Repetitions/Iterations

Recall that everytime the user enters an invalid input, we simply prompt the user about it and the program terminates. How do we allow the users to re-enter the inputs instead of simply terminating the program?

You have seen how this was done with flowcharts. And yes, it was via the use of a loop construct. In c, the loop construct is implemented using 3 different iterative statements. And we look at them one by one beginning with the `while` statement.

while statement

The `while` statement has 3 components as shown below.

```
while(<condition>)
    statement;
```

The code snippet above shows that it looks exactly like the if-statement you've met in the previous section. This time, instead of using the keyword `if` , `while` replaces it. This means that statement will be executed many times over, so long as the condition in the `while` evaluates to true. This is the exact opposite with the flowcharts we were working on with Raptor where the iteration is executed when the condition evaluates to false or answers no.

The statement in the `while` could be any valid c statement like expression statements, if-statements and all its variations, and what do you know, loops themselves!

The version below does exactly the same thing as described above but this time it repeatedly executes multiple statements, hence the use of the curly braces to set the scope of iteration.

```c
while(<condition>){
    statement1;
    statement2;
    statement3;
    ...
    statementN;
}
```

Let's apply to this to a simple counting problem. Let's print the numbers from 1 – 10000. Although this can be achieved by without using any loop construct, imagine how incovenient it would be to code 10000 `printf` statements. Since the printing has to be repeated many times over, 10000 times to be exact, a loop construct is in order. As with the loop construct in flowcharts, a `counter` has to be introduced. This `counter` starts from 1 and goes all the way to the upper limit 10000. Naturally, the `counter` has to be incremented so the next item or number gets printed. To check whether the counter has reached the desired limit, a condition is utilized. Check the source code below.

```c
#include <stdio.h>

int main(){
    int counter = 1;

    while(counter <= 10000){
        printf("%i ",counter);
        counter++;
    }

    return 0;
}
```

We can even extend this by getting the upper limit from standard input.

```c
#include <stdio.h>

int main(){
    int counter = 1, upperLimit;

    printf("Enter the upper limit: ");
    scanf("%i", &upperLimit);

    while(counter <= upperLimit){
        printf("%i ",counter);
        counter++;
    }

    return 0;
}
```

The same print from 1 – 10000 can also be simulated if we count from `upperLimit` down to 1.

```c
#include <stdio.h>

int main(){
    int counter = 1, upperLimit;

    printf("Enter the upper limit: ");
    scanf("%i", &upperLimit);

    while(upperLimit > 0){
        printf("%i ",counter);
        counter++;
        upperLimit--;
    }

    return 0;
}
```

And we if we want to print the values in reverse, i.e. 10000 – 1, we can do that as well.

```c
#include <stdio.h>

int main(){
    int upperLimit;

    printf("Enter the upper limit: ");
    scanf("%i", &upperLimit);

    while(upperLimit > 0){
        printf("%i ",upperLimit);
        upperLimit--;
    }

    return 0;
}
```

Summation problem

Recall the summation problem from flowcharts. This is computing for the sum starting from 1 all the way to some upper bound n (user input). A variable that iterates from 1 to the the upper bound n has to be declared. We are going to call this, iter (short for iterator as it should iterate from 1 all the way to n). And finally, a *sum* where the result of the additions should be stored.

```c
int n = 0, iter = 1, sum = 0;
```

Next step is to set up the `while`. A condition (relational expression), must be constructed. Again, the loop is going to iterate only when the condition evaluates to true. And since what we want is for *iter* to reach n, and seeing that n is the upper bound, this tells us *iter* will be lesser than the n. So long as this truth holds, *iter* gets incremented but only until it reaches n.

```c
while(iter <= n){
    iter++;
}
```

When *iter* reaches n, one last iteration is executed. This will bring *iter* equal to $n + 1$. When this happens, *iter* is NO LONGER $<= n$. The loop or iteration terminates!

So far, The solution above does half the trick! As you expected, we are not done yet. We have not computed for the *sum* yet. *sum* needs to accumulate value at for each repetition. By how much should *sum* accumulate? You guessed right!

