Introduction to Computer Science
Lecture 3

Nazareno Aguirre

(based on material by Guillaume Hoffmann)

Today's Topics

- logic operators
- interval and set conditions
- while statements
- boolean values as integers
- do-while

Logic Operators

Logic operators allow us to combine conditions:

- `EXPR || EXPR` : **logical 'or'** of two expressions
- `EXPR && EXPR` : **logical 'and'** of two expressions

These conditions can be used in the context of **conditional statements** and **loops**.

Logic operators have less priority than the comparison operators, so you can write the following without parentheses:

- `a == 10 || b >= 5 || c < 50`
- `a == 10 && b >= 5 && c < 50`

Interval Conditions

To express that some expression EXPR's value belongs to an interval [a,b], you need to write two comparison expressions:

- EXPR is greater than or equal to a: `EXPR >= a`
- EXPR is less than or equal to b: `EXPR <= b`

Then you combine both into a single condition with the logical 'and' operator:

- `EXPR >= a && EXPR <= b`

expression `(a <= EXPR <= b)` is syntactically correct in C, but it does not have the meaning one usually associates with such interval conditions.

Example

```
main() {
    int age;
    scanf("%d", &age);

    // check if user age belongs to range [18,60]
    if (age >= 18 && age <= 60)
        printf("You can apply to this job!\n");
    else
        printf("Sorry, you are too young or too old.\n");
}
```

Check if some value belongs to a set of values

If the set of options is very small, you can enumerate its elements and check whether a given value is equal to the first one, or equal to the second one, etc.

Example:

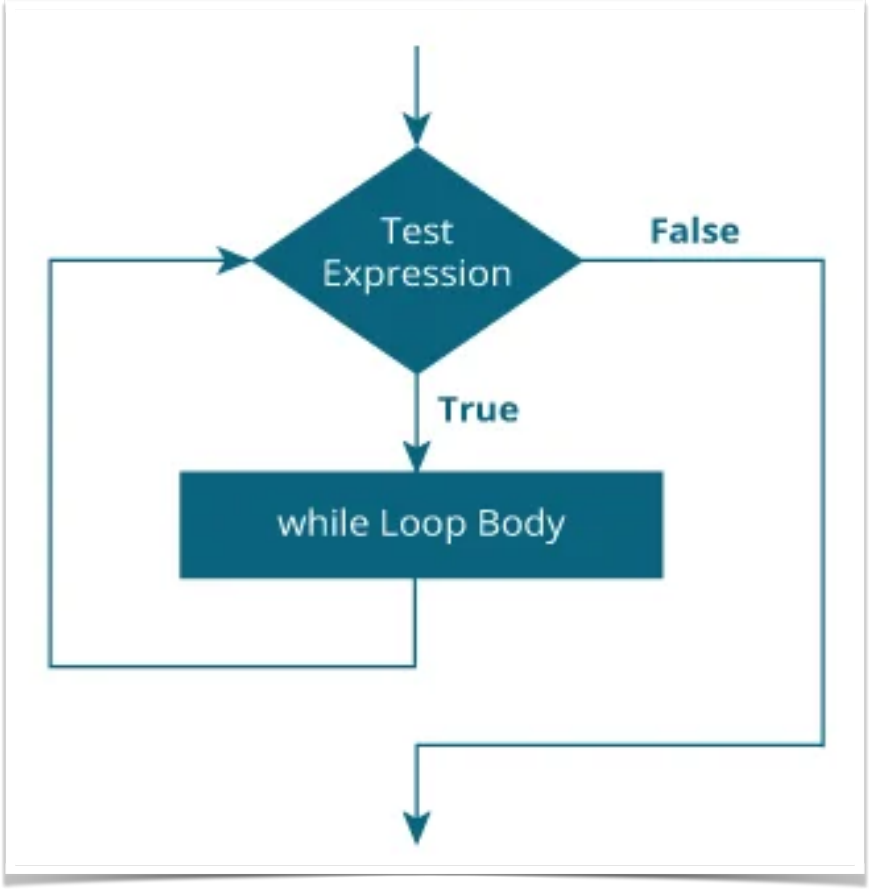
```
main() {
    int age;
    scanf("%d", &age);

    // check if user age belongs to set {20,30,40}
    if (age == 20 || age == 30 || age == 40)
        printf("We have a special gift for you!\n");
}
```

The while statement

```
while (CONDITION)
    [STATEMENT]
[NEXT STATEMENTS]
...
```

```
while (CONDITION) {
    [STATEMENTS]
}
[NEXT STATEMENTS]
...
```



while statement example

```
main() {
    int count;
    printf("Please enter a number to do a countdown.\n");
    scanf("%d", &count);

    while (count >= 0){
      printf("%d.\n", count);
      count = count - 1;
    }
}
```

While Statement: Things to Care About

- **A while statement repeats a sequence of statements** (the "body" of the while loop) while some condition is true.
- It stops repeating when the condition is false.
- Condition is checked when the while statement is first reached, and after the whole body is executed, before it is repeated again.
- If you write a while statement and the condition is always true, your program will never stop; then you need to use CTRL+c to terminate it (or use the `kill` command).

