Topic 4 – Iteration

Fast Waiting?

- Computers are not just good at following instructions.
  - They're good at following instructions FAST.

- But consider how long it takes for a programmer to write these instructions!
- Even if the programmer knows what they want the computer to do and has it clear in their mind, the process of typing out each line of code will still take at least 2–3 seconds.

- Given this limitation, how is it that we can ever take advantage of the computer's speed?
  - It takes millions of times longer to type out instructions for a computer than it does for the computer to execute these instructions.
  - Therefore the limiting factor in how fast a computer can operate will always be the typing speed of the programmer.

- Right..?

OF COURSE NOT!

Looping

- The fact is, almost all computational work involves repetition.
- Consider the following high level generic algorithms:

```
Get data
Process data
Get next bit of data
Process data
Get next bit of data
Process data
...
```

Compared with:

```
while there is still Data left
    Get Data
    Process Data
```

- Their behaviour is equivalent but clearly the second one is better. **(Why?)**

- It is *not* more efficient code in terms of the **computational** work the computer has to do but it's certainly more efficient in terms of the **programmer's time**, which is ultimately much more expensive than the computer's time!
- It also allows for a variable number of data items.

- This type of code which involves controlled repetition is called *iteration* or, more commonly, *looping*.

- This is significant because often the most intensive computational work involves repeating the same task over and over for a different set of data each time.

Looping Conditions

- Before we look at some specific iteration statements, we first need to cover a bit of theory.

- Just like selection statements, boolean expressions (conditions) play a critical role in loops.

- For selection this related to whether or not a certain set of statements would be executed:

```
if x == y
        do something
else
        do something different
```

- However, for iteration these boolean expressions determine whether a loop will keep going or stop.

- These are *looping conditions*, e.g.:

```
while x == y
        do this
```

- However, the rules for the boolean expressions used for looping conditions are exactly the same as for selection statements like *if-else*.

- As with *if-else* remember that if you use the assignment operator "=" instead of the equivalence test "==" then you will get incorrect results.
  - Since it is a boolean expression, if the result is zero then the expression will be false otherwise it will be true.

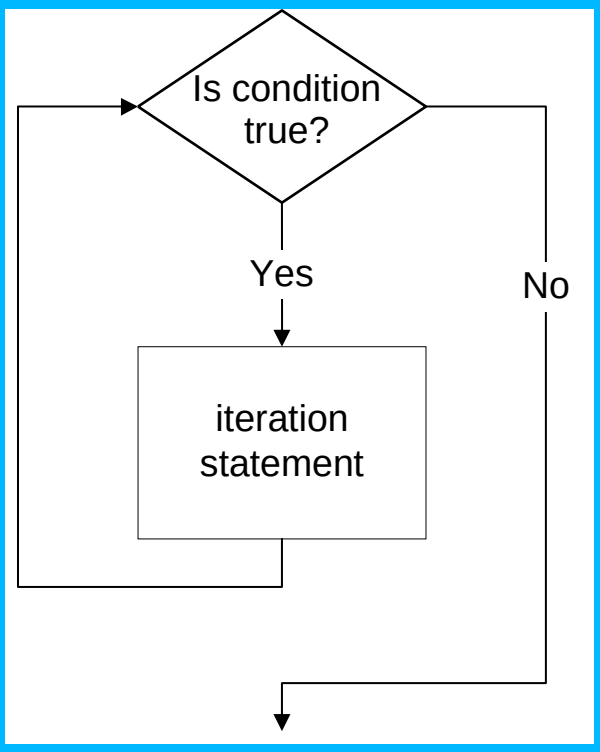
WHILE LOOPS

Overview

- The most commonly used loop in C is the *while* loop.

- It is called a "pre-test" loop meaning that it does the test for its looping condition *before* actually beginning execution of the loop.

- Each time it loops around it then tests this condition again.
  - If the condition is true then the loop executes again, otherwise it stops.



The syntax of the *while* loop is:

```
while(condition)
{
   /*   This is the "body" of the loop.
        These statements get executed
        each time the loop iterates.
   */
}
```

- Again the curly brackets showing the body of the loop are only required when there is more than one line of code.

while *Loop Examples*

- We need to write a program to print all *even* numbers between 10 and 20 inclusive.