Whatever *iter* is! This is correct because, remember, *iter* begins in 1, then increments to 2, then 3, all the way to n. The exact definition of summation!

```c
sum = sum + iter;
```

Now, let's put together the solution!

```c
#include <stdio.h>

int main(){
    int n = 0, iter = 1, sum = 0;

    printf("Enter the upper bound: ");
    scanf("%i",&n);

    while(iter <= n){
        sum = sum + iter;
        iter++;
    }

    printf("The summation from 1 to %i is %i.",n,sum);
    return 0;
}
```

It is important to enclose `iter++` and `sum = sum + iter` in the `while` with the use of the pair of curly braces. Had we not done that, as shown below:

```c
#include <stdio.h>

int main(){
    int n = 0, iter = 1, sum = 0;

    printf("Enter the upper bound: ");
    scanf("%i",&n);

    while(iter <= n)
        sum = sum + iter;
        iter++;

    printf("The summation from 1 to %i is %i.",n,sum);
    return 0;
}
```

only `sum = sum + iter` gets iterated and executed many times over. `iter++`, even if it is indented to align with the accumulation of *sum*, will not form part of the entire iteration. We say the `iter++` is an unreachable statement. And as a result, *iter* never changes value. All throughout the iteration *iter* is always 1. And as a consequence, the iteration will loop infinitely many times.

Alterntively, we can begin the summation from n down to 1. This is correct because addition is commutative (`x + y` is the same as `y + x`).

```c
#include <stdio.h>

int main(){
    int n = 0, iterDown = 0, sum = 0;

    printf("Enter the upper bound: ");
    scanf("%i",&n);

    iterDown = n;

    while(iterDown > 0){
        sum = sum + iter;
        iterDown--;
    }

    printf("The summation from 1 to %i is %i.",n,sum);
    return 0;
}
```

Power Problem

The power problem was introduced in one of the video lectures on iterations/repetitions using flowcharts. But we will discuss it here again.

Recall that we have 2 positive inputs here, the base and the exponent. And a series of multiplications are executed to get the desired result.

```c
#include <stdio.h>

int main(){
    int base, exp, iter = 1, ans = 1;

    printf("Enter the upper bound: ");
    scanf("%i%i",&base,&exp);

    while(iter <= exp){
        ans = ans * base;
        iter++;
    }

    printf("%i^%i is %i.",base,exp,ans);
    return 0;
}
```

Compare how *sum* (from summation) and *ans* are initialized. `sum = 0` while `ans = 1`. The reason for this is because of the series of multiplications that have to be executed. When `ans` is set to 0, no matter many times `ans = ans * base` is executed, `ans` will always be 0. We say that 0 is the additive identity and 1 is the multiplicative identity.

do-while Statement

Recall that we introduced this section to solve the problemm of handling invalid inputs. Whenever the user enters an invalid input, we are to prompt them to re-enter the input. (Whew! That was a lot of input!)

Remember the kilometer to meter conversion problem? What we want to simulate here is whenever the user enters a non-positive value, we are to prompt them to re-enter a positive kilometer measurement.

This should be straightforward. All we need to do is the `while` construct and set the condition to `km < 1`.

```c
while(km < 1){
    printf("Enter a positive kilometer measurement: ");
    scanf("%i",&km);
}
```

The code snippet above requires that `km` should already have a value. There are two ways we can do that. One is to ask for `km` prior to the loop.

```
printf("Enter a positive kilometer measurement: ");
scanf("%i",&km);
while(km < 1){
    printf("Enter a positive kilometer measurement: ");
    scanf("%i",&km);
}
```

There should be one glaring observation here. There are eaxctly 2 copies of the `printf` and the `scanf`.

The other solution is to initialize `km` to a value that will force the condition in the `while` to evaluate to true.

```
int km = 0;
while(km < 1){
    printf("Enter a positive kilometer measurement: ");
    scanf("%i",&km);
}
```

It is easy to see that since `km` is 0, it is definitely less than 1.

For this simple problem, it is easy to figure out what possible initial values to assign to our input variables. But for other problems, this may not be as easy. Again, we go to all this trouble just so the initial/first check on the condition of the loop evaluates to true.

