

# CS 101: Computer Programming and Utilization

Jul-Nov 2016

Bernard Menezes  
([cs101@cse.iitb.ac.in](mailto:cs101@cse.iitb.ac.in))

Lecture 4: Programming Idioms and 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

# Recall

- How to declare variables of basic types
- How to read numbers into the variables from the keyboard
- How to perform arithmetic
- How to print numbers on the screen

# Continuing ...

- More about arithmetic
- Updating the values in variables
- Basic idioms of repeated computations

# Variables and Memory

- A variable has an address in the memory
- Memory is not homogenous
  - Main Memory (on the same board, different chip)
  - Cache Memory (in the same processor)
  - Registers (in the Arithmetic Unit)
- The value of a variable may also be stored temporarily in other memories
  - Much like permanent address and hostel address
- The mapping from a variable to value is unaffected by its multiple host locations and their updates

# Integer Division

```
int x=2, y=3, p=4, q=5, u;  
u = x/y + p/q;  
cout << p/y;
```

- $x/y$  : both are `int`. So truncation. Hence 0
- $p/q$  : similarly 0
- $p/y$  :  $4/3$  after truncation will be 1
- So the output is 1

# More Examples of Division

```
int noosides=100, i_angle1, i_angle2;  
i_angle1 = 360/noosides + 0.1;           // 3  
i_angle2 = 360.0/noosides + 0.1         // 3
```

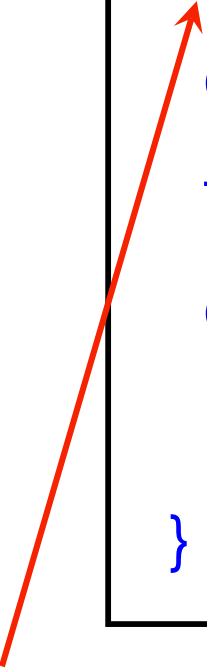
```
float f_angle1, f_angle2;  
f_angle1 = 360/noosides + 0.1;           // 3.1  
f_angle2 = 360.0/noosides + 0.1         // 3.7
```





# Program Example

```
main_program{  
    double centigrade, fahrenheit;  
    cout << "Give temperature in Centigrade: ";  
    cin >> centigrade;  
    fahrenheit = centigrade * 9 / 5 + 32;  
    cout << "In Fahrenheit: " << fahrenheit  
        << endl; // newline  
}
```



Prompting for input is meaningless in Prutor because it is non-interactive

# Re-Assignment

- Same variable can be assigned a value again
- When a variable appears in a statement, its value at the time of the execution of the statement gets used

```
int p=3, q=4, r;  
r = p + q;           // 7 stored into r  
cout << r << endl; // 7 printed as the value of r  
r = p * q;           // 12 stored into r (could be its  
                    // temporary location)  
cout << r << endl; // 12 printed as the value of r
```

# An Interesting Re-Assignment

```
int p=12;  
p = p+1;
```

See it as:  $p \leftarrow p+1;$  // Let p *become* p+1

Rule for evaluation:

- FIRST evaluate the RHS and THEN store the result into the LHS variable
- So 1 is added to 12, the value of p
- The result, 13, is then stored in p
- Thus p *finally becomes* 13

$p = p + 1$  is nonsensical in mathematics

“=” in C++ is different from “=” in mathematics

# Repeat And Reassignment

```
main_program{  
    int i=1;  
    repeat(10){  
        cout << i << endl;  
        i = i + 1;  
    }  
}
```

This program will print 1, 2,..., 10 on separate lines

# Another Idiom: Accumulation

```
main_program{
    int term, s = 0;
    repeat(10){
        cin >> term;
        s = s + term;
    }
    cout << s << endl;
}
```

- Values read are accumulated into `s`
- Accumulation happens here using `+`
- We could use other operators too

# Fundamental idiom

## Sequence generation

- Can you make  $i$  take values 1, 3, 5, 7, ...?
- Can you make  $i$  take values 1, 2, 4, 8, 16, ...?
- Both can be done by making slight modifications to previous program.