- We could do it with 6 print statements, but using a `while` loop is much more efficient and flexible.
  - We can easily modify the program later on to cater for different start and end points, particularly if we use constants where appropriate.

- We will design our program using pseudocode.

- The pseudocode for our solution using a `while` loop is:

```
value = 10
while value <= 20
        Print value
        value = value + 2
```

Here is a C program to do the same thing:

```
#include <stdio.h>

/* Assume that we begin and end with an even
   number.
*/
const int START = 10;
const int END = 20;

int main()
{
   int value = START;

   while(value <= END)
   {
     printf("%d ", value);
     value += 2;
   }

   /* Print a new line to finish */
   printf("\n");

   return(0);
}
```

- Note the use of the constants.
  - If we wanted the program to work between 0 and 1000 instead we could make this change by altering the two values found easily at the very top of the program.
  - Even for this simple program this would be trickier without using constants.
  - But it would be practically impossible without the loop!

A Menu Program with a while *Loop*

- Now we have the tool necessary to allow our menu program from the previous topic to keep executing after the user has made a selection.
- Now the user can enter *q* to quit.

- We can also add a feature to convert the user's input to lower case.
  - This allows us to simplify our *switch-case* statements.

```
/* Menu Program in C demonstrating
    the use of while loops.
*/

#include <stdio.h>
#include <ctype.h>

int main()
{
    char response;

    /* Print menu giving choices */
    printf("You have three choices.\n");
    printf("Enter a for option 1\n");
    printf("Enter b for option 2\n");
    printf("Enter c for option 3\n");
    printf("Enter q to quit.\n");

    /* Read in user's response and convert
         it to lower case.
    */
    scanf("%c%*c", &response);
    response = tolower(response);
```

```
    while(response != 'q')
    {
       /* Perform appropriate response */
       switch(response)
       {
           case 'a':
             printf("You have selected option
                    1\n");
             break;
           case 'b':
             printf("You have selected option
                    2\n");
             break;
           case 'c':
             printf("You have selected option
                    3\n");
             break;
           default:
             printf("You haven't selected a
valid option.\n");
       }

       /* Re-display menu */
       printf("\n");
       printf("You have three choices.\n");
       printf("Enter a for option 1\n");
       printf("Enter b for option 2\n");
       printf("Enter c for option 3\n");
       printf("Enter q to quit.\n");

       scanf("%c%*c", &response);
       response = tolower(response);
    }

    return(0);
}
```

FOR LOOPS

Fixed Iteration while *Loops*

- A *while* loop is designed for situations where you don't know how many times the loop is supposed to repeat.
  - The loop will just keep on going (possibly indefinitely) until the specified condition becomes true.

- However, it is very easy to write a *while* loop that executes a fixed number of times.

- To do this we need a variable called a "counter" that counts the number of times the loop has executed.
- We also need to know the number of times the loop is supposed to execute.

Here is a simple algorithm for this:

```
count = 0
while count < MAX
        process data
        count = count + 1
```

- Note the use of *count* and *MAX* here, the loop counter and maximum number of iterations respectively.
- Also note that we initialise *count* to **zero** and have the loop repeat only while *count* is **less than** *MAX*.