What we really want here is to repeatingly ask the user for the input as long as it is invalid. But the kind of repition we want is one that guarantees that the loop will execute at least once!

The `while` construct does not guarantee that. Since it checks the condition first before proceeding with the iterative process, there is a chance that the iteration might not be executed at all. This is exactly what we had with the flowcharts. Recall this.

There is a loop construct in c that exactly does this and it is the aptly named the `do-while` statement. It is aptly named because it will the statements first, before checking the condition.

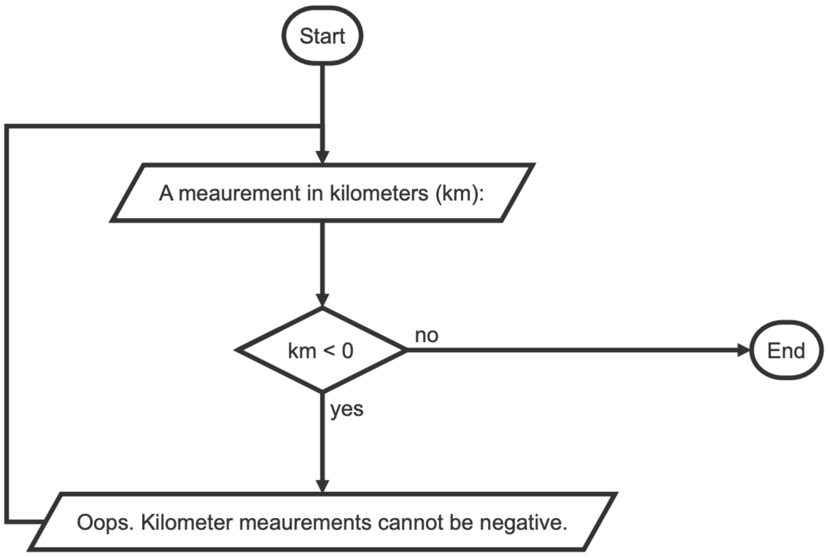
Start
A meaurement in kilometers (km):
km < 0
no
End
yes
Oops. Kilometer meaurements cannot be negative.

Figure 1: Kilometer input measure check flowchart

```
do{
    statement1;
    statement2;
    statement3;
    ...
    statementN;
}while(<condition>);
```

Input Validation

Using the `do-while` will execute the statements for, before checking the condition. This will guarantee that the statements that need to be repeated many times will be executed at least ONCE.

```c
#include <stdio.h>
int main(){
    int km;

    do{
        printf("Enter a positive kilometer measurement: ");
        scanf("%i",&km);
    }while(km < 1);

    return 0;
}
```

We didn't have to figure out an initial value for `km`. And we didn't have to do `printf` and `scanf` twice!

We can now apply this to the summation problem especially because the upper bound should be greater than or equal to 1. If the user will enter 0, it does not pose any problem with the loop (the loop will not be executed since `iter` is not less than nor is it equal to 0) except for saying "The summation from 1 to 0 is 0." The same behavior can be observed when the user enters a negative integer. The program will say "The summation from 1 to 0 is n." where n is some integer less than 0.

```c
#include <stdio.h>

int main(){
    int n = 0, iter = 1, sum = 0;

    do{
        printf("Enter the upper bound: ");
        scanf("%i",&n);
    }while(n < 1);

    while(iter <= n){
        sum = sum + iter;
        iter++;
    }

    printf("The summation from 1 to %i is %i.",n,sum);
    return 0;
}
```

Without the check, the same can be observed with the down-to-1 version of the solution.

```c
#include <stdio.h>

int main(){
    int n = 0, iterDown = 0, sum = 0;

    do{
        printf("Enter the upper bound: ");
        scanf("%i",&n);
    }while(n < 1);

    iterDown = n;

    while(iterDown > 0){
        sum = sum + iter;
        iterDown--;
    }

    printf("The summation from 1 to %i is %i.",n,sum);
    return 0;
}
```

Body Mass Index Version 3.0

```
```

More on Iteration

Factors

Check the source code below:

```
#include <stdio.h>

int main(){
    int num, cf = 2;

    do{
        printf("Enter a positive integer: ");
        scanf("%i",&num);
    }while(num < 1);

    printf("1 %i ",num);
    while(cf <= num/2){
        if(num%cf == 0)
            printf("%i ", cf);
        cf++;
    }
    printf("\n");
    return 0;
}
```

The program prints the factors of a given number including the factor 1 and the number itself. This is exactly one of the 3 versions we had with flowcharts.