Manual Execution with while

```
int i = 0;
int a = 0;
while (i < 4) {
    if (i % 2 == 0)
        a = a + 1;
    else
        a = a - 1;
    i = i + 1;
}
```

i	a	i < 4	i % 2 == 0
0	-	-	-
0	0	-	-
0	0	true	-
0	0	-	true
0	1	-	-
1	1	-	-
1	1	true	-
1	1	-	false
1	0	-	-
2	0	-	-
2	0	true	-
2	0	-	true
2	1	-	-
3	1	-	-
3	1	true	-
3	1	-	false
3	0	-	-
4	0	-	-
4	0	false	-

How Conditions are Evaluated

- The C language uses integers for boolean testing. What does this mean?
- 0 represents "false", and 1 represents "true".
  - example: `(1 > 0)` evaluates to 1
  - example: `(1 != 1)` has a value of 0
- Consider `(a <= EXPR <= b)`
  - Read as `((a <= EXPR) <= b)`, the inner expression is evaluated first as a boolean value 0 or 1.
- Then, evaluation continues with either `(0 <= b)` or `(1 <= b)`
- This means *something*, but probably not what you want!

How Conditions are Evaluated

- In a condition, 0 is interpreted as false and **anything non-zero** is interpreted as true.
- That is: `if(0)` is always false, `if(1)` is always true, `if(-2)` too, etc.
- Same for `while(1)`.
- Use `gcc -Wall` to detect possible problems (tcc is not good enough):

```
$ gcc -Wall interval.c -o interval
interval.c: In function ‘main’:
interval.c:5:15: warning: comparison of constant ‘100’ with boolean expression is always true [-Wbool-compare]
    5 |   if (50 <= x <= 100)
      |               ^~
interval.c:5:10: warning: comparisons like ‘X<=Y<=Z’ do not have their mathematical meaning [-Wparentheses]
    5 |   if (50 <= x <= 100)
      |       ~~~^~~~
```

Conditions have values, continued

Let us read the manual of GCC (`man gcc` , does not work in JSLinux):

```
-Wbool-compare
    Warn about boolean expression compared with an
    integer value different from "true"/"false".
    For instance, the following comparison is always
    false:

            int n = 5;
            ...
            if ((n > 1) == 2) { ... }

    This warning is enabled by -Wall.
```

GCC options on the command line

GCC's most important options for us are:

-o name: specifies a name for the executable file generated; by default it’s a.out

-Wall: enable all warnings detection

-Werror: make all warnings into errors

You can combine all of them:

gcc main.c -o program -Wall -Werror

TCCs

- The TCC compiler has a "compile and execute" mode activated with the `-run` flag.
  - Useful in the typical case in which we want to quickly check the behavior of our program.
  - Usually we do not care about warnings in that case, because we just want to see the program's output, except if there is an error. Hence we can use the -w flag to suppress any warning:

```
tcc -w -run main.c
```

Another Useful Warning

Another common error is to confuse assignment (=) with equality testing (==). For instance, the following program compiles but it does not do what one would intuitively expect:

```
main() {
 int a = 10;
 if (a = 20)
   printf("You should not see this.\n");
}
```

This is because assignments `a=b` are expressions that also have an associated value

The if statement takes assignment `a=20` (value 20, non-zero) as a true condition!

The Logical NOT Operator

Aside the binary logical operators `||` and `&&` ("or" and "and"), we also have the "not" operator, that changes the truth value of some expression:

```
if(!(a > b)) ...
```

It is very useful to express while loop conditions as a negated "until" condition:

