Programs with Loops, The while and do-while Loops

ESC101: Fundamentals of Computing

Nisheeth

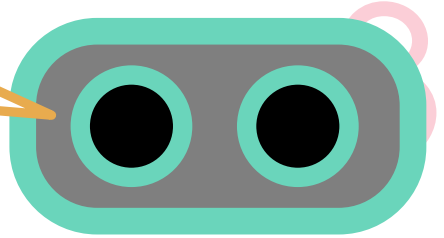
Announcements

- Major Quiz 1 tomorrow (L-20, 12:00-12:50). Instructions already shared

- Must write your name on answer sheets (minor/major quizzes/exams)
  - Your responsibility. If you miss, it makes it very hard/impossible for us to locate it

Recap: for Loo

Brackets essential if you want me to do many things while looping



General form of the for loop

```
for(init_expr; stopping_expr; update_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

Initialization expression is executed only once

What this piece of code means?

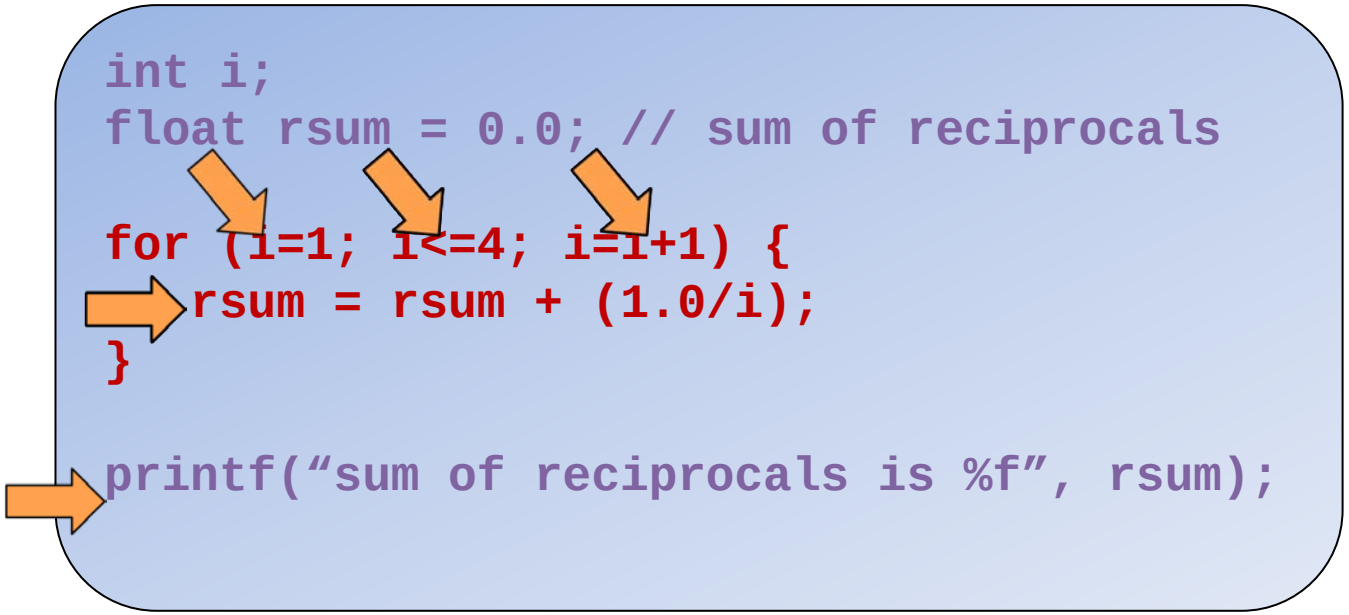
1. First do what is told in initialization expression
2. Then check the stopping expression
3. If stopping expression is true
   - Execute all statements inside braces
   - Execute update expression
   - Go back to step 2

   Else stop looping and execute rest of code

```
int i;
float rsum = 0.0; // sum of reciprocals

for (i=1; i<=4; i=i+1) {
    rsum = rsum + (1.0/i);
}

printf("sum of reciprocals is %f", rsum);
```



i	rsum
5	2.0833333..

rsum = 1 + 1/2 + 1/3 + 1/4

1. Evaluate init_expr; i.e., i=1;
2. Evaluate test_expr i.e., i<=4 TRUE
3. Enter body of loop and execute.
4. Execute update_expr; i=i+1; i is 2
5. Evaluate test_expr i<=4: TRUE
6. Enter body of loop and execute.
7. Execute i=i+1; i is 3
8. Evaluate test_expr i<=4: TRUE
9. Enter body of loop and execute.
10. Execute i=i+1; i is 4
11. Evaluate test_expr i<=4: TRUE
12. Enter body of loop and execute.
13. Execute i=i+1; i is 5
14. Evaluate test_expr i<=4: FALSE
15. Exit loop & jump to printf

sum of reciprocals is 2.083333

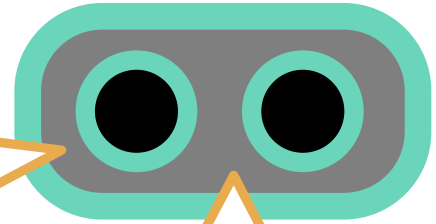
The for Loop: More on its syntax..

Many forms possible for the init/stopping/update expressions. Some examples:

