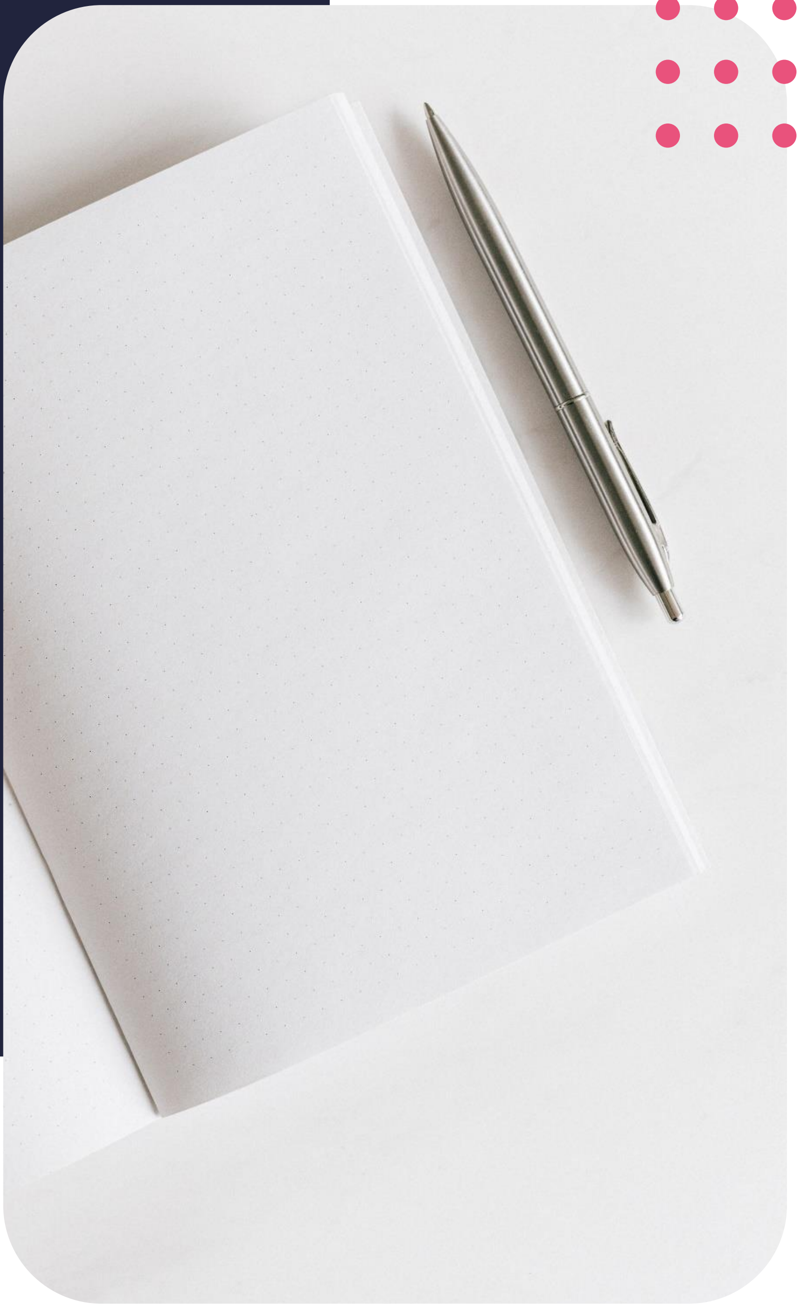
Diploma in Computer Science

Loops and functions

Contents



Lesson outcomes

By this lesson, you should be able to:

- Understand how computers achieve repetition
- Discuss the methods for carrying out repetitive tasks
- Explore common repetitive programs
- Appreciate the common mistakes

Repetition

Let’s face it, human’s hate repetition. Imagine sitting at a desk. With a 500-sheet list of items that need to be counted and classified. I, personally, would rather die than do such a thing, and I’m sure a lot of other people agree with me. Repetition breaks your soul; it feeds on your muscles and mental energy. Luckily, there are people like you have undertaken to write code that automates repetition, in fact, humans hate repetition so much that it’s one of the main reasons why computers were invented. Unlike humans, computers perform repetitive tasks accurately, every single time. That said, this only works when you, yes you, the programmer, instruct the computer correctly. There are 3 main types of repetition that we are going to look at now.

