

CS 101: Computer Programming and Utilization

Jan-Apr 2017

Sharat

(piazza.com/iitb.ac.in/summer2017/cs101iitb/home)

Lecture 6: More On Loops

About These Slides

- Based on Chapter 3 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

The Need of a More General Loop

Read marks of students from the keyboard and print the average

- Number of students not given explicitly
- Two cases
 1. If a negative number is entered as marks, then it is a signal that all marks have been entered

Examples

- Input: 98 96 -1, Output: 97
- Input: 90 80 70 60 -1, Output: 75

2. No such artificial signal

- b. The **repeat** statement repeats a fixed number of times.
Not useful

Outline

The **while** statement

- Some simple examples
- Mark averaging

The **break** statement

The **continue** statement

The **do while** statement

The **for** statement

Recap

The **while** statement

- Some simple examples

Mark averaging

- with no negative numbers
- with negative numbers

A celebrated algorithm: GCD

Nested WHILE Statements

We can put one while statement inside another. The execution is as you might expect. Example:

```
int i=3;
while(i > 0) {
    i--;
    int j=5;
    while(j > 0){
        j--;
        cout << "A";
    }
    cout << endl;
}
```

What do you think this will print?

Time To Demo

The CONTINUE Statement

- `continue` is another single word statement
- If it is encountered in execution, the control directly goes to the beginning of the loop for the next iteration, skipping the statements from the `continue` statement to the end of the loop body

Example

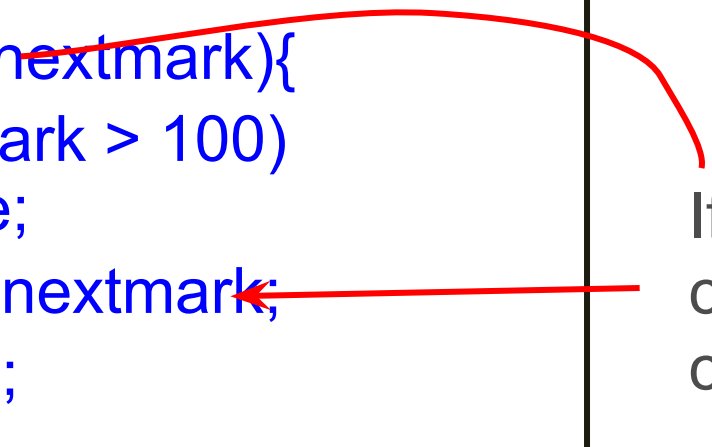
Mark averaging with an additional condition :

- if a number > 100 is read, discard it (say because marks can only be at most 100) and continue with the next number. As before stop and print the average only when a negative number is read

Code For New Mark Averaging

```
main_program{  
    float nextmark, sum = 0;  
    int count = 0;  
    while (cin >> nextmark){  
        if(nextmark > 100)  
            continue;  
        sum += nextmark;  
        count++;  
    }  
    cout << sum/count << endl;  
}
```

If executed, the control goes back to condition evaluation




Time To Demo

The BREAK Statement

- The `break` keyword is a statement by itself
- When it is encountered in execution, the execution of the innermost while statement which contains it is terminated, and the execution continues from the next statement following the while statement

Code For New Mark Averaging

```
main_program{
  float nextmark, sum = 0;
  int count = 0;
  while (true){
    cin >> nextmark;
    if(nextmark > 100)
      continue;
    if(nextmark < 0)
      break;
    sum += nextmark;
    count++;
  }
  cout << sum/count << endl;
}
```

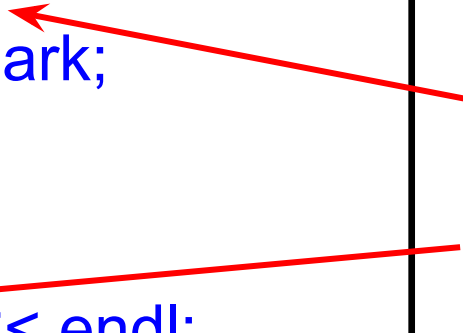


If executed, the control goes back to condition evaluation

Example of BREAK

```
main_program{
  float nextmark, sum = 0;
  int count = 0;
  while(true){
    cin >> nextmark;
    if(nextmark < 0)
      break;
    sum += nextmark;
    count++;
  }
  cout << sum/count << endl;
}
```