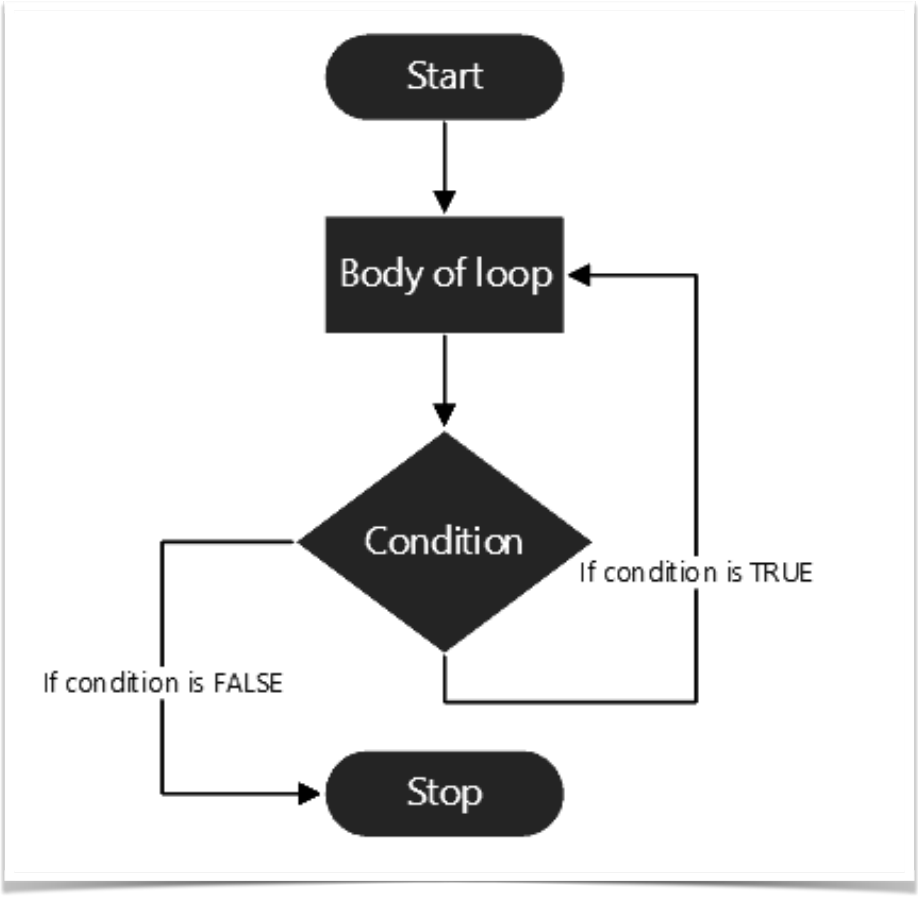
```
while ( !(a == 0) )    // iterate until a == 0
```

The do-while loop

The do-while statement is similar to the while statement, except that the loop body is executed before the condition is checked for the first time.

Syntax:

```
do {
     STATEMENT
} while (CONDITION);
```



Using do-while to validate user input

- You can put a scanf() statement in the do-while loop body and repeat it until the input value fulfills some condition.
- For instance, asking for a non-negative integer:

```
do {
    scanf("%d", &x);
} while (!(x >= 0)); // repeat until x is non-negative
```

Equivalence

```
do {
    do_work();
} while (condition);
```

is equivalent to:

```
do_work();

while (condition) {
    do_work();
}
```

Remarks

- Do-while loops are sometimes useful if you want the code to output some sort of menu to a screen so that the menu is guaranteed to show once.
- If there is no need to use a do-while loop, then a regular while loop is preferred.

While loop examples

The following program uses a while loop to repeat a message a number of times:

```c
#include <stdio.h>

int main() {
    int i = 0;
    while (i < 10) {
        printf("tick...\n");
        i = i + 1;
    }
    printf("BOOOOOM!\n");
}
```

While loop examples

Let's compute an integer summation using a while loop:

```c
#include <stdio.h>

int main() {
    int i = 0;
    int sum = 0;
    int n;
    scanf("%d", &n);
    while (i <= n) {
        sum = sum + i;
        i = i + 1;
    }
    printf("The sum from i = 0 to %d of i is %d\n", n, sum);
}
```

While loop examples

Let’s compute the average of the first n even (natural) numbers:

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    int i = 0;
    int sum = 0;
    while (i < n) {
        sum = sum + 2*i;
        i = i + 1;
    }
    printf("%d\n", sum / n);
}
```

While loop examples

Let's write a more challenging program, least common multiple:

```c
#include <stdio.h>

int main() {
    int n, m;
    scanf("%d", &n);
    scanf("%d", &m);
    int i = n;
    while ((i % n != 0) || (i % m != 0)) {
        i = i + 1;
    }
    printf("least common multiple of %d and %d is: %d\n", n, m, i);
}
```

Final remarks: Recommendations for Loops

- Make the loop condition clear, with regards to the specification of the problem.
- If the problem specifies an amount of repeats of the loop, make the loop condition refer to the repeats.
- If the problem is given as a bound condition, put the bound condition in the loop condition.
- In general, try to make your code the most transparent possible translation of the problem specification. Avoid simplifying conditions in an effort to make your program more efficient.
  - Program readability is an important issue too!