While and do-while

The while loop is a repetitive structure that is used to repetitively execute a specific block of code until a condition is m. This block of code is enclosed in curly brackets and can only be run from within the while loop. We actually use while loops a lot in our daily lives. An example is when you are making soup. While the soup is simmering, you need to stir it to prevent it from sticking to the bottom of the pot, until it’s done. Let’s say you were a robot that understands pseudocode, wed give you an instruction that looks something like this: while soup is not cooked, stir it. Another simple example is when you hold up an umbrella when it starts raining. While it’s raining, you will keep holding up your umbrella until it stops raining.

It probably goes without saying that the loop is controlled by a Boolean statement. We use the word “loop” because the code works in pretty much the same way as an actual loop of, say, a ball of yarn. The yarn will go over and over and over itself until the entire length of the yarn runs out.

The Boolean statement is evaluated whenever the loop “goes over” itself. If you are winding a ball of yarn, you would check if there us still any more yarn to loop onto the ball, then you go ahead and roll it, if there isn’t any, then it means you have reached the end of the length of string, so you drop the ball and move on to doing something else.

The while statement is presented with a condition, every time the loop goes through one iteration, the condition is evaluated, if the condition is true, the program proceeds with that iteration. Typically,

If you run an iteration in the loop, somewhere in the code there is a step that alters the condition that is evaluated at every iteration. After running the block of code, the computer will go back to the condition and re-evaluates it. When we reach a point in the program when the condition is no longer true, the computer ‘drops’ the loop and carries on with the rest of the program, much like we dropped the ball of yarn when we were done winding it in our earlier example.

The variable that is used in the condition is known as the control variable. It is so called because for the loop to start executing, the control must be of a certain desirable value. Also, for the loop to stop executing, the control variable should have been modified so that it no longer satisfies the condition.

Syntax

The syntax for the while loop is fairly simple. As straightforward as we described in the previous section. The syntax looks like this

```
while (<boolean_expression>)
{
 block of code;
};
```

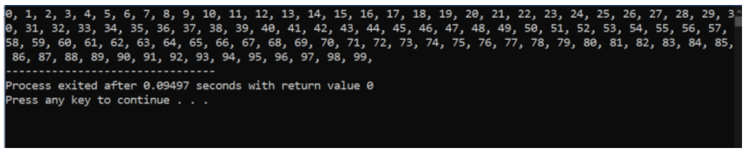
Again, we need to make sure that whatever condition we put in the while statement is a Boolean condition, remember our program is going to test this to see if it is true, and will keep executing till the condition is false.

Let's say, for example, we want to print a list of all numbers that come before the number 100. We would instruct the computer to print a variable, add 1 to it and print it again until we have printed all the numbers we want. The most apparent way of determining if we still need to continue counting would be to check if the value of the last printed number is not 100. Of course, that's not the only way of doing it, but we will go with that for now. You can play around with the code and look at other ways of coming up with this list.

Our code will look like this:

```
int num=0;
while(num<100){
printf("%d, ", num);
num++;
};
```

And if we run it, we get this



```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 3
0, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
--------------------------------
Process exited after 0.09497 seconds with return value 0
Press any key to continue . . .
```

Keep in mind that the code above is a snippet, you still have to add header files and declare the main function to make it run.

In our code snippet, when the computer first encounters the keyword while, it immediately knows that it's entering a loop. It then looks at the condition. When we declared the variable num and initialised it to zero, so the first comparison is 0<100, which is true. the computer then proceeds to print zero, increments the value by 1 and goes back to test the condition. The condition is true until we get to 100<100. The condition is now false, so the computer drops the loop and carries on with the rest of the program. Pretty simple right? The key here is getting the condition and the portion that modifies the condition right, and the rest is a piece of cake!