When *num* is 1 million, how many times will the loop iterate? Half of 1 million less 1, which 499999! That's a lot of iterations!

Let's reduce the value of a to 100, how many times will the loop iterate? Yes, 49 times! cf will go from 2 to 50! When looking for factors, it is not necessary to check all values from 2 to half the number! We have explained this with flowcharts but we will reiterate it.

There are definitely far fewer factors of a number than there are values from 1 to the number! Think of 100. How many factors does 100 have? Let's list them but let's exclude 1 and 100 itself because we know these as factors by default. They are 2, 4, 5, 10, 20, 25, 50. That's it! Definitely significantly fewer than 100 values!

Excluding itself, what is the biggest possible factor of a given number? Right! Half the number! Exactly the reason why we set the upper limit for the candidate factor to `num/2`. But that's still a lot of numbers! Again, as highlighted above, imagine if the number were 1,000,000! 499,000 is a lot! It is!

And explained in Chapter 1, we do not really need to reach 50, because if we know 2 is a factor of 100, then so is 100/2. Always remember that factors come in PAIRS!

Let's do that again! Let's start listing them by pair!

- 1 and 100
- 2 and 50
- 4 and 25
- 5 and 20
- 10 and 10 (well technically, just 10)

At which value did we stop looking for factors? 1. And what is 10 in relation to 100? Perfect! It is the square root of 100 (10 * 10 or 10^2)!

Now, let's have a look at the code below and verify if we have made some progress.

```
#include <stdio.h>

int main(){
    int num, cf = 2;

    do{
        printf("Enter a positive integer: ");
        scanf("%i",&num);
    }while(num < 1);

    printf("1 %i ",num);
    while(cf * cf <= num){ //looks for factors all the way
     ↪   to the square root of num
        if(num%cf == 0)
            printf("%i %i", cf, num/cf); //this is printing
 ↪   the pair, if cf is a factor then so is num/cf
        cf++;
    }
    printf("\n");
    return 0;
}
```

As an exercise, trace the code above when num is 100.

Primality Check

We have defined prime numbers in Chapter 1 (Primeality Chekc Video Lecture Using Raptor) as positive numbers whose factors are 1 and itself only, and these two have to be unique values. Hence, 1 is not prime and the smallest prime number is 2. Its only factors are 1 and 2 (itself). The first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, . . . And we have mentioned in Chapter 1, that the prime numbers do not have a general pattern.

The source code below, should be reminiscent of our solution to the prime problem with flowcharts.

```c
#include <stdio.h>

int main(){
    int num, cf = 2, prime = 1;

    do{
        printf("Enter a positive integer: ");
        scanf("%i",&num);
    }while(num < 1);

    while((cf * cf <= num) && (prime == 1)){ //looks for
    ↪  factors all the way to the square root of num
        if(num%cf == 0)
            prime = 0;
        cf++;
    }
    if(prime == 1)
        printf("%i is prime.\n");
    else
        printf("%i is not prime.\n");

    return 0;
}
```

While we're here, check the primality check revision below and observe the `if(prime)`.

```
#include <stdio.h>

int main(){
    int num, cf = 2, prime = 1;

    do{
        printf("Enter a positive integer: ");
        scanf("%i",&num);
    }while(num < 1);

    while(cf * cf <= num){  //looks for factors all the way
     ↪  to the square root of num
        if(num%cf == 0){    //verifies if cf is a factor
            prime = 0;      //sets prime to false
            break;
        }
        cf++;               //goes to the next candidate
 ↪  factor
    }

    if(prime)               //checks if prime is true
        printf("%i is prime.\n");
    else
        printf("%i is not prime.\n");

    return 0;
}
```

This will work correctly. Recall again that in c, all non-zero values are considered true. And zero is considered false.

There should be a noticeable difference with what we have here in the condition as opposed to what we used with the flowchart solution using Raptor.