# Composing The Two Idioms

Write a program to calculate  $n!$  given  $n$ .

```
main_program{  
  int n, nfac=1, i=1;  
  cin >> n;  
  repeat(n){  
    nfac = nfac * i;  
    i = i + 1;  
  }  
  cout << nfac << endl;  
}
```



Accummulation idiom



Sequence idiom

# Some Additional Operators

- The fragment  $i = i + 1$  is required very frequently, and so can be abbreviated as  $i++$

$++$  : increment operator. Unary

- Similarly we may write  $j--$  which means  $j = j - 1$

$--$  : decrement operator. Unary



# Intricacies Of ++ and --

++ and -- can be written after the variable, and this also cause the variable to increment or decrement

```
int i=5, j=6;
```

```
++i; --j;           // i becomes 6 and j becomes 5
```

++ and -- can be put inside expressions but not recommended in good programming

# Finding Remainder

- $x \% y$  computes the remainder of dividing  $x$  by  $y$
- Both  $x$  and  $y$  must be integer expressions
- Example

```
int n=12345678, d0, d1;  
d0 = n % 10;           // 8  
d1 = (n / 10) % 10;   // 7
```

$d0$  will equal 8 (the least significant digit of  $n$ )

$d1$  will equal 7 (the second least significant digit of  $n$ )

# Compound Assignment

The fragments of the form `sum = sum + expression` occur frequently, and hence they can be shortened to `sum += expression`

Likewise you may have `*=`, `-=`, ...

Example

```
int x=5, y=6, z=7, w=8;
```

```
x += z; // x becomes x+z = 12
```

```
y *= z+w; // y becomes y*(z+w) = 90
```

# Concluding Remarks

- Variables are regions of memory which can store values
- Variables have a type, as decided at the time of creation
- Choose variable names to fit the purpose for which the variable is defined
- The name of the variable may refer to the region of memory (if the name appears on the left hand side of an assignment), or its value (if the name appears on the right hand side of an assignment)

# Further Remarks

- Expressions in C++ are similar to those in mathematics, except that values may get converted from integer to real or vice versa and truncation might happen
- Truncation may also happen when values get stored into a variable
- Sequence generation and accumulation are very common idioms
- Increment/decrement operators and compound assignment operators also are commonly used (they are not found in mathematics)

# More Remarks

- Variables can be defined inside any block
- Variables defined outside a block may get shadowed by variables defined inside