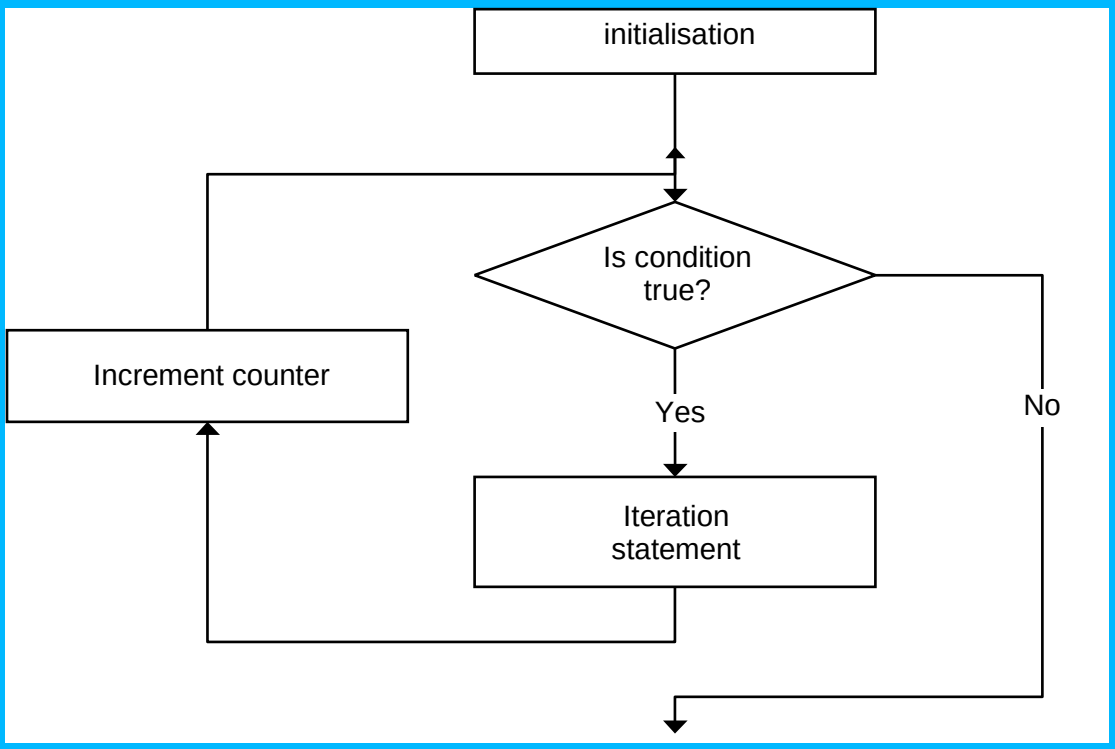
- We could start *count* at one and loop while it is less than or equal to *MAX* but it is conventional (and often useful) to start at zero instead.

for *Loop Syntax*

- However, because fixed-iteration loops are such a common requirement, most programming languages include a special looping construct specifically to do this.

- These are called *for* loops and they do exactly the same thing as the example above with the *while*.

The structure of a *for* loop is:



for Loop Syntax

- The syntax of the *for* loop is:

```
for(initialisation; condition; increment)
{

}
```

Note:

- There are three parts, separated by semicolons.
- The *initialisation* part is performed just before the loop begins the very first time.
  - It is most often concerned with zeroing the counter variable e.g., *count = 0*
- The *condition* part is the loop condition which determines whether the loop keeps going (e.g., *count* < *MAX*).
  - This is tested before the loop begins executing making *for* a pre-test loop just like *while*.
  - It is also tested for **before** beginning each subsequent iteration (just like *while*).
- The *increment* part is executed after the loop has finished executing the code between the curly brackets **each time** the loop iterates.
  - It is normally used to increment a counter variable
    - e.g., `count = count + 1` or, more conventionally, `count++`
  - Note the increment part is executed before the loop *condition* is re-evaluated.

for *Loop Example*

Here is a code fragment to print out all one hundred numbers between zero and 99 inclusive, separated by spaces:

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

Note:

- Curly brackets again aren't required if there is only **one** line of code that makes up the loop body (however, you could still use them if you wanted to).
- The initialisation part is nearly always of the form:
  counter = 0
- The condition part is always a boolean expression (just as for *while* loops and *if-else* statements etc.).
- It is nearly always of the form:
  counter < MAX
- The increment part is nearly always of the form:
  counter++
- Therefore the code uses the convention that if you have *x* values to process, you start with the counter at zero and loop while it is less than *x*.