Recall that *prime* here is used as a flag. Initially we assume that *num* is prime, and therefore *prime* is set to 1. Again recall that in c, any non-zero value is considered true, and zero considered false.

For the raptor flowchart the condition is `cf * cf > num OR prime == 0)` and for the c code it is `cf * cf <= num && prime == 1`. The difference is mainly because of how loops operate in raptor vs how loops behave in c. In raptor, the loop will iterate whenever the condition evaluates to false (we put the condition that will terminate the loop). And in c, the loop will iterate whenever the condition evaluates to true. They are exact opposites, so to speak. It should now be easy to see that for the raptor flowchart, the loop should stop when the

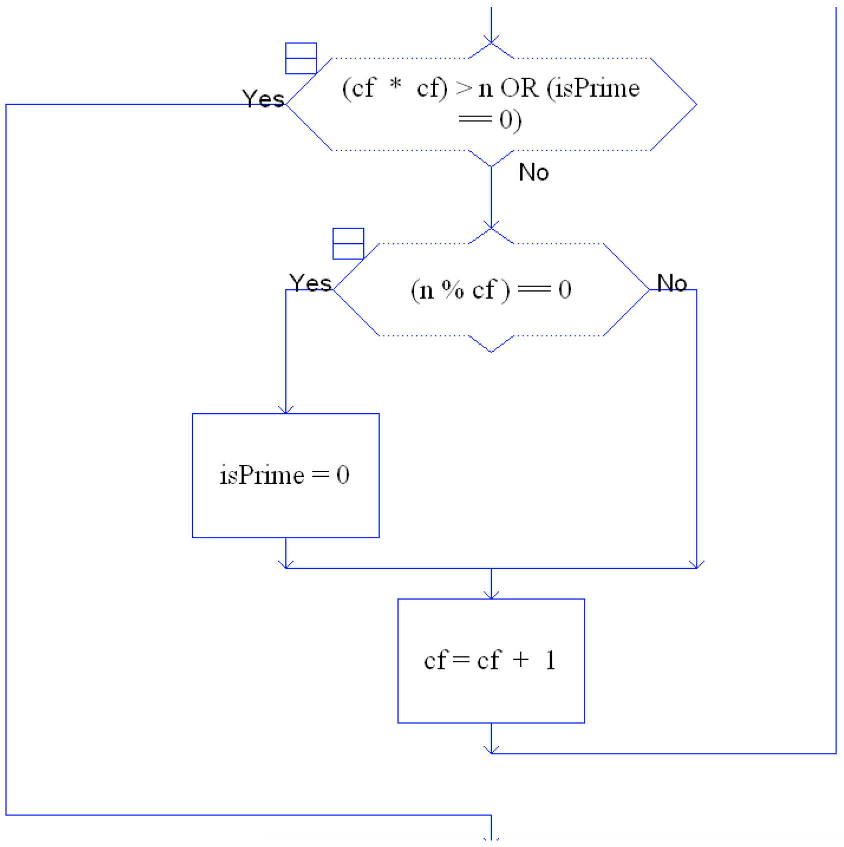

Figure 2: Primality Check Flowchart Snippet

candidate factor is greater than the square root of `num` or `prime` is 0 - this means a new candidate factor has been verified to be a factor.

For the solution in c, because the loops in c iterate whenever condition evaluates to true, we cannot use the same condition as in the raptor flowchart. If we used the same condition as shown below:

```
while((cf * cf > num) || (prime == 0)){ //looks for factors
↪   all the way to the square root of num
    if(num%cf == 0)
        prime = 0; //this is printing the pair, if cf is a
↪   factor then so is num/cf
    cf++;
}
```

the loop never iterates. *cf* is initially 2 and *num*is almost always bigger than *cf*. And *prime* is initially 1. So both checks evaluate to false, and exits from the loop even on the first check!

There's actually a 3rd logical operator and that is the NOT logical operator denoted by the bang (!). The NOT is a unary logical operator. This means that it only needs one operand. Below is the truth table for NOT.