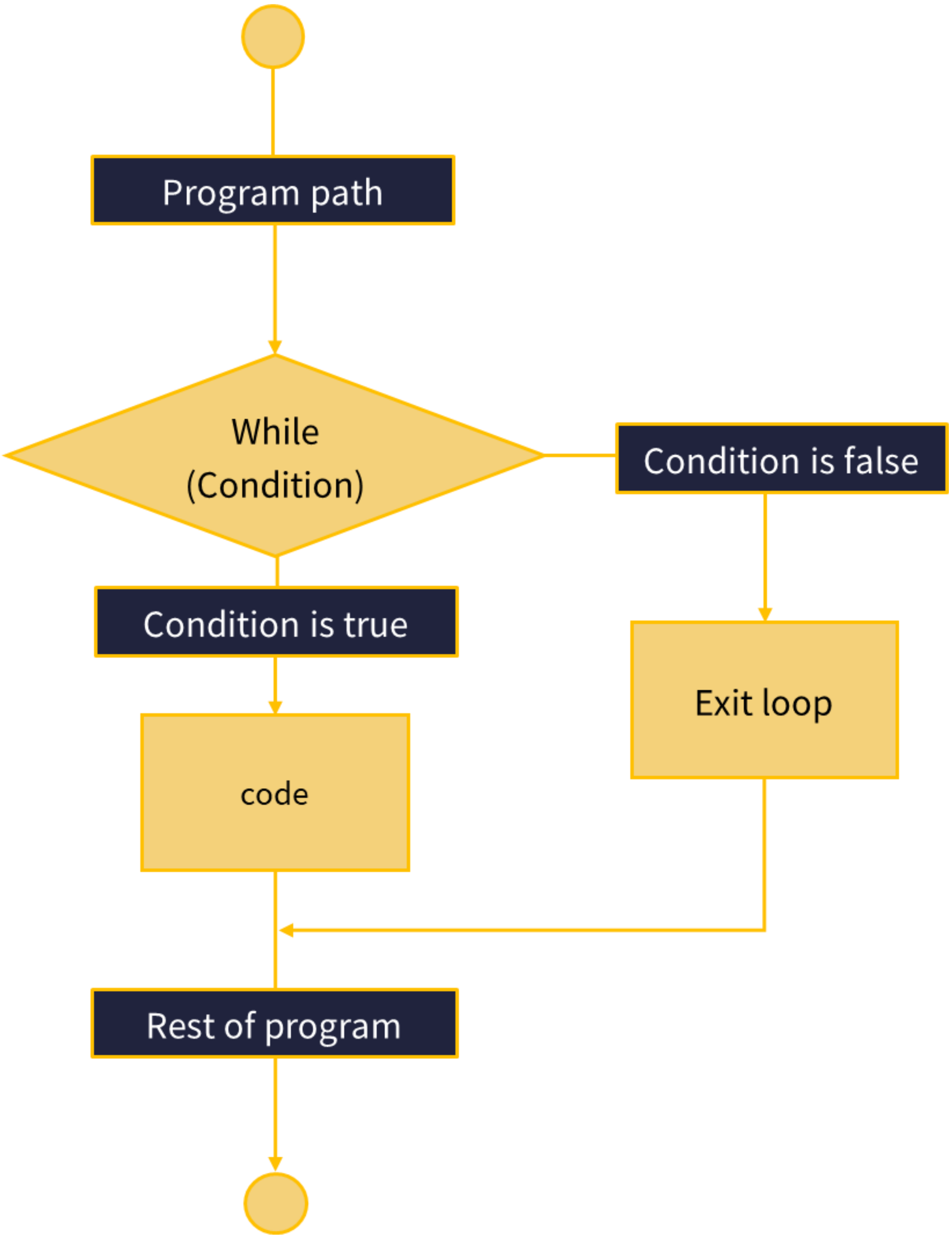
In our examples, we have been checking the condition first, then executing the code. There is also another way of evaluating the while loop, where the computer first executes the code, then checks if the condition is true. this is known as the do-while loop. Not getting the point. Well, sometimes it doesn't make sense to evaluate the condition first; it might be that the condition needs to be modified first from within the loop, then evaluated, or that the condition simply doesn't exist yet, such as if the condition depends on user input.