```
for (i=1;i <= 10; i+=2)
```

```
for (i=1;i <= 64; i*=2)
```

init_expr can also be before the start of for loop
update_expr can also be inside the body of for loop



Learn more by practicing ☺

```
for (i=-5,i <= 10; i++)
```

```
for (i=10;i >= 0; i--)
```

```
for (i=1,j=2;i <= 10 && j <= 20; i++,j=j+2)
```

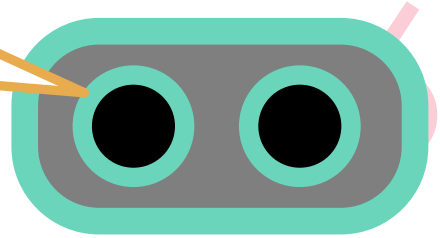
Multiple loop counter variables for the init/ stopping/update expressions

```
for (i=1;i <= 10; i++){
      for(j=1; j<= i; j++){
      }
}
```

Inner loop's counter can depend on outer loop counter's current value

The while loop

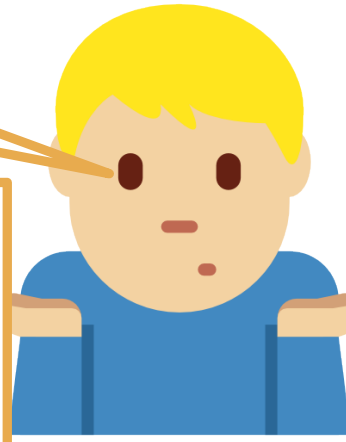
Brackets essential if you want me to do many things while looping



General form of a while loop

```
while(stopping_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
...
```

So what is the difference between for and while?



In general not much – it is a matter of style. Often we use while when we don't exactly know how many iterations will loop run (but usually it can be done with for loop too)

What this piece of code does?

1. First check the stopping expression
2. If stopping expression is true
   Execute all statements inside braces
   Go back to step 2
   Else stop looping and execute rest of code

The while loop in action..


```
int a;
scanf("%d", &a);                /* read into a */
while ( a !=  -1) {
    scanf("%d", &a);  /*read into a inside loop*/
}
```

INPUT

4
15
-5
-1
-3

-1

Trace of memory location a

- **One scanf is executed every time body of the loop is executed.**
- **Every scanf execution reads one integer.**

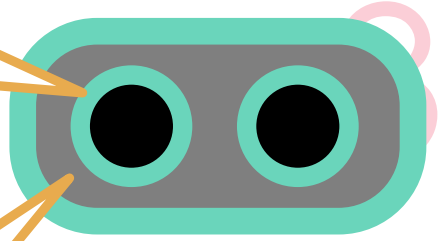
The do-while lo

General form of a do-while loop

```
do{
    statement1;
    statement2;
    ...
}while(stopping_expr);
statement3;
statement4;
...
```

When to use do-while instead of while?

Brackets essential if you want me to do many things while looping



Notice additional semi-colon

Yet another minor quiz question

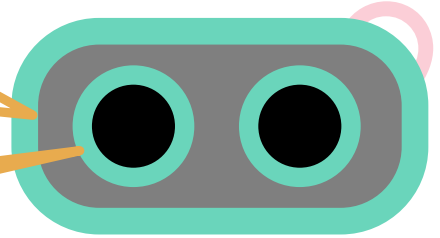
What this piece of code does?

1. First execute statements inside braces
2. Then check stopping criterion

2. If stopping expression is true
   Execute all statements inside braces
   Go back to step 2
   Else stop looping, execute rest of code

The use of do-wh

Notice proper indentation for while and do-while loops



while and do-while equally powerful, sometimes one looks prettier, easier to read than the other

The do-while loop is e

Example: *read intege* *r -1 and …*

```
int num, sum = 0;
scanf(“%d”, &num);
while(num != -1){
    sum += num;
    scanf(“%d”, &num);
}
printf("%d",sum);
```

```
int num, sum = 0;
do{
    scanf(“%d”, &num);
    if(num != -1)
        sum += num;
}while(num != -1);
printf("%d",sum);
```

Some more examples, tips, and guidelines on using loops

Properly Divide Task into Subtasks

1
1 2
1 2 3
1 2 3 4
...
1 2 3 4 ... 10

Consider printing the pattern shown on right

Step 1: Divide problem into smaller tasks that are very similar and have to be repeated

Often, more than one way may seem possible. Not all may be implementable

For this problem, column-wise printing will be hard. But row-wise printing seems like an implementable idea. Row i can be printed using the following for loop