Another for *Loop Example*

```
/* Find all even numbers between user-specified
   start and end points inclusive */

#include <stdio.h>

int main()
{
   int startval, endval;
   int value;

   printf("Enter start value: ");
   scanf("%d%*c", &startval);

   printf("Enter end value: ");
   scanf("%d%*c", &endval);

   for(value = startval; value <= endval; value++)
   {
        if((value % 2) == 0)
        {
           printf("%d ", value);
        }
   }

   /* Print a new line to finish */
   printf("\n");

   return(0);
}
```

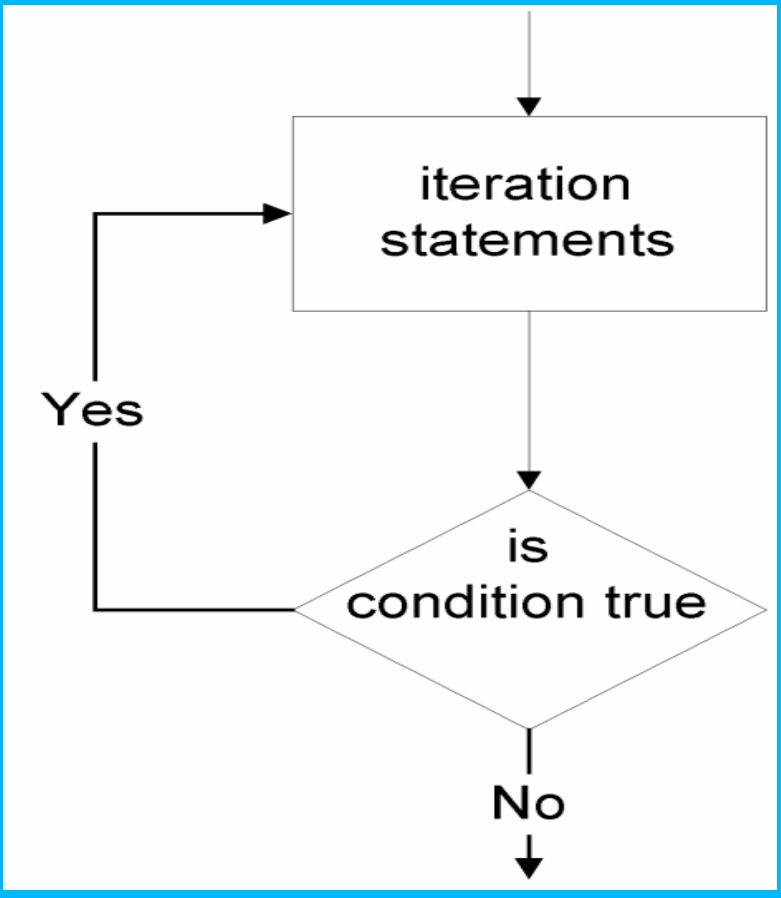
- Note the use of the mod operator %: if there is no remainder when dividing by 2, the number must be even.

DO-WHILE LOOPS

- Both *while* and *for* loops are *pre-test* loops where the loop condition is tested before beginning execution of the loop.

- However, sometimes a *post-test* loop that tests this condition after the execution of the loop body is more appropriate.
  - This is useful where you know that the loop should execute at least once.

- In C this type of loop is called a *do-while* loop.



The syntax of a *do-while* loop is:

```
do
{
   /* Loop code in here */
} while(condition);
```

- The loop *condition* is a boolean expression, just like the other types of loop.
  - Also note the semicolon at the end.

- Compared to *while* and *for* loops, *do-while* loops tend to be less common.

- However “menu” programs that need to loop around to allow the user to continue to select options must loop at least once (to give the user at least one choice).
- We will now look at the menu program from above, using a *do-while* loop.

```
/* Menu Program in C demonstrating
   the use of do-while loops.
*/

#include <stdio.h>
#include <ctype.h>

int main()
{
   char response;

   do
   {
      /* Print menu giving choices */
      printf("You have three choices.\n");
      printf("Enter a for option 1\n");
      printf("Enter b for option 2\n");
      printf("Enter c for option 3\n");
      printf("Enter q to quit.\n");

      /* Read in user's response and convert
        it to lower case.
      */
      scanf("%c%*c", &response);
      response = tolower(response);

      /* Perform appropriate response */
      switch(response)
      {
         case 'a':
            printf("You have selected option
                   1\n");
            break;
         case 'b':
            printf("You have selected option
                   2\n");
```

```
            break;
        case 'c':
            printf("You have selected option
                    3\n");
            break;
        case 'q':
            break;
        default:
            printf("You haven't selected a
valid option.\n");
      }

   } while(response != 'q');

   return(0);
}
```

- If you compare this to the *while* example you'll see it's both shorter and more logical.
  - It is no longer necessary to print out the menu and read in user input in two separate places.