Let's look at another short example. Let's say you want to read input from the user until the user enters a desired value, say a single digit number. The user enters values, and the computer evaluates the input until it encounters a value that makes the condition false, then exits the loop. Let's fire up our IDE and see how this works

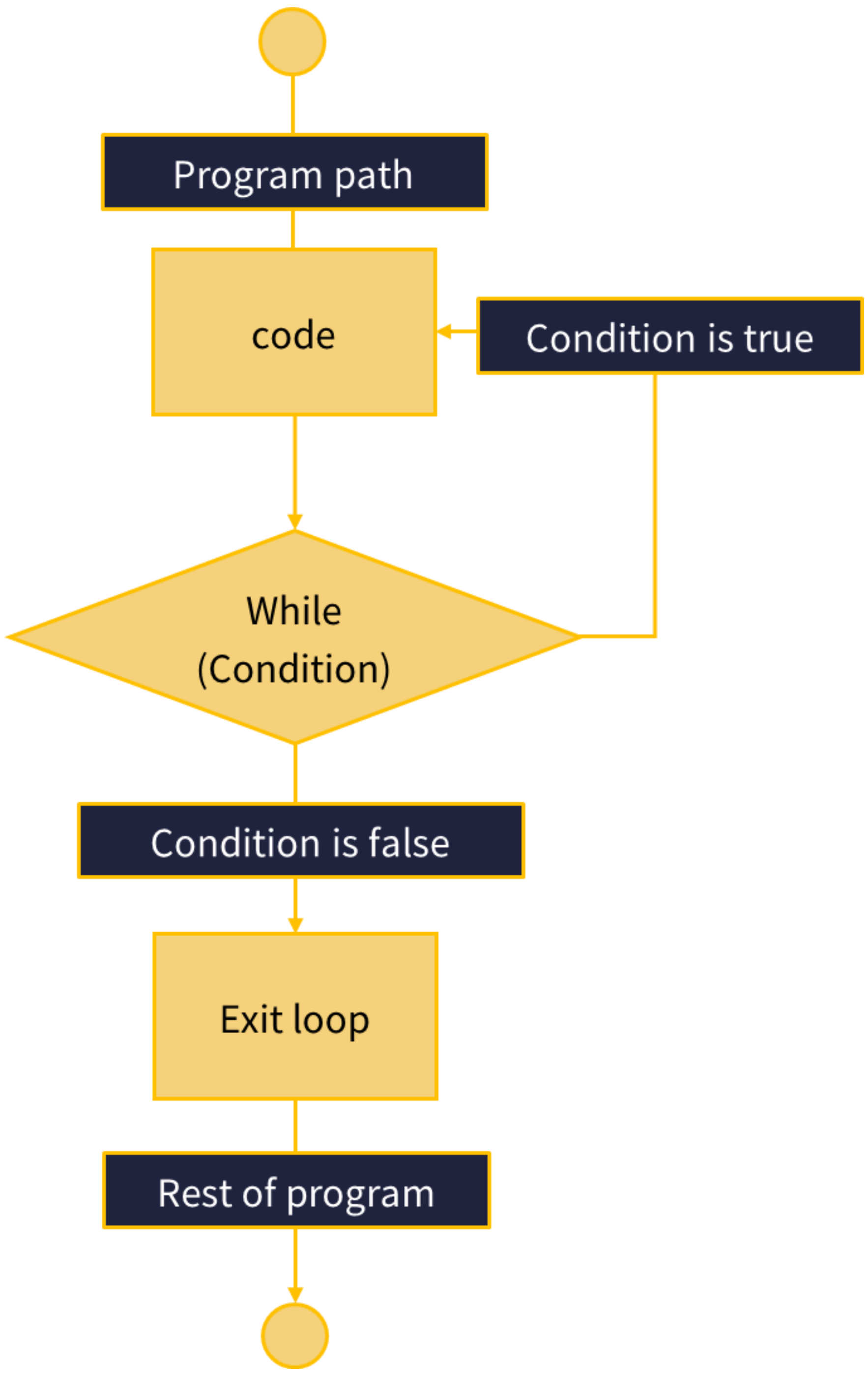
```
int num=0;
        do{
        printf("enter a single digit number to exit the loop\n");
                scanf("%d", &num);
                system("cls");
        }
        while(num>10);
printf("%d is a single digit number. exited loop\n",num);
```

like we described before, the block of code.