`A`	`!A`
True	False
False	True

The NOT operator is used to negate boolean values or the results of relational operations (which are boolean values). Recall that in c, any non-zero value is considered TRUE, and zero means FALSE. There is no data type called `boolean`. This means there no literal values `true` and `false` in c. They are simulated via the use of an integer.

The samples below demonstrate :

```
!(5 < 10)
!(num >0)
!(num == y)
!(cf * cf > num || prime == 0)
```

Let's look at them one by one.

- `!(5 < 10)`, `!(num > 0)`, and `!(num == y)`

    - 5 is always less than 10. So, `5 < 10` will evaluate to TRUE. Negating this result, `!(5 < 10)` will ultimately evaluate to FALSE. This is essentially **not less than**, so **use** `>=` **instead**. `!(num > 0)` is essentially **not greater than**, so **use** `<=` **instead**. `!(num == y)` is essentially **not equal**, so **use** `!=` **instead**.
- `!(cf * cf > num || prime == 0)`
    - we will go back to this in a while.

Recall the check on whether the character is a digit?

```c
#include <stdio.h>

int main(){

    char digit;

    printf("Enter a character: ");
    scanf("%c",&digit);

    if(digit >= '0' && digit <= '9')
        printf("%c is a digit.\n",digit);
    else
        printf("%c is not a digit.\n",digit);

    return 0;
}
```

Let's revise the problem, just a bit. When the character is NOT a digit, we print it is not. And when it is, we DO NOT DO ANYTHING.

```c
#include <stdio.h>

int main(){

    char digit;

    printf("Enter a character: ");
    scanf("%c",&digit);

    if(digit >= '0' && digit <= '9');
    else
        printf("%c is not a digit.\n",digit);

    return 0;
}
```

Recall that part of the `if statement` is at least an **expression statement** that has to be executed whenever the condition evaluates to TRUE. And recall that we have defined an **expression statement** to be an optional expression followed by a `;`. Optional expression means we can opt to not to have them as show above.

`if(digit >= '0' && digit <= '9');` means that when the character is a digit, the program will NOT DO ANYTHING. The reason why we might want to do this is probably are having a hard time figuring out what to put in the condition. Because there tons of characters that are NOT DIGITS. Are we going to list them one one by in the condition? Of we course, we are NOT.

Instead of doing this, use the NOT operator as shown below.

```c
#include <stdio.h>

int main(){

    char digit;

    printf("Enter a character: ");
    scanf("%c",&digit);

    if(!(digit >= '0' && digit <= '9'))
        printf("%c is not a digit.\n",digit);

    return 0;
}
```

By doing this, the program simply checks if the character is a digit, and then negates that result of the check. And it is readable. NOT digit.

Always be careful when the seemingly innocent `;`. Complext programs and systems break because of innocent mistakes like this. Check the code below.

```c
#include <stdio.h>

int main(){
    int counter = 1;

    while(counter <= 10000);
        counter++;

    return 0;
}
```

Try to compile and run this. You will notice a blinking cursor does seemingly does nothing. But does something. It repeatingly checks whether `counter` is less than 10000. AND IT NEVER ENDS! Like the `if`, the entire `while statement` requires at least an expression statement where the expression may be optional. Hence, this will compile and not produce any errors. Since `coounter++` is now an unreachable statement, *counter* never gets updated! And the loop will execute infinitely many times.

SO BE CAREFUL WITH MISTAKES LIKE THIS.

Let's back to `!(cf * cf > num || prime == 0)`. Let's check its truth table. Let A be `cf * cf > num` and we let B be `prime == 0`.

`A`	`B`	`A || B`	`!(A || B)`
True	True	True	False
True	False	True	False
False	True	True	False
False	False	False	True

When are the instances that the entire check `!(cf * cf > num || prime == 0)` evaluates to true? Only when both `cf * cf > num` AND `prime == 0` are FALSE. When will `cf * cf > num` be false? When `cf * cf <= num`. When will `prime == 0` be false? When `prime != 0` or to be specific to the problem of primality check, when `prime == 1`. And this is exactly `cf * cf <= num && prime == 1`!