If **break** is executed, control goes here, out of the loop



Explanation

- In our mark averaging program, we did not want to check the condition at the beginning of the repeated portion
- The `break` statement allows us just that!
- So we have specified the loop condition as `true`, but have put a `break` inside
- The statements in the loop will repeatedly execute; however when a negative number is read, the loop will be exited immediately, without even finishing the current iteration
- The `break` statement is of course useful in general

The DO-WHILE Statement

Not very common

Discussed in the book

The FOR Statement: Motivation

- Example: Write a program to print a table of cubes of numbers from 1 to 100

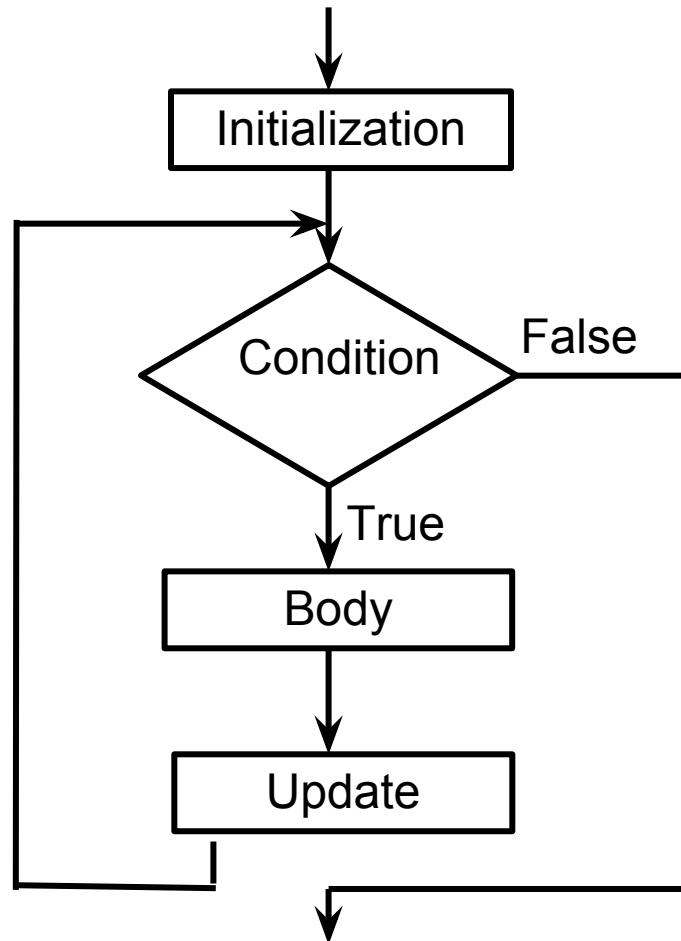
```
int i = 1;
repeat(100){
    cout << i << ' ' << i*i*i << endl;
    i++;
}
```

- This idiom: do something for every number between x and y occurs very commonly
- The **for** statement makes it easy to express this idiom, as follows:

```
for(int i=1; i<= 100; i++)
    cout << i << ' ' << i*i*i << endl;
```

Flowchart for FOR Statement

Previous statement in the program



Next statement in the Program

The FOR Statement

```
for(initialization; condition; update)  
  body
```

- **initialization, update** : Typically assignments (without semi-colon)
- **condition** : boolean expression
- Before the first iteration of the loop the **initialization** is executed
- Within each iteration the **condition** is first tested. If it fails, the loop execution ends. If the **condition** succeeds, then the **body** is executed. After that the **update** is executed. Then the next iteration begins

Definition of Repeat

repeat(n)

is same as

```
for (int _iterator_i = 0, _iterator_limit = n;  
     _iterator_i < _iterator_limit;  
     _iterator_i ++)
```