Just like in our last lesson, a flowchart will really help us to get a clearer picture of what's going on here. Let's take a look at what the flowchart for the while loop is like.



When the computer encounters the while keyword, it branches away from the default flow of the program and evaluates the condition. If it is true, then the code within the loop is executed. If the condition is false, the computer navigates back to the default path of the program.



The flowchart for the do- while loop is a bit different.

The computer encounters a block of code and executes it. the computer then evaluates a condition. if the condition is true, then the computer repeats the block of code. If the condition is false, then the computer continues with the normal course of the program. The while loop may be executed or may not be executed at all if the initial condition is false, so it is known as an entry-controlled loop. The do-while loop guarantees that the loop is executed at least once.

What to avoid

It’s not all roses and champagne when working with while loops, in fact, it’s pretty easy to land in a trap. The first kind of trap is an unreachable loop. Let’s look at this code snippet. It’s exactly the same as the one we looked at before, except this time we intentionally made a small mistake. I’m going to pause for a few seconds and let you try and find what we changed.

```
int num=0;
while(num>100){
printf("%d, ", num);
num++;
};
```

There’s an itsy bitsy error, the comparison sign is facing the wrong way. This means that if we run the code exactly as it is, there is no way that the value of num is ever going to be greater than 100, so there is no way that the code inside the while loop is going to run, ever! It just becomes a useless piece of code that just sits there. the interesting thing is that it’s such a small error and will probably take a very sharp eye to pick up. Which brings us to rule number 1: Always, ALWAYS make sure that your condition is satisfiable.

The other type of trap is a loop that cannot be exited. Let’s take our earlier example and mutilate it intentionally again.

```
int num=0;
while(num<100){
printf("%d, ", num);
num--;
};
```

This time our error is fairly easy to pick up, but this is not always the case. The code is pretty much correct, but there’s one little mistake; instead of incrementing our variable, we are decrementing it, so the value of num will forever be below 100, and the loop will run and run and run and run and run and run till your system runs out of memory. if your computer has a farm of RAM, then the program will run until your computer looks like that dusty old furniture in the garage, or till lizards evolve back into dinosaurs, or till goodness knows when. The point is, there is no way to exit the loop, so the computer will be stuck in an infinite loop, which brings us to rule number 2: make sure your condition eventually becomes false. You’d be surprised with how dead easy it is to end up in an infinite loop, and the worst part is, you will probably only see it when you execute your program; the compiler is perfectly happy with infinite loops since they don’t break any rules. This is because infinite loops aren’t always a bad thing. There are times when we actually need an infinite loop, such as a program that reads input from the keyboard. For as long as the computer is on, the program needs to run.

Remember when we talked about the importance of initialising variables? Well, in iteration that’s where you really see the importance of initialisation. Imagine your while loop is supposed to check if the value of a variable num is less than 5. Now, you were feeling really lazy because it’s late afternoon and you just want to get the code over and done with, then you decide to save yourself a few button presses and leave the variable uninitialized. When your program is run, the variable is initialised and takes on an unknown value that was already at that memory location. Now, when your program runs, the value happens to be greater than 5, and the while loop never runs. Now you have to debug the code since it’s being inconsistent, yet it’s syntactically and semantically correct.

Which gives us a few additions to our book of rules and laws

Rule number 3: The variables that are used in the loop condition must be initialised when the while loop is first encountered.

Rule number 4: Test the loop control variables in the condition of the loop. The condition must specify the values of the control variables for which the loop must continue repeating, and implicitly define the values for which the loop must terminate.

For loops

The while loop is a great way of running repetitive code. But wait, it gets better, there is another way of running repetitive code. The for loop is also used for running code that needs to be run over and over.

What's the difference?

What's the point of having 2 methods of doing the same thing, especially in programming, where we have already established many times that ambiguity is an unwelcome thing? Well, the two actually serve slightly different purposes.

The while loop is more suited for when the actual number of iterations is not clear. Remember that example when we bugged the user until we got a single digit number? The user could sit there and enter double digit numbers until next year, or simply enter a single digit number the first time. But what if we want 10 numbers every single time, without fail? This is where the for loop comes in. It lets you keep track of the number of iterations by using the control variable as a counter.