Whew! What we discovered here is `!(A || B)` is equivalent to `!A && !B`. This should be familiar. Yes, this is de Morgan's Law!

break and continue

The reason why we included the check on whether *prime* is 1 in condition of the `while statement` is to provide a mechanism to break from the iteration. Yes, break. Or a way to stop the loop. Once a candidate factor evenly divides num, *prime* is set to 0. So, even if `cf * cf > num` still holds, the check `prime == 1` will fail. And since TRUE && FALSE is FALSE, the loop breaks or stops.

There is a c construct that we can use to force loops to terminate or stop without relying on the loop condition. This construct is aptly, again, named `break`. Using the `break` force the loop to terminate.

Compile and run the source code below.

```c
#include <stdio.h>

int main(){

    while(1 < 1000000){
        break;
        printf("1 is less than 1000000\n");
    }

    printf("This is outside and after the loop now.\n");
    return 0;
}
```

Notice that even if `1 < 1000000` holds, not one "1 is less than 1000000" is printed. Instead, only "This is outside and after the loop now." was printed. This means that when the execution of the program checked the condition, execution proceeded to the `break` statement. And this has forced the loop to stop.

Just so you are convinced further, try the source code below.

```c
#include <stdio.h>

int main(){
    int num = 1;
    while(num < 1000000){
        num = 10;
        break;
        num = 100;
        printf("num is less than 1000000\n");
    }

    printf("This is outside and after the loop now. num is
↪   %i.\n",num);
    return 0;
}
```

When num is printed outside the loop, it till prints 10 and not 1 (this tells us that, indeed, the execution of the program went inside the loop) and not 100 (this tells us that the execution did not proceed to `num = 100`.

Of course, the use of `break` above is not useful. If this were the intention, then we would not have used a loop at all. The `break` is often used with `if statements` (it is also used with `switch` – this is discussed in a video lecture).

Let's apply the `break` in the solution to the primality check problem.

```c
#include <stdio.h>

int main(){
    int num, cf = 2, prime = 1;

    do{
        printf("Enter a positive integer: ");
        scanf("%i",&num);
    }while(num < 1);

    while(cf * cf <= num){ //looks for factors all the way
    ↪   to the square root of num
        if(num%cf == 0){
            prime = 0;
            break;
        }
        cf++;
    }
    if(prime == 1)
        printf("%i is prime.\n");
    else
        printf("%i is not prime.\n");

    return 0;
}
```

There's also the `continue` construct in c. Check the code below:

```c
#include <stdio.h>

int main(){
    int num = 1;
    while(num < 10){
        num++;

        if(num == 7)
            continue;
        printf("%i ", num);
    }

    return 0;
}
```

Like the use of `break`, using `continue` will not execute the `printf` statement. The execution of the program will skip all statements that come after `continue`. But unlike `break`, `continue` will direct the flow of the execution back to the condition check of the `while statement`. If the condition still holds, then iteration or loop proceeds. In the example above, 7 will not be printed.

for statement

There is another loop construct in c called the for statement. It achieves the same thing as the while, executing statements repeatedly every time the condition holds. Here is how it looks like:

```
for(<initializer>;<condition>;<iterator>)
    statement;
```

or

```
for(<initializer>;<condition>;<iterator>){
    statement1;
    statement2;
    statement3;
    ...
}
```

The **statements and the condition** that you see in the `for` above are the same statements and condition used in `while` statements. The **initializer** is place holder for setting the initial values of the iterators, and other identifiers or variables that need initialized. This is executed only once. Even when the loop will iterate a million times, the initializer will be executed once, hence its name.

The **iterator** is the place holder for statements when executed eventually lead to breaking the loop sometimes referred to as **control variables**. Control variables are essentially those used in the condition to control the number of repetitions executed by the loop. Other forms of iterators including all forms of accumulation are also allowed to be put there. Generally, the purpose of the iterator part is to perform updates on the variables used in the loop.

The initiliazer, condition, and iterator are separated by a semi-colon.

