CS 101: Computer Programming and Utilization

Jan-Apr 2016

Uday Khedker
(cs101@cse.iitb.ac.in)

Lecture 5: Program Design

About These Slides

- Based on Chapter 3 and 4 of the book
  An Introduction to Programming Through C++
  by Abhiram Ranade (Tata McGraw Hill, 2014)

- Original slides by Abhiram Ranade
  - First update by Varsha Apte
  - Second update by Uday Khedker
  - Third update by Sunita Sarawagi

A Program Design Example

How To Write Programs

So far, we wrote very simple programs

Simple programs can be written intuitively

Even slightly complex programs should be written with some care and planning

You must try to ensure that your program works correctly no matter what input is given to it

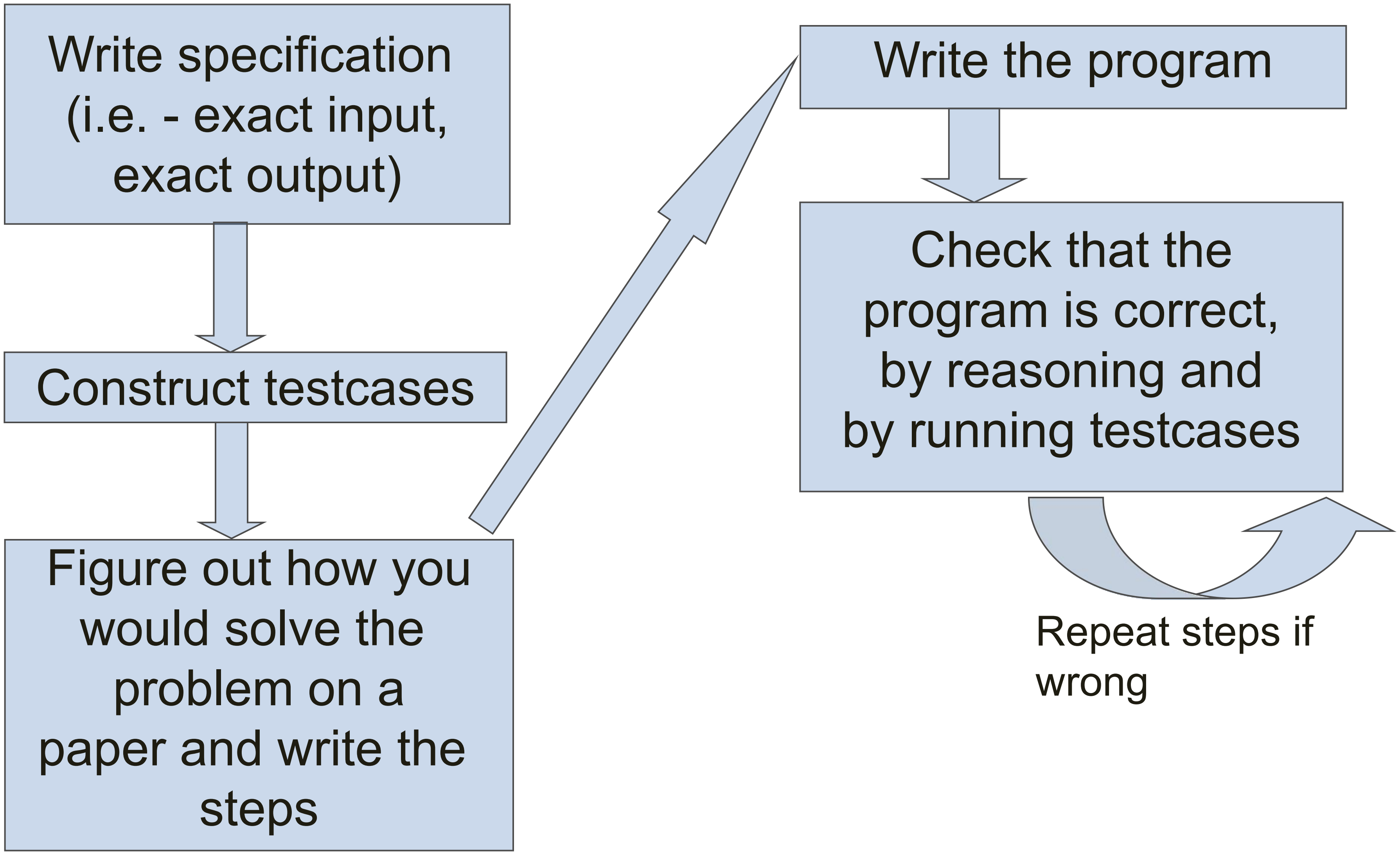
This is tricky even for slightly complex programs

As a professional programmer, you must remember that an incorrect program could cause a plane to crash, an X-ray machine to supply the wrong amount of radiation: your program may be controlling such devices

Program Development Strategy

1. Writing specification
2. Constructing test cases
3. Thinking how to solve the problem on pencil and paper
4. Writing out your ideas formally and making a plan
5. Writing the program
6. Checking mentally if your program is following your plan, or if you made a mistake in writing the program
7. Running the test cases
8. Redoing steps if some test cases fail

Program Development Strategy



Write specification (i.e. - exact input, exact output)
Construct testcases
Figure out how you would solve the problem on a paper and write the steps
Write the program
Check that the program is correct, by reasoning and by running testcases
Repeat steps if wrong

The Problem

The following series approaches e as n increases:

$$e = 1/0! + 1/1! + 1/2! + \dots + 1/n!$$

Write a program which takes n as input and prints the sum of the above series

The Specification

- Usually, the problem will be specified in real life terms, where there may be some ambiguity, or possibilities of confusion. So it is desirable to write to write down what is given and what is needed very precisely