# 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

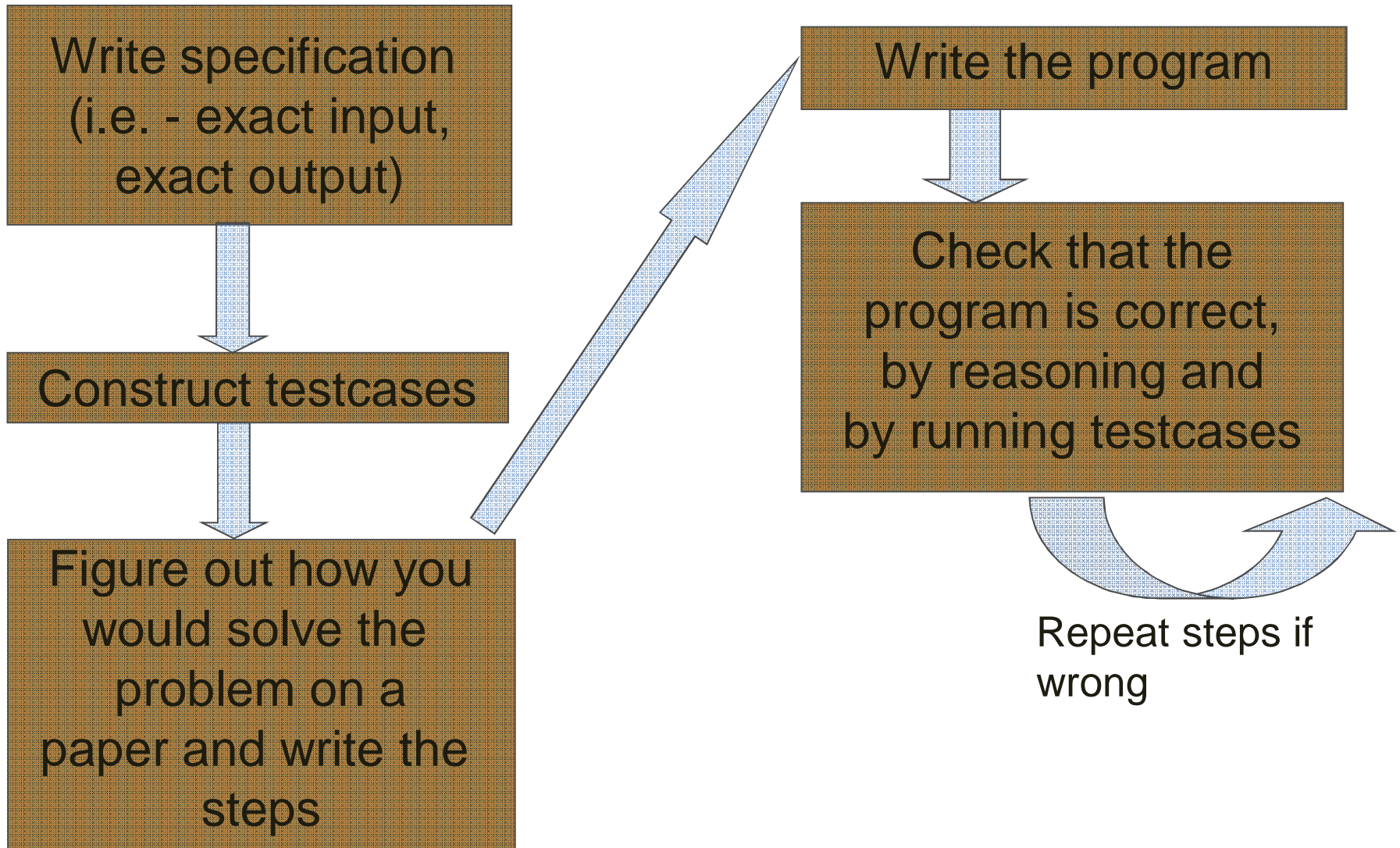
As a professional programmer, you must remember that an incorrect program could cause a plane to crash, an X-ray machine to supply the wrong amount of radiation: your program may be controlling such devices



# Program Development Strategy

1. Writing specification
2. Constructing test cases
3. Thinking how you would 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



# 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  $1/0! + \dots + 1/n!$

- 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

# Designing the Algorithm (2)

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



# What Variables To Use

- When we solve the problem on paper, we will write a lot of numbers; we do not need separate variables to store all those
- As you do the calculation on paper, think of how many of the numbers you have written down are potentially useful at the same time. 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  $t^{\text{th}}$  term from the  $t-1^{\text{th}}$ . So we need to remember the previous term. So let us have a variable **term** to remember this
- 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 have to decide what values  $i$ ,  $term$  should have when we enter the loop for the  $t^{\text{th}}$  time, where  $t=1, 2, \dots, n$
- In the iterations of the loop 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  $t^{\text{th}}$  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  $t-1 = 0$
  - $\text{term}$  must have the value  $(t-1)!=1$
- Thus before the loop we must initialize
  - $i=0$ ;  $\text{term}=1$ ;
- Inside the loop we have to add the next term to result. But  $i$  and  $\text{term}$  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:
  - $\text{term} = \text{term}/i$
- Now we have to add this into result. So we have:
  - $\text{result} = \text{result} + \text{term}$
- Now result has the sum upto  $1/t!$ , so  $t^{\text{th}}$  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  $t+1^{\text{th}}$  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



# 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