you understand each step and its consequences. Can you see clearly that the step is correct? Can you prove that it is correct?
4. **Looking back:** Can you check the result? Can you check the argument? Can you derive the result differently? Can you see it at a glance? Can you use the result, or the method, for some other problem?

- You already know that you can (and should) make your programming easier by designing in pseudocode first and re-using it to code.
- This general method is overkill for little problems - but big problems consist of many little problems (and then some).

- Example: Your height in light-years (programming assignment).

    1. **Understand:** Separate problem into **constants** (`SPEED_OF_LIGHT`, `YEAR_IN_SECONDS`) and **variables** (input: `height` in [m] and in [light-years]).
    2. **Plan:** Use formula (distance [m] = c[m/s]* year [s] [ly]).
    3. **Execute**: Compute & use conversion factor on variables heights, and print with the correct formatting specifications (need right **data types**).
    4. **Check**: Do the resulting units of measurement come out right [m]? Does the resulting order of magnitude (E-16) seem plausible?[1]

- **Tip:** Cycle back to old problems/programs that you once found challenging to check if they can be improved with new tools.

Games and Early Computing

- Did you know that games helped shape modern computers?[2]

A Simple Game Problem

Let's look at a small problem from a game:

> When a character's health is 100 or less, drink a health potion.
> If the health is more than 100, go back to fighting.

We need to write this as **pseudocode**, which is like a pretend code that's easy to read and helps us plan before writing real code.

First Try at Pseudocode

Here's a simple way to write this idea in pseudocode:

[1] Light travels 300,000 m/s (E+6). A year has 31,000,000 seconds (E+10). So 1 [m] is approx. E-(6+10) = E-16 light-years.

[2] Dennis Ritchie and Ken Thompson developed the Unix operating system and the programming language C largely because Ritchie wanted to play a game "Space Travels" (like "No Man's Sky", 2016-2025).

```
if health is less than 100
    Drink a health potion
else
    Go back to fighting
endif
```

- This is like giving instructions: "If the health is low (less than 100), drink a potion. Otherwise, keep fighting."
- **Note**: This isn't real code that a computer can run—it's just a plan written in plain words.

Why This Isn't Real Code

The pseudocode above wouldn't work if we tried to run it in a real programming language like C. For example:

- It says **"health is less than 100,"** but in real code, we need to use a symbol like `<` (e.g., `health < 100`).
- **"Drink a health potion"** isn't something a computer understands — we'd need to tell it exactly how to do that in real code.
- What else might a computer complain about if we tried to run this?[3]

Making the Pseudocode Better

Let's fix the pseudocode to be a little closer to real code by using `<` for "less than":

```
if health < 100
    Drink a health potion
else
    Go back to fighting
endif
```

- Now it's clearer: we're checking if `health` is less than 100 using the `<` symbol, which computers understand.

[3] Undeclared variable `health`, missing closure semi-colons after the statements, functions `Drink` and `Resume` not known, and more.

Trying a Different Way

What if we checked the opposite condition? Instead of `health < 100`, we could use `health >= 100` (which means health is 100 or more). Here's how that would look:

```
if health >= 100
    Go back to fighting
else
    Drink a health potion
endif
```

- This does the same thing, but we flipped the order:
  - If health is 100 or more, we fight.
  - If health is less than 100, we drink a potion.
- Both ways work, but the first way (`health < 100`) matches the problem's words more closely, which is usually better when you're starting out.

A Tip for Beginning Pseudocode Speakers

When you're turning a problem into pseudocode (or real code), try to:

- Use words and ideas that match the problem description as closely as possible.
- If you're not sure what the problem means, ask someone to explain it better (remember that 'understanding' always comes first).
- Unfortunately, trying to understand something can (and perhaps should make your brain hurt and make you feel bad for a bit).

Questions to Think About

Using the pseudocode examples above, let's think about these ideas:

1. **Words**: What words in the pseudocode match the problem description?

2. **Logic**: How does the pseudocode decide what to do (like choosing between drinking a potion or fighting)?
3. **Style**: Does the pseudocode sound clear and simple, like the problem description?
4. **Anything Else**: Are there other details in the problem we might need to think about?

Practice: Turn a Problem into Pseudocode and Code

Step 1: Understand the Problem

Here's a problem about a game:

> "Allow a player to deposit or withdraw gold from a game bank account."

Step 2: Write Pseudocode as Comments

- Open `https://onecompiler.com/c`.
- You'll write your pseudocode as comments using '//' in the code editor.
- Pseudocode is a simple way to plan your code using plain words. For this problem, think about:
  - What choices does the player have? (Deposit or withdraw.)
  - How can you write this as an "if-else" decision?
- The pseudocode for this problem could look like this:

```
// If the player chooses to deposit, add gold to the account
// Else if the player chooses to withdraw, subtract gold from the account
```

Step 3: Write Code

- The pseudocode contains a decision and an action. This pseudocode is closer to the C programming language:

```
// If (deposit chosen)
      // add gold to the account
// Else if (withdrawal chosen)
      // subtract gold from the account
```

- Open this code fragment in your browser as starter code: `https://onecompiler.com/c/43bc9bmaa`
- Complete the '...' so that the program runs with two numbers as input: 1 or 2 for the `choice`, and an integer for the `amount`.
- Run the code and test it:
  1. Try 'choice = 1' and 'amount = 50' to deposit.
  2. Try 'choice = 2' and 'amount = 30' to withdraw.
- What kind of **error handling** might be necessary?
  1. Checking for negative amounts (`amount < 0`).
  2. Ensuring enough `gold` for withdrawal (`gold >= amount`).
  3. Handling invalid choices (`choice NOT 1 OR choice NOT 2`).
  4. Dealing with input mistakes (`choice` or `amount` NOT a number).
  5. Handling maximum `gold` limits (`gold + amount > 10000`).
  6. Dealing with input overflow (`amount > 100000000`).

