A Review of Problem Solving Skills

Outline

1. Input and Output
2. Variables and Data Type
3. Work Flow of Tasks
4. Possibilities and Choices
5. Repetition and Generalization
6. Improving the Program

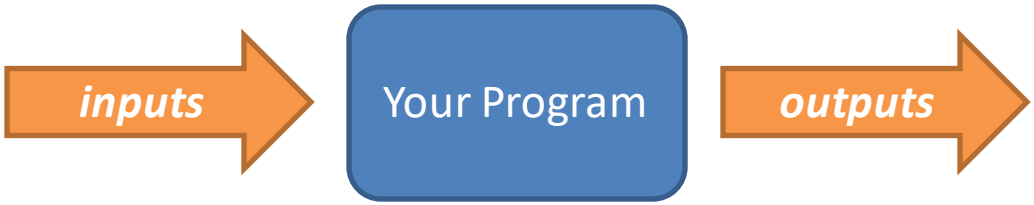
Problem Solving

- There is always a reason for us to write a C program
  - To solve a problem!
- But what are the **necessary steps** when planning for or improving a program?

- You have learnt these implicit skills in every lecture, but let's have a review

1. Input and Output

- Looking at a problem, you have to determine what the input into the system, and the output from the system are



- Objective: ask user for two numbers, and display the multiplication result of them
  - ***Input***: two numbers (integer? floating point? +ve/-ve?)
  - ***Output***: a number (how to display it? formating requirements?)

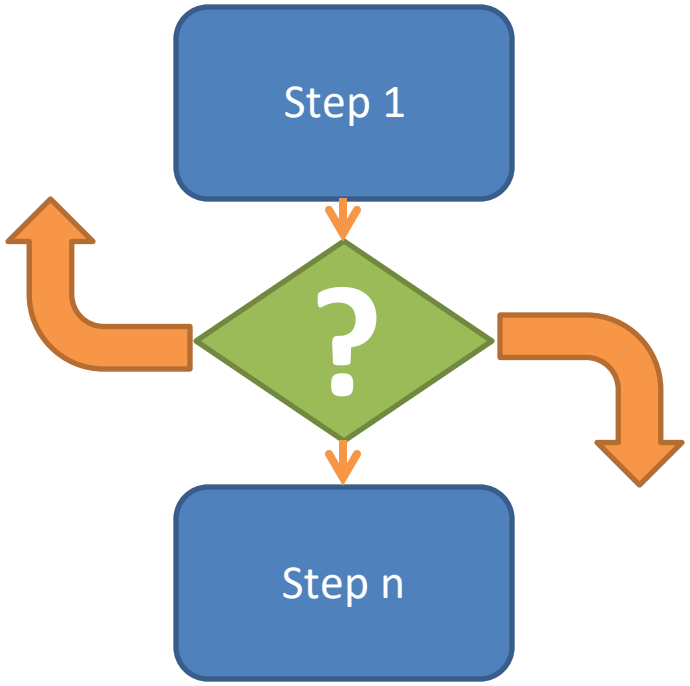
2. Variables and Data Type

- There are often requirements for the inputs and outputs, and you must store them properly for processing

- What are the appropriate data type that could produce the more precise and concise results?
  - *Numbers* vs *characters*
  - *Integers* vs *floating point numbers*
  - *Arrays* vs *single variables*
  - *…*

3. Work Flow of Tasks

- Usually there are a few steps in the problem, e.g.
  1. Get inputs from the user
  2. Process them with some formula
  3. Show outputs onto the screen



- Does this fit well with the current problem to solve?
- Are there any repeated steps?
- Are there any branched steps?