Hence changing n in the loop will have no effect in the number of iterations

Whether a number is prime

```
main_program{  
  int n; cin >> n;  
  bool found = false;  
  for(int i=2; i < n && !found; i++){  
    if(n % i == 0){  
      found = true;  
  
    }  
  }  
  if(found) cout << "Composite.\n";  
  else cout << "Prime.\n";  
}
```

Remarks

- `while`, `do while`, `for` are the C++ statements that allow you to write loops
- `repeat` allows you to write a loop, but it is not a part of C++ It is a part of `simplecpp`; it was introduced because it is very easy to understand.
- Now that you know `while`, `do while`, `for`, you should stop using `repeat`

Remarks

An important issues in writing a loop is how to break out of the loop. You may not necessarily wish to break at the beginning of the repeated portion. In which case you can either duplicate code, or use ***break***

Learn Methods For Common Mathematical Operations

- Evaluating common mathematical functions such as
Sin(x)
log(x)
- All the methods we study are approximate. However, we can use them to get answers that have as small error as we want
- The programs will be simple, using just a single loop

Series expansion for f

- Taylor's series

$$f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + f'''(x) h^3/3!$$

- MacLaurin Series: Choose $x = 0$

MacLaurin Series

When x is close to 0:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)x^2}{2!} + \frac{f'''(0)x^3}{3!} + \dots$$

E.g. if $f(x) = \sin x$

$$f(x) = \sin(x), \quad f(0) = 0$$

$$f'(x) = \cos(x), \quad f'(0) = 1$$

$$f''(x) = -\sin(x), \quad f''(0) = 0$$

$$f'''(x) = -\cos(x), \quad f'''(0) = -1$$

$$f''''(x) = \sin(x), \quad f''''(0) = 0$$

Now the pattern will repeat

Example

$$\text{Thus } \sin(x) = x - x^3/3! + x^5/5! - x^7/7! \dots$$

A fairly accurate value of $\sin(x)$ can be obtained by using sufficiently many terms

Error after taking i terms is at most the absolute value of the $i+1$ th term

Program Plan-High Level

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! \dots$$

Use the *accumulation idiom*

Use a variable called **term**

This will keep taking successive values of the terms

Use a variable called **sum**

Keep adding **term** into this variable

Program Plan: Details

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! \dots$$

- Sum can be initialized to the value of the first term So
sum = x
- Now we need to figure out initialization of term and it's
update
- First figure out how to get the k th term from the $(k-1)$ th
term

Program Plan: Terms

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! \dots$$

Let t_k = kth term of the series, $k=1, 2, 3\dots$

$$t_k = (-1)^{k+1} x^{2k-1} / (2k-1)!$$

$$t_{k-1} = (-1)^k x^{2k-3} / (2k-3)!$$

$$t_k = (-1)^k x^{2k-3} / (2k-3)! * (-1)(x^2) / ((2k-2)(2k-1))$$

$$= - t_{k-1} (x)^2 / ((2k-2)(2k-1))$$

Program Plan

- Loop control variable will be k
- In each iteration we calculate t_k from t_{k-1}
- The term t_k is added to sum
- A variable $term$ will keep track of t_k
At the beginning of k^{th} iteration, $term$ will have the value t_{k-1} , and at the end of k^{th} iteration it will have the value t_k
- After k^{th} iteration, sum will have the value = sum of the first k terms of the Taylor series
- Initialize $sum = x$, $term = x$
- In the first iteration of the loop we calculate the sum of 2 terms. So initialize $k = 2$
- We stop the loop when $term$ becomes small enough

Program

```
main_program{
    double x; cin >> x;
    double epsilon = 1.0E-20; // arbitrary.
    double sum = x, term = x;
    for(int k=2; abs(term) > epsilon; k++){
        term *= -x*x / (2*k - 1) / (2*k - 2);
        sum += term;
    }
    cout << sum << endl;
}
```