The for loop has an initial condition, a final condition, and a modification for the variable. Say for example, we have an array of 10 elements, which we want to fill using input from the user. Of course, we can do this using a while loop without any problem at all, but a for loop will make things straightforward and really readable. It allows you to use less variables too in some cases. In short, the for loop is more commonly used for enumerated repetition and the while loop is more commonly used for unpredictable repetition. The while loop will execute for whatever number of times it takes to accomplish its goal, while the for loop will only execute for the number of steps between the initial condition and the final condition. With a for loop, you know the exact value that you start with, the number of steps that are going to be taken and the final value that will result in the loop being executed. The good thing here is that it's fairly difficult to run into an infinite for loop, since the controls are right there, in your face and do not depend on a different line of code that you can forget and end up with messed up code.

It's considered good programming practice to always gravitate towards for loops whenever you can tell beforehand the number of iterations that are going to performed.

Syntax

As mentioned already, the for loop has 3 components, the initialisation, which is the initial condition, the final condition, and the counter.

- The initialisation step is a one-time occurrence which happens before the loop begins.
- The condition is tested at the beginning of each iteration of the loop.
- If the condition is true, then the body of the loop is executed next.

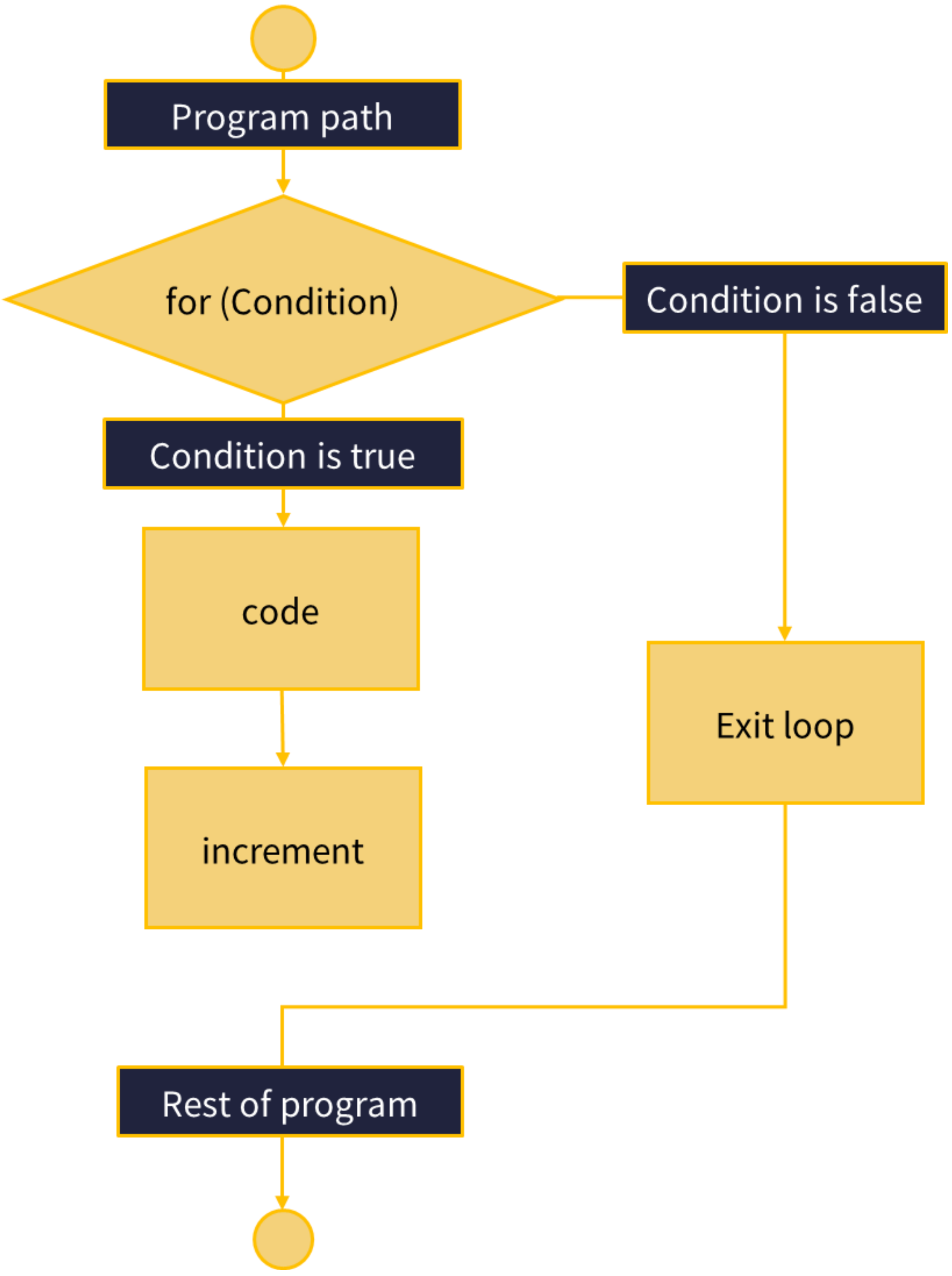
If the condition is false, then the block of code within the for loop is not executed, and execution continues with the code that is after the loop.