If we were to convert the `for` statement format above to its equivalent in `while`, it would look like the one below:

```
<initializer>;

while(<condition>){
    statement;
    <iterator>;
}
```

Let's use the for statement in computing for the summation.

```c
#include <stdio.h>
int main(){
    int iter, n, sum;

    do{
        printf("Enter a positive upper bound: ");
        scanf("%i", &n);
    }while(n < 1);


    for(iter = 1, sum = 0; iter <= n; iter++)
        sum = sum + iter;

    printf("The summation from 1 to %i is %i.\n",n,sum);

    return 0;
}
```

This is how the for-loop works. First, the initializer gets executed. When you need to do multiple initializations, simply comma-separate them. Then the control of the execution goes to the condition check. When this evaluates to false, the for-loop, like the while-loop, stops. When the condition evaluates to true, control of the execution goes to the statement `sum = sum + iter`. Control of the execution goes to the statements found in the iterator part of the for-loop, which, in this case, is iter++. Then execution goes back to the condition and checks whether it still evaluates to true. This repetition stops when the condition finally evaluates to false.

We mentioned that in the iterator, forms of accumulation are allowed as well. Check the source code below.

```c
#include <stdio.h>
int main(){
    int iter, n, sum;

    do{
        printf("Enter a positive upper bound: ");
        scanf("%i", &n);
    }while(n < 1);


    for(iter = 1, sum = 0; iter <= n; iter++,sum+=iter);

    printf("The summation from 1 to %i is %i.\n",n,sum);

    return 0;
}
```

Similar to the initializer, when a number of statements have to be executed in the iterator part, comma-separate these statements as show above. Similar to the `if` and the `while` statements, the expression statement of the `for` may be optional as well. Since there are no other statements have to executed by the `for`, you see the awkward semi-colon terminating the `for` statement.

For a more readable code, avoid doing this.

The initializer and iterator maybe left blank. And if we did that, the `for` statement will look like the one below:

```c
#include <stdio.h>
int main(){
    int iter = 1, n, sum = 0;

    do{
        printf("Enter a positive upper bound: ");
        scanf("%i", &n);
    }while(n < 1);


    for(;iter <= n;){
        sum = sum + iter;
        iter++;
    }
    printf("The summation from 1 to %i is %i.\n",n,sum);

    return 0;
}
```

Inspecting the solution above, except for the required semi-colons that separate the initializer, condition, and iterator, the `for` pretty much looks like the while.

Actually, we can even leave the condition blank as well.

```c
#include <stdio.h>
int main(){
    int iter = 1, n, sum = 0;

    do{
        printf("Enter a positive upper bound: ");
        scanf("%i", &n);
    }while(n < 1);


    for(;;){
        sum = sum + iter;
        iter++;
        printf("What is happening?\n");
    }
    printf("The summation from 1 to %i is %i.\n",n,sum);

    return 0;
}
```

Blank conditions in `for` statements, by default, evaluate to true. Hence, the infinite loop. Of course, you could include `break` statements inside the `for`.

```c
#include <stdio.h>
int main(){
    int iter = 1, n, sum = 0;

    do{
        printf("Enter a positive upper bound: ");
        scanf("%i", &n);
    }while(n < 1);


    for(;;){
        sum = sum + iter;
        iter++;
        if(iter > n)
            break;
    }
    printf("The summation from 1 to %i is %i.\n",n,sum);

    return 0;
}
```

Leaving the condition of the `for` statement blank is not a good idea. Infinite loops are useful in server applications where server programs have to *listen* for connection requests infinitely. You will learn a great deal on Data Communications between computers in CMSC 137 in your senior year.

Checkout the solutions to the factorial and fibonacci problems using the `for` statement.

factorial using for

```c
#include <stdio.h>

int main(){
    int n, ans,iter;

    do{
        printf("Enter a positive upper bound: ");
        scanf("%i", &n);
    }while(n < 1);

    for(iter=n, ans=1; iter>0; iter--)
        ans = ans * iter;

    printf("The factorial of %i is %i.\n",n,ans);
    return 0;
}
```

fibonacci using for

```c
#include <stdio.h>

int main(){
    int n, a, b, c,iter;

    do{
        printf("Enter a positive upper bound: ");
        scanf("%i", &n);
    }while(n < 1);

    for(a=1,b=0,iter-0; iter<n; iter++){
        c = a + b;
        a = b;
        b = c;

        printf("%i ",c);
    }

    printf("\n");
    return 0;
}
```