Lecture 6: Conditionals and Loops

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/06-conditionals-loops.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

In this lecture we discuss different control flow statements in C: conditionals, loops, and functions.

Conditionals

To create a **conditional** we use the `if` statement. If the condition within the parentheses is true the code within the braces will be executed. If the condition is false, the `if` body is not executed and execution resumes right after. This will call `printf` only if `x` is equal to 50:

```
if (x == 50) {
    printf("The value of x is 50\n");
}
```

- We can use `else` to define a block of code that is executed only if the condition is false:

```
if (x == 50) {
    printf("The value of x is 50\n");
} else {
    printf("The value of x is not 50\n");
}
```

- We can chain the `if` statements as presented on the slide:

```
if(x < 50) {
    printf("The value of x is strictly less than 50\n");
} else if (x == 50) {
    printf("The value of x is exactly 50\n");
} else {
    printf("The value of x is strictly more than 50\n");
}
```

On this example, only one of the three `printf` statements will be executed depending on the value of `x`. To write the condition one can use various operators. Note the use of `==` for the equality. Do not mix it with the assignment operator `=`, this can produce nasty bugs. Inequality can be checked with `!=`.

Writing the Condition

There is no boolean type per se in C. A condition is false if it evaluates to `0`. And it is true if it evaluates to anything but `0`. We have the following boolean operators: `&&` (logical AND), `||` (logical OR), as well as the negation `!`. Here are a few examples of usage:

```c
int one = 1;
int zero = 0;
if (one) { /* will take this branch */}
if (zero) { /* won't be taken */}
if (!one) { /* won't take */}
if (one || zero) { /* will take */ }
if (one && zero) { /* won't take */ }
if (one && zero || one && zero) { /* won't take, && evaluated before || */ }
if (one && (one || zero)) { /* will take */}
```

The negation operator `!` is placed before the expression one want to negate. The `||` and `&&` operators in between two expressions. Be careful about operator precedence, which is the order of priority of evaluation. The negation `!` has the highest priority, then comes `&&`, then `||`. To modify the evaluation order we can use parentheses.

Switch Statement

When one has to write a long `if ... else` chain checking a given integer value, the switch statement is useful. We put the condition to evaluate or a variable after the `switch` between parentheses. Then we write a series of `case` blocks where the execution will jump according to the condition value:

```c
switch(x) {
    case 1:
        printf("x is 1\n");
        break;
    case 2:
        printf("x is 2\n");
        break;
    case 3:
        printf("x is 3\n");
        break;
    default:
        printf("x is neither 1, nor 2, nor 3\n");
}
```

For example here let us assume `x` is equal to 2. The code will jump to `case 2`, execute the `printf` statement. When encountering a `break` statement the code will exit the switch. It is important not to forget breaks otherwise all the cases below the one taken will be executed. There is a special case named `default` that is taken if none of the other match. Note that the switch's condition needs to evaluate to an integer.

While Loop

The `while` loop keeps executing its body as long as a given condition is true. In this first example the code will keep printing and decrement `x` as long as `x` is superior to 0:

```c
int x = 10;
while (x > 0) {
    printf("x is %d\n", x);
    x = x - 1;
}
```

There are two flavours of the while loop. The first, corresponding to the example above, in which the condition is checked before entering the loop:

```c
while(/* condition */) {
    /* body */
}
```

And the other with `do ...while`, where the condition is checked after the loop body:

```c
do {
    /* body */
```

```
} while (/* condition */);
```

The following code is an infinite loop, the condition is always true:

```
while(1) {
    printf("hello\n");
}
```

If a program gets stuck, one can type `ctrl` + `c` to kill it.

For Loop

Very often we use some kind of iterator within a loop. The `for` loop is perfect for that. In the header of the loop there are 3 things separated by semicolons:

- An initial statement executed once at the beginning of the loop.
- A condition checked before each iteration of the loop.
- If true, the loop continues, if false, it exits.

And a last statement executed after each iteration, generally used to update the iterator. In this example we set `i` to `0` before starting the loop, which runs for as long as `i` is inferior to `10`:

```
for(int i = 0; i<10; i = i + 1) {
    printf("i is %d\n", i);
}
```

After each iteration, we increment `i` by one.

`break` and `continue`

The `break` statement within a `for` or `while` loop body directly exits the loop. In this example the loop will iterate until i reaches 5:

```
for(int i = 0; i < 10; i = i + 1) {
    printf("iteration %d\n", i);
    if(i == 5) {
        break;
    }
}
```

The `continue` statement within a `for` or `while` loop body jumps directly to the next iteration. In this example we print all numbers from 0 to 9, except 5:

```
int i = 0;
while(i < 10) {
    i = i + 1;
    if(i == 5) {
        continue;
    }
    printf("iteration %d\n", i);
}
```

Loops and Arrays

Loops are useful to manipulate arrays. Here we fill and print an array within a loop, using the iterator as index:

```
int my_array[4];
for(int i=0; i<4; i = i +1) {
    my_array[i] = 100 + i;
    printf("index %d contains: %d\n", i, my_array[i]);
}
```

To iterate over a multidimensional array we can use nested loops:

```
int my_2d_array[3][2];
for(int i=0; i<3; i++) {
    for(int j=0; j<2; j++) {
        my_2d_array[i][j] = i*j;
        printf("Index [%d][%d] = %d\n", i, j, my_2d_array[i][j]);
    }
}
```

On this 3 by 2 array we use a first loop doing 3 iterations and inside a second loop doing 2 iterations.

Back on Command Line Arguments

With proper control flow instructions, we can now fix our code sample printing command line arguments from the previous lecture. Remember that with a hard coded number of arguments printed, we were at the risk of memory errors. With a conditional we can ensure the program is called with the proper number of arguments, like in this example:

```
int print_help_and_exit(char **argv) {
    printf("Usage: %s <number 1> <number 2>\n", argv[0]);
    exit(-1);
}

/* Sum the two integers passed as command line integers */
int main(int argc, char **argv) {
    int a, b;
    if(argc != 3)
        print_help_and_exit(argv);
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("%d + %d = %d\n", a, b, a+b);
    return 0;
}
```

Braces in Conditionals and Loops

When a conditional/loop body consists of a single statement, we can omit the braces:

```
if (x)
    printf("x is non-null\n");
else
    printf("x is 0\n");
for (int i = 0; i<10; i = i +1)
    printf("i is %d\n", i);
```

If later more statements need to be added, one needs to put the braces back. Forgetting to do so can lead to nasty bugs:

```
if(one)
    if(two)
        foo();
  else           // whose else is this?
        bar();
```