WHICH LOOP WHEN?

- Once you've identified that you need an iterative structure in an algorithm, the next step is to identify which type of loop is required.

- The following table summarises the properties of each type of loop:

Loop Property	Pre-test (while)	Post-test (do-while)	Fixed Iteration (for)
Condition tested	Before loop begins	After each iteration	Before loop begins
Min. No. Iterations	Zero	One	Zero
Special Features	Nil	Nil	Initialisation, Automatic Increment

- Using these properties, it becomes possible to apply some simple rules in attempting to select the appropriate loop.

1. Do you know how many times the loop should repeat?
   - Or, more accurately, is the number of times it should repeat known immediately before the loop is to begin?
   - If so, then a ***for loop*** should be chosen.
   - This is common where there is a fixed or known amount of data that needs to be processed.

2. Does the loop need to execute *at least once*?
   - If so, then a ***do-while loop*** is required.
   - This is common where there is data or some input from the user etc. that must be received and processed before the decision about whether the loop should repeat can be made.
   - An example for this might be a menu program.

3. For anything else, a ***while loop*** is probably appropriate.
   - Specifically, there often cases where there may be zero or more pieces of data to be processed.
   - The actual amount is not known and since there could be no data at all to be processed, a while loop that may not even execute once is appropriate.
   - An example of this that we will see later in the unit is working with data from files:
     - When opening a file and attempting to read data out of it, the file may contain many records, each of which will need to be processed, or simply be empty meaning there is nothing to be processed.

- Despite the above guidelines, there is often a fair bit of flexibility in the use of these loops.
- For example, a while loop can be used in almost any situation.
  - One technique that involves this is known as a *primed while loop*.
  - This applies where the data that is used in the loop condition is obtained before the beginning of the loop, thus allowing a while loop to emulate a do-while.
  - The menu program described previously on Page 7 is an example of this.

- However, most of the time it is probably better to try and use the type of loop that is most appropriate to the requirements of the algorithm.

NESTED LOOPS

- In the previous topic we saw how *if-else* statements could be "nested" within one another.
- This is also possible with loops and can be very powerful.

- However, it is less common than nesting *if*s and can also be conceptually more difficult so we only cover the basics in this unit.

- If you think of one loop within another then for each single iteration of the outer loop, the inner loop will have to complete all of its iterations.

- This leads to a "slow" loop (the outer) and a "fast" loop (the inner).
  - The slow outer loop must wait for the fast inner loop to complete each time.

- Consider a hypothetical **simplified** calendar year made up of 12 months, each of 31 days.
  - An outer loop will loop for each of the months and the inner loop for each day of each month.

- Here is an algorithm involving nested loops to print a year's calendar using this scheme:

```
for month = 1 to 12
        print "Month ", month
        for day = 1 to 31
                print day
```

Here is a slightly enhanced version of the above algorithm written in C.

```
#include <stdio.h>

/* Use these constant values for simplicity */
const int NUM_MONTHS = 12;
const int NUM_DAYS = 31;

int main()
{
   int month, day;

   /* Outer "slow" loop iterates for each
       month */
   for(month = 1; month <= NUM_MONTHS; month++)
   {
       printf("Month %d\n", month);
       for(day = 1; day <= NUM_DAYS; day++)
       {
         /* Inner "fast" loop iterates for
       each day of each month */
         printf("%d ", day);

         /* Print a blank line every
         seven days */
         if((day % 7) == 0)
            printf("\n");
       }

       printf("\n\n");
   }

   return(0);
}
```

GETTING OUT AND GOING BACK

The break *Statement*

- We've already seen the *break* statement used when writing *switch-case* statements.
- There it “breaks out” of the *switch-case* rather than letting execution fall through to the next *case*.