- Specification: A statement of what is the input and the corresponding output. Clear description of when the output is to be considered correct

The Specification For Our Problem

Input: an integer n, where $n \geq 0$

Output: The sum $1/0! + \ldots + 1/n!$

- This is simple enough, but note that we have made explicit that n cannot be a negative number
- Also, it is worth reading this carefully yourself and asking, can something be misunderstood in this?
- You may realize that carelessly, you may think of n as also being the number of terms to be added up.
- The number of terms being added together is n+1.
- The number of additions is indeed n, however

Constructing Test Cases

- Write down some specific input values, and the corresponding expected output values
- This will help ensure that you understand the problem and cross-check the specification you wrote
- 3 test cases are enough for this simple problem
  - For n=0, clearly the answer must be 1
  - For n=1, answer = 1+1/1! = 2
  - For n=2, answer = 1+1/1!+1/2! = 2.5
  - We can put the test cases into a table:

Input (n)	0	1	2
Output	1	2	2.5

Designing the Algorithm (1)

Solving the problem by pencil and paper

- Calculate the first term, 1/0!, which is just 1
- Calculate the second term, 1/1! which is just 1. Add to 1
- Calculate the third term, 1/2!, add to sum so far
- Calculate the fourth term 1/3! …

Now, you can calculate the fourth term by observing that it is just the third term multiplied by 1/3:

- 1/3! = 1/2! * 1/3

This idea will save work in your program too

But you need to find the general pattern, which is:

- 1/t! = 1/(t-1)! * 1/t

So now you can think of a program

What Variables To Use

- When we solve on paper, we write many numbers; we do not need separate variables to store them
- As you calculate on paper, identify the numbers that are reused. These must be stored in a variable. Usually these will be few
- We need to keep track of the sum, so clearly we need a variable for it: let us call it result
- We generate the tth term from the t-1th. So we need to remember the previous term. Store it as variable term
- According to our general pattern, we also need to remember t, so we will have a variable i for that

A Program Sketch

There are (n+1) terms

We need to perform n additions. Clearly we should have a loop for that

So our program should have the following form

```
main_program{
    int n; cin >> n;
    double i = …, term = …, result = …;
    repeat(n){
        …
    }
    cout << result << endl;
}
```

Filling in the Details (1)

- If n is given as 0, then the loop does not execute even once, and the result is printed
  - The value that is printed is the value we initialize result with
  - Since we want 1 to be printed, we must initialize result = 1

Filling in the Details (2)

- We next decide what values (i, term) should have when we enter the loop for the tth time, where t=1, 2, …, n
- In the loop iterations the terms 1/1!, 1/2!, 1/3!....1/n! need to get added one by one into the variable result
- We can do this in the following way. When we enter the loop the tth time
  - i has the value t-1
  - term has the value 1/(t-1)! i.e. the value of the previous term added
  - result has the sum till 1/(t-1)!

Filling in the Details (3)

- So on entering for the first time, i.e. when t=1:
  - i should have the value t-1 = 0
  - term must have the value (t-1)!=1
- Thus before the loop we must initialize
  - i=0; term=1;
- Inside the loop we have to add the next term to result. But i and term holds the previous values
  - So the first statement in the loop should be:
  - i = i+ 1;
- i now has the value t. So Next statement is:
  - term = term/i
- Now we have to add this into result. So we have:
  - result = result + term
- Now result has the sum upto 1/t!, so tth iteration is complete, and coding is done

The Final Code

```
main_program{
    int n; cin >> n;
    int i=0;
    double term = 1, result = 1;
    repeat(n)  {              // On entry for tth time, t=1..n
                                 // i=t-1, term=1/(t-1)!
                                 // result =1/0!+..+1/(t-1)!
      i = i + 1;                // now i = t
      term = term/i;               // now term = 1/t!
      result = result + term; // now result =1+..+1/t!
   }
    cout << result << endl;
}
```

Code Review

It is useful to go over the code again to see that the values of the variables indeed satisfy what we say about them

Specially check: will the values of the variables agree with what we say about them on the t+1th iteration?

Testing

- Next, compile and run the program for the test cases you generated.
- Check if the program output agrees with what was in the table.
- If the program does not agree, you are said to have a *bug*.
- Now you must remove the bug, or *debug*.

Debugging

- Simplest strategy: print intermediate results.

```
main_program{
   int n; cin >> n;
   int i=0, term = 1, result = 1;
   repeat(n){// On tth entry, t=1..n
            // i=t, term=1/(t-1)!
            // result =1/0!+..+1/(t-1)!
     term = term / i
     result = result + term;
                 // now result =1+..+1/t!
     i = i + 1;
     cout << i <<‘ ‘<< term <<’ ‘<< result << endl;
   }
   cout << result << endl;
}
```

- From the printed values you should know what is going wrong.

Concluding Remarks

- There are many, many ways to write a program.
- Most of them will have very similar statements, e.g. i=i+1; term=term/i; which may appear in different orders
- Correctness requires the order to be right, and the statement to be exactly right, i.e. cannot have term=term/i if term=term/(i+1) is needed
- Having a plan and sticking to it is useful
- The plan must be stated as comments in code
- The input output test cases must be constructed and also be written down, as a part of the code, or elsewhere
- Professional programs require all of the above and more as due dilligence

Concluding Remarks 2

How you solve a problem on a computer is often similar to how you solve it by hand

If a certain trick helps you save manual work, it may help on a computer too

Finding the general pattern is very important

You may not deduce all the variables needed right at the beginning, or may discover that the plan you formed does not work. So do add more variables, or revise the plan. But have a plan at all times