Solution

Code: In OneCompiler

```
#include <stdio.h>

int main() {
  // variables
  int gold = 100; // Example: starting gold amount
  int choice;     // To store the player's choice (1 for deposit, 2 for withdraw)
  int amount;     // Amount to deposit or withdraw

  // Print balance and ask for action
  printf("Your gold: %d\n", gold);
  printf("Choose an action (1 to deposit, 2 to withdraw):\n");
  // Get action (print confirmation)
  scanf("%d", &choice);
  printf("Your choice: %d\n", choice);
  // Get amount (print confirmation)
```

```
  printf("Enter amount:\n");
  scanf("%d", &amount);
  printf("Your amount: %d\n", amount);

  // If (deposit chosen)
  if (choice == 1) {
    // add gold to the account
    gold = gold + amount;
    printf("Deposited %d gold. New balance: %d\n", amount, gold);
  } else {   // Else if (withdrawal chosen)
    // subtract gold from the account
    gold = gold - amount;
    printf("Withdrew %d gold. New balance: %d\n", amount, gold);
  }

  return 0;
}
```

```
Your gold: 100
Choose an action (1 to deposit, 2 to withdraw):
Your choice: 2
Enter amount:
Your amount: 200
Withdrew 200 gold. New balance: -100
```

Practice: Modify solution to handle errors

- First copy your entire program into a new program (so as not to spoil your nice solution).
- Now modify your copy to check if there is enough `gold` for a withdrawal (`choice == 2`).
- You have to
    1. find the right place in the program for the check
    2. use an `if` statement that tests the error condition
    3. print an error message as an action with `puts`
    4. `return` with `EXIT_FAILURE` (`#include <stdlib.h>`)

```
    // if amount exceeds gold
        // print error message
```

- Run the code with choice `2` and the withdrawal `amount` of 200 to see if your error message is triggered or not.
- To think about until next time: Can this program be improved?

Solution

Code: In OneCompiler

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  // variables
  int gold = 100; // Example: starting gold amount
  int choice;     // To store the player's choice (1 for deposit, 2 for withdraw)
  int amount;     // Amount to deposit or withdraw

  // Print balance and ask for action
  printf("Your gold: %d\n", gold);
  printf("Choose an action (1 to deposit, 2 to withdraw):\n");
  // Get action (print confirmation)
  scanf("%d", &choice);
  printf("Your choice: %d\n", choice);
  // Get amount (print confirmation)
  printf("Enter amount:\n");
  scanf("%d", &amount);
  printf("Your amount: %d\n", amount);

  // if amount exceeds gold
  if (amount > gold && choice == 2) {
    // print error message
    puts("Error: Withdrawal amount too large.");
    return EXIT_FAILURE;
  }

  // If (deposit chosen)
  if (choice == 1) {
```

```
    // add gold to the account
    gold = gold + amount;
    printf("Deposited %d gold. New balance: %d\n", amount, gold);
  } else {   // Else if (withdrawal chosen)
    // subtract gold from the account
    gold = gold - amount;
    printf("Withdrew %d gold. New balance: %d\n", amount, gold);
  }

  return 0;
}
```

Solution: Improvement

1. Move the error handling inside of the `choice == 2` block.
2. Now you can save one part of the condition check.

Code: In OneCompiler

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  // variables
  int gold = 100; // Example: starting gold amount
  int choice;     // To store the player's choice (1 for deposit, 2 for withdraw)
  int amount;     // Amount to deposit or withdraw

  // Print balance and ask for action
  printf("Your gold: %d\n", gold);
  printf("Choose an action (1 to deposit, 2 to withdraw):\n");
  // Get action (print confirmation)
  scanf("%d", &choice);
  printf("Your choice: %d\n", choice);
  // Get amount (print confirmation)
  printf("Enter amount:\n");
  scanf("%d", &amount);
  printf("Your amount: %d\n", amount);

  // If (deposit chosen)
```

```
  if (choice == 1) {
    // add gold to the account
    gold = gold + amount;
    printf("Deposited %d gold. New balance: %d\n", amount, gold);
  } else {   // Else if (withdrawal chosen)
    // if amount exceeds gold
    if (amount > gold) {
      // print error message
      printf("Error: Withdrawal %d too large. Only %d in account.\n", amount, gold);
      return EXIT_FAILURE;
    }
    // subtract gold from the account
    gold = gold - amount;
    printf("Withdrew %d gold. New balance: %d\n", amount, gold);
  }

  return 0;
}
```