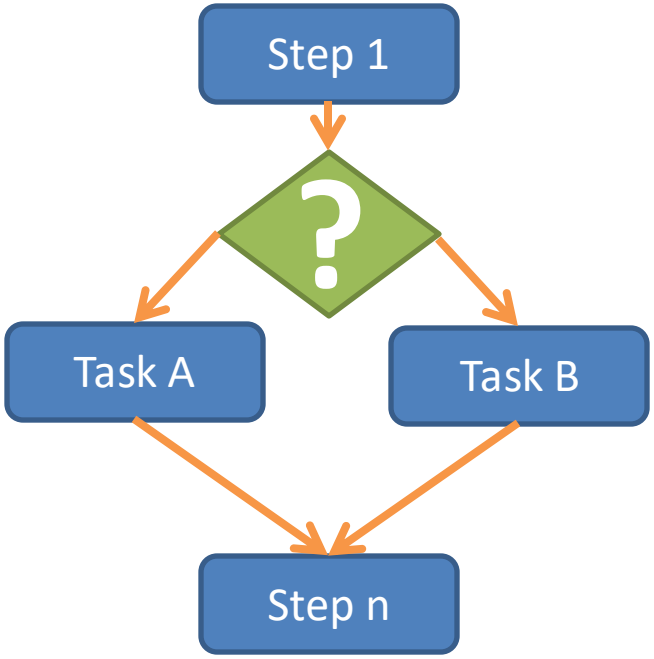
4. Possibilities and Choices

- There may be different tasks, depending on some conditions, e.g.
  - If the user inputs 'x', then do task x, otherwise do task y
  - If the number is larger than 0, then do task z, otherwise nothing happens

- What are the choices users can make?
  - E.g. CC canteen, NA canteen, MorningSide canteen, …
- What are the possibilities you need to face?
  - E.g. either odd or even numbers

4.1. Mutual Exclusion of Cases

- In a usual scenario, the cases are ***mutually exclusive***, i.e. only either of two (or more) situations should happen but not both, e.g.
  - If you eat at Chung Chi canteen for lunch, you will not have another lunch at New Asia canteen
  - If you are having an ENGG1110 class now, you should not be having another lesson at the same time
- How to use the if-else building blocks properly?
- It is still possible to see multiple tasks that are sequential but not exclusive
- What is the appropriate syntax?



5. Repetition and Generalization

- Computers are good at ***tedious and repetitive*** tasks, e.g.
  - Consider 100 integers and 200 doubles
  - Keep doing revision while you haven't understood all lectures
  - Do this task for 50 times
- Looking at a problem, you need to find out what the ***repeating patterns*** are
  - A number of *variables* that are very similar
    - You may be using *arrays*
  - A number of *tasks* that needs to be done again and again
    - You may be using *loops*
- ***Generalize*** your code for these blocks

5. Repetition and Generalization

- Generalizing into arrays
  - E.g.: Store for the number of students for 10 classes
    - An integer (no half students!) array of size 10 is appropriate

```
int studentsno[10];
```

- Generalizing into loops
  - E.g.1: Print 'A' for line 1, 'B' for line 2, … until line 10
    - We need a loop with counter of 1 ➔ 10, and convert that into 'A' ➔ 'J'

```
i=1;
while (i<=10) {
    printf("%d\n",'A'+i-1);
    i++;
}
```

  - E.g.2: Ask user to guess a number, until it matches with our secret
    - We need a loop with condition check

```
while (guess <> secret) …
```

5. Repetition and Generalization

- Looping with arrays
  - Since loops can generate numbers (➔array indices) easily, we often use a loop with counter to go through all elements in an array
    - E.g. Print all the contents of int array x of size 20

```
while (…) { // this should be a loop with counter
    printf("%d\n", x[…]);
    …
}
```

5.1. Looping Conditions and Ranges

- Looping should not go on forever, so the condition for the loop ***must change*** at some point, e.g.
  - Study while today is not Sunday

    →Day will eventually become Sunday
  - Eat while you are hungry

    →Your stomach will eventually become full
- Consider carefully to set an appropriate stopping condition
  - It can be a user input, or a counting variable (counter)
  - Make sure that this terminating point will arrive eventually

6. Improving the Program

- In the planning stage, you may draw a flow chart to represent your ideas
  - Considering inputs/outputs, order of tasks, branching and looping tasks
- Then ***pseudocode*** would be helpful as an intermediate step before actual C program code
  - Write down the tasks in text
  - ***Dry run*** your pseudocode to verify its correctness
- Translate your thoughts into C program code
  - Dry run when you are writing each statements
    - Ask yourself, "What would happen after this step?"

```
// declare variables
// ask user for input
// add to sum
// ask again and repeat
// find average with sum
// display average
```

6. Improving the Program

- You don't need to get frustrated for an "incorrect" program in the first round
  1. Observe carefully the error messages from the compiler, or the wrong outputs
  2. Ask yourself,
     "Is it a problem of the program ***logic***, or program ***syntax***?"
  3. Use proper debugging techniques to tackle issues
     - Print out variables after intermediates steps
     - Comment out part of your code to ensure partial correctness
     - ***Narrow down the scope and identify the problems one by one!***

Summary

- Attention to details from planning to execution
  - Input/output
  - Variable and data type
  - Work flow
  - Possibilities, choices, cases
  - Generalizing for repetitions
  - Improving and debugging
- Good luck!