The incrementation happens after the execution of the block of code within the for loop, and it only happens if the block of code is executed.

the basic syntax of the for statement looks like this

```
for ( init; condition; increment ) {
   code;
}
```

The flowchart looks like this:



Here's a code snippet that does this task

```
int number, multiple;
printf("Enter a number: \n");
scanf("%d",&number);
printf("The %d multiplication table is:",number);
for (int i = 1; i <= 12; i++)
{
multiple = number * i;
printf(" %d   ",multiple);
```

the program asks the user for a number then uses this number to determine which multiplication series it is going to display. The program goes through exactly 12 iterations, multiplying the value of i by the step number. The program then prints the values, using the same loop that calculates them.

What to avoid

We have already mentioned that the possibility of an unreachable loop or an infinite loop are pretty low when working with for loops. Our emphasis here is counting, and that is where most of our troubles lie. Let's take for example, if your loop has to take exactly 10 iterations. If you set your initial condition as 1 and your execution as <10, it means there will be only 9 iterations instead of 10

Recursion

The process of repetition doesn't just work with variables. You can actually make an entire function run itself over and over, arguments and all! Let's take a look at what recursion is and why we even need it.

What am i looking at?

You may be wondering what we are even talking about here. The concept itself is quite simple, but the code might prove a little bit challenging at first, but you will eventually get the hang of things, it becomes dead simple.

Recursion is when a function calls itself repeatedly. It's worth noting here that C emphasises iteration over recursion, but it doesn't mean that you cand use recursion at all. Recursion allows your program to do much much more!

There are 2 components in a recursive function

The first is the general case, where the solution is expressed in terms of s smaller version of itself. In other words, the problem is made smaller and sent to the calling function.

The base case is the case in which the solution can be stated non recursively. This is where the solution is found. Gosh, this really sounds like a mouthful, but hear me out.

To understand this, lets dust off our mathematics textbooks and look at a concept called the factorial. A factorial is the product of an integer and all the integers below it. It is denoted by n! = n * (n-1) * (n-2) * ... * 1 where n is a positive integer.

For example, 5! =5*4*3*2*1

We can regroup this so that it reads 5*(4!)

Then again 5*(4*(3!))

And again 5*(4*(3*(2!)))

And again 5*(4*(3*(2*(1!))))

And again 5*(4*(3*(2*(1*(0!)))))

And again 5*(4*(3*(2*(1*(1)))))

The first 5 steps in the computation are recursive, and n! Is evaluated in terms of (n-1)! The final step, The final step (0!=1) isn't recursive, but that's the part that we're most interested in this rather elaborate explanation.

If we are to write this down, it would read

N! = {

1 if n = 0 which is the base case, and

N * (n - 1)! If n > 0 which is the recursive step

Here, you will see that in each recursive step we are dealing with a smaller problem by calculating a value that is n-1 times smaller than the previous step. That, ladies and gentlemen, is the power of recursion! Let's demonstrate it using code.

long factorial (int n)

```
{
        long result-0;

        if ( n == 0 )
                result = 1;
        else
                result = n * factorial (n-1);
        return result;
}
```

This code will run the first step. For the sake of simplicity, we will use the same example, 5!

```
Factorial(5)
    ↓                         recursive                  step
    5 * factorial (4)
        ↓                       recursive                step
        4 * factorial (3)
            ↓                     recursive              step
            3 * factorial (2)
                ↓                   recursive            step
                2 * factorial (1)
                    ↓                 recursive          step
                    1 * factorial (0)
                        ↓                base            case
                        1
```

The recursion continues until the base case is reached, and that is determined by the value of n being equal to 0. The function calls to the factorial function are evaluated in reverse order and the last step returns the value 120.