- However, *break* has another use when it comes to loops.
- A *break* statement anywhere in a loop will cause the loop to stop and execution will continue on from the end of the loop.
- In other words, you can also “break out” of a loop.
  - Note that each *break* statement will break out of only the inner most structure (i.e., *switch-case* or loop).

- This is generally used in one of three ways.
  - Firstly there may be one main condition where you want the loop to stop but also one or more other less significant conditions too.
    - Here you can use *if* statements to test for these conditions and break out of the loop.
  - Secondly it may necessary (or at least much easier) to structure your loop such that you can get out of the loop at any point rather than at the start (*while/for*) or end (*do-while*).
    - Again, you can use *if* statements to exit the loop at any point, although generally it's better to have fixed exit points if you can.

- Finally sometimes the main loop condition is set to a boolean expression which is always true and *break* statements take over completely.

```
while(1)
{
    /* do things */
    if(condition)
      break;
}
```

- This works because in C there is no boolean type and any expression that evaluates non-zero is considered true (zero is false).
- Therefore `while(1)` will always be true and an infinite loop results.
- Instead the condition is moved to the *if* statement and *break* is used to exit the loop.

- This technique is very much in the C idiom and actually quite common so you do need to be aware of it.
- However, it can easily become confusing so it is usually best to avoid it since it is mostly not necessary.

This example of *break* is code that calculates the sum of a fixed number (*MAX*) of positive integers.

However, it allows the user to enter less than *MAX* numbers and tell the program to stop adding up by entering a negative number.

```
for(count = 0; count < MAX; count++)
{
    printf("Enter number: ");
    scanf("%d", &num);

    /* If the user enters a negative,
       break out of the loop */
    if(num < 0)
       break;

    sum += num;
}

printf("Sum is: %d", sum);
```

- It is possible to re-write this code without the use of a *break* statement but it becomes more complex and harder to understand.

The continue *Statement*

- The *continue* statement is similar to *break* except it takes execution back to the start of the loop.

- This can sometimes be useful to avoid writing complicated code to skip around parts of your loop and go back to the start.
- However, like *break* it can become confusing if you are not very careful.

The following code fragment adds up all the **odd** numbers between 1 and 100.

```
for(i = 1; i < 100; i++)
{
    /* Skip back to the start of the loop if
          the number is even.
    */
    if(i % 2 == 0)
        continue;

    sum = sum + i;
}

printf("The sum of all odd numbers is %d",
sum);
```

- This code wouldn't be too difficult to modify to avoid using the *continue* statement and would arguably be easier to understand **without** the *continue*!

When to use break *and* continue*?*

- Both *break* and *continue* can be useful programming tools because they provide greater flexibility in determining the flow of your program.

- However, this great strength is also a great weakness.

- By limiting the way code execution can flow, programmers generally write better, easier to understand code.
  - Code execution should follow tightly controlled, orderly paths that are easy to analyse.

- Some strongly believe that *all* loops should only have one entry point and one exit point and few would disagree that this is by far the best way to structure your programs.
  - The problem is, this is not *always* easy to do.

- So *break* and *continue* can make it much harder to work out how a piece of code is supposed to behave.
- Hence they need to be used extremely carefully and probably shouldn't be used very often.

- So when should you use them?

- My advice is not to use *break* and *continue* in your every day programming.
- However, if you are having trouble structuring your code to do what you want, ask yourself "Would *break* or *continue* **make things simpler** here?"

- If the answer is "yes", then use it but carefully comment your code!

SUMMARY

- Iteration or looping allows us to take advantage of a computer's speed in processing large amounts of data in a repetitive way.

- Loops have conditional (boolean) expressions that decide whether the loop will continue to execute.

- The *while* loop is very common and is a pre-test loop in that the test for whether it should execute is done at the start of the loop.
- The *for* loop is essentially the same as a *while* loop but it has special provisions for looping a fixed number of times.
- The *do-while* loop is a post-test loop and tests its looping condition at the end of the loop.

- The *break* and *continue* constructs allow you to modify the flow of loop code, however, they should be used very sparingly as they can often make it very hard to determine who a piece of code behaves.