```
for(j = 1; j <= i; j++)
    printf("%d ", j);
printf("\n");
```

```
for(i=1;i<=10;i++){
    for(j = 1; j <= i; j++)
        printf("%d ", j);
    printf("\n");
}
```

Example of nested for

Can repeat the above for i = 1 to i = 10 using an outer loop

Order of statements is important

- Given positive real numbers r and a, and a positive integer, n, the n^{th} term of the geometric progression with a as the first term and r as the common ratio is ar^{n-1}.
- Write a program that given r, a, and n, displays the first n terms of the corresponding geometric progression.

Order of statements is important

```
int main(){
    int n, i; float r, a, term;
   // Reading inputs from the user
   scanf("%f", &r);
   scanf("%f", &a);
   scanf("%d", &n);
   term = a;
   for (i=1; i<=n; i=i+1) {
        printf("%f\n", term); // Displaying i^th term
        term = term * r;        // Computing (i + 1)^th term
   }
    return 0;
}
```

Order of statements is important

```
int main(){
    int n, i; float r, a, term;
   // Reading inputs from the user
   scanf("%f", &r);
   scanf("%f", &a);
   scanf("%d", &n);
   term = a;
   for (i=1; i<=n; i=i+1) {
        term = term * r;
       printf("%f\n", term);
   }
    return 0;
}
```

Careful: Changing the order of statements changes the meaning of the program.

Computation of

$a, ar, \ldots, ar^{n-1}$ (previous program)

$ar, ar^2, \ldots, ar^n$ (this program)

The break keyword

Allows us to stop execu nd exit immediately

- Even if the stopping conditio
- Can be used inside a for loop, while loop, d while loop

If we did not have break, infinite loop!

When to use break

- Avoid if possible
- Can make code error-prone and hard to read
- Used when one stopping condition not enough
- Or sometimes to make code more elegant
- Allows us to avoid specifying a stopping condition

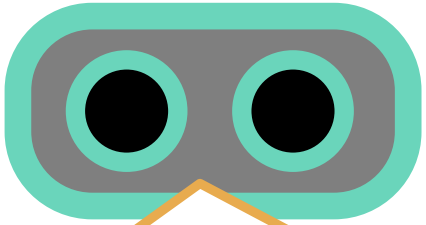
Note: Here, the else not even needed since Mr C. neglects all remaining statements in loop body upon encountering break;

```
int num, sum = 0;
while(1){
    scanf("%d", &num);
    if(num == -1)
break;
    sum += num;
}
printf("%d",sum);
```

A [illegible]ng break

If we did not have break this would have been an infinite loop since no stop_expr

```
for(init_expr; stop_expr; upd_expr){
    statement1;
    statement2;
    ...
}
statement3;
statement4;
```



Remember, the order is stop_expr → body → update → stop_expr → …

```
init_expr;
for(;;){
    if(!(stop_expr)) break;
    statement1;
    statement2;
    ...
    upd_expr;
}
statement3;
statement4;
```

The continue keyword

- Allows us to skip the rest of the statements in body of the loop
- Upon encountering continue, Mr C thinks that body of loop is over
- Loop not exited (unlike break)
  - If we say continue in for loop, update_expr evaluated, then stop condition checked
  - If we say continue in while or do-while loop, then stop condition checked
  - In all cases, rest of body not executed
- Read 100 integers and print sum of only positive numbers

```
int sum = 0, i, num;
for(i = 1; i <= 100; i++){
    scanf("%d", &num);
    if (num < 0){
        continue;
    }
    sum += num;
}
```

Careful u e

If there are ne ner l apply only to the inner loop

If we have a sequ numbers, it will re we wanted to rea continue stateme

Much better, always updates counter whether skipping the sum+=num step using continue or not

Be careful not to create an infinite loop using continue if you bypass any update steps.

```
for (i = 0; i < 100; i++){
    for (j = 0; j < 100; j++){
        if (…) continue;
    }
    statement1;
}
statement2
```

```
int i = 0, sum = 0, num;
while (i < 100){
    i++;
    scanf("%d", &num);
    if (num < 0) continue;
    sum += num;
}
```

Careful using break, continue

Excessive use of break and continue can make your program error-prone, and hard for you to correct

If you have 10 break statements inside the same loop body, you will have a hard time figuring out which one caused your loop to end

If you have 10 continue statements inside the same loop body, you will have a hard time figuring out why body statements are not getting executed.

Should not misuse break, continue - used in moderation these can result in nice, beautiful code

We will see some elegant alternatives to break, continue

Break and Continue: Summary

Break helps us exit loop immediately

- In for loops, even update_expr or stop_expr not checked – just exit
- In while, do-while loops, even stop_expr not checked – just exit

Continue helps us skip the rest of the body of loop

- In for loops, after Mr C receives a continue statement, he evaluates the update_expr (if it's inside for() part), then checks the stop_expr and so on …
- In while loops, after Mr C receives a continue statement, he checks the stop_expr
- Loop not exited just because of continue, stop_expr still controls exit

Warning: Break legal only in body of loops and switch

- Illegal inside body of if, if-else statements

Warning: Continue legal only in body of loops

- Illegal inside body of if, if-else, switch statements