Why do we need recursion?

Why do we even need this head cracking code? Well, there are actually quite a lot of applications. When we were explaining the concept, we used, you guessed it right, mathematics. This is one of the greatest applications. We really have to note, however, that recursion is an expensive way of solving problems. That said, recursion is a much more intuitive way of solving problems, and sometimes convert the solution to the iterative equivalent in situations where every last bit of performance counts. There are applications in mathematics that depend on recursion, such as exponents and factorials, like the example that we just walked through.

Recursion is the basis of divide and conquer algorithms. The most common example of this is the Merge Sort, which recursively divides an array into single elements that are then "conquered" by recursively merging the elements together in the proper order. Other examples that we shall see later in this course, such as linked lists, use recursion for inserting data into the list

Another application of recursion is computer graphics. Fractals and Koch's snowflake are examples of computer graphics that use recursion.

In artificial intelligence, problems such as the eight queen's problem and maze generation are examples of problems that use recursion.

Best coding practices

When using recursion, there are a few rules that you need to adhere to. Let's go through them.

Base cases: there must always be some base or trivial case, which can be solved without recursion.

Making progress: For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward the base case.

Design rule: Assume that all the recursive calls work. Use proof by induction, which is a mathematical concept that tests if a problem can be solved.

Compound Interest Rule: Never duplicate work by solving the same instance of a problem in separate recursive calls.

Famous examples

To close off the lesson, we will look at the Fibonacci sequence using recursion.

```
#include<stdio.h>
int Fibonacci(int);
int main()
{
   int n, i = 0, c;

   scanf("%d",&n);
   printf("Fibonacci series\n");
   for ( c = 1 ; c <= n ; c++ )
   {
      printf("%d\n", Fibonacci(i));
      i++;
   }
   return 0;
}
int Fibonacci(int n)
{
   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Conclusion

We have come to the end of our lesson, where we explored repetition as a tool for accomplishing the tasks that humans never seem to do accurately, we looked at the various techniques, such as while loops, for loops and recursion. We threw a bunch of examples in the mix too. In our next lesson, we are going to look at storage classes. We will look at how these are used to limit the scope of variables. Ill also take this time to encourage you to write code as much as you can. Not only does it cement programming concepts, but it also allows you to understand computer science better.

References

Emory.edu. (2020). [online] Available at: http://www.mathcs.emory.edu/~cheung/Courses/561/Syllabus/2-C/while.html

Khan Academy. (2018). Recursion (article) | Recursive algorithms | Khan Academy. [online] Available at: https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursion#

K-state.edu. (2010). 5.7. Recursion — Applications in C for Engineering Technology. [online] Available at: http://faculty.salina.k-state.edu/tim/CMST302/study_guide/topic4/recursion.html

Princeton.edu. (2014). Conditionals and Loops. [online] Available at: https://introcs.cs.princeton.edu/java/13flow/

The while Loop and Practice Problems Use. (n.d.). [online] Available at: http://www.bowdoin.edu/~ltoma/teaching/cs107/spring05/Lectures/while.pdf

Uic.edu. (2020). C Programming Course Notes - Looping Constructs. [online] Available at: https://www.cs.uic.edu/~jbell/CourseNotes/C_Programming/Looping.html

Unisa.edu.au. (2016). Sign In. [online] Available at: https://lo.unisa.edu.au/mod/book/tool/print/index.php?id=466679

Uregina.ca. (2020). Recursion. [online] Available at: https://www.cs.uregina.ca/Links/class-info/210/Recursion/

Utah.edu. (2020). Programming - While Loop. [online] Available at: https://www.cs.utah.edu/~germain/PPS/Topics/while_loops.html