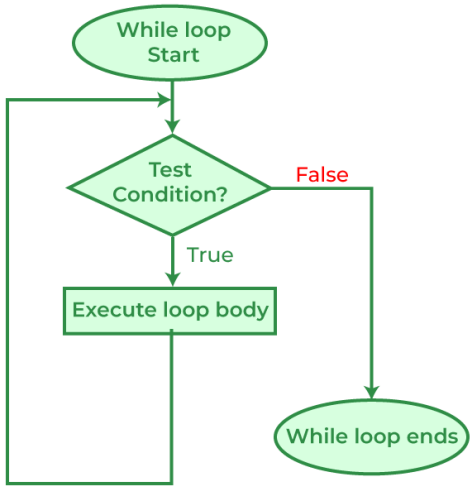
Cycles

Pedro Ribeiro

DCC/FCUP

2024/2025



(based and/or partially inspired by Pedro Vasconcelos's slides for Imperative Programming)

Cycles

- A **cycle** is an instruction that executes other instructions several times (the **body** of the cycle)
- Cycles in C are controlled by an **expression**
- The expression is evaluated at each **iteration**
  - if its value is zero (*false*), the cycle ends
  - if it is not zero (*true*), the cycle continues

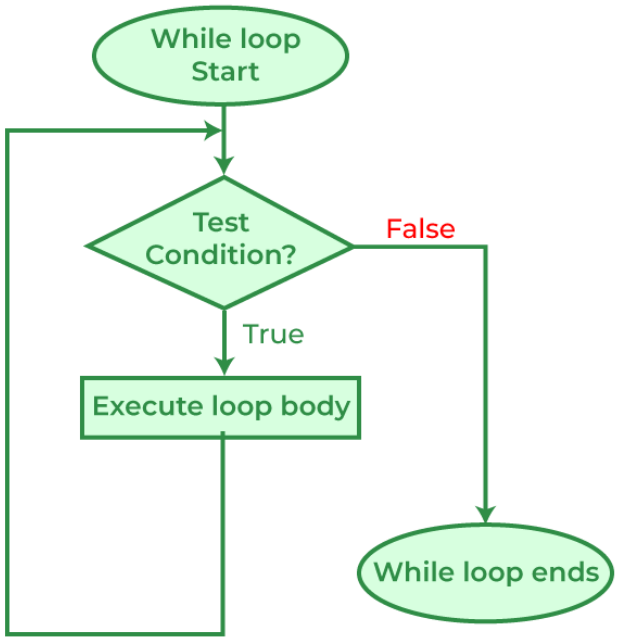
Cycle instructions

- **while**
  is used for cycles in which the expression is tested before executing the body of the cycle
- **do ... while**
  is used for cycles in which the expression is tested after executing the body
- **for**
  is a convenient form for cycles with a control variable

While statement

- `while ( expression ) statement`
  - The **expression** controls the termination of the cycle
  - The **statement** is the body of the cycle

- Execution:
  1. First evaluates the expression:
  2. If it is zero (*false*), the loop ends immediately;
     If non-zero (*true*), executes instruction and repeats 1.



(image source: geeksforgeeks)

While statement - example

```
i = 1;
while (i < 10)   // control expression
  i = i * 2;    // body of the cycle
```

`i = 1;`	
`i < 10?`	1 (*true*)
`i = i * 2 = 2`	
`i < 10?`	1 (*true*)
`i = i * 2 = 4`	
`i < 10?`	1 (*true*)
`i = i * 2 = 8`	
`i < 10?`	1 (*true*)
`i = i * 2 = 16`	
`i < 10?`	0 (*false*)

While statement

- The body can be a **block** of instructions instead of just one:

```
i = 1;
while (i < 10) {
  printf("%d\n", i);
  i = i * 2;
}
```

- We can use curly braces even with a single instruction:

```
i = 1;
while (i < 10) {
  i = i * 2;
}
```

Termination

- The `while` loop ends when the value of the expression is `0` (*false*)
  - e.g., if the expression is `i < 10` then the cycle ends when $i \geq 10$
- The body may not execute (because the control expression is tested first)
- If the control expression is *always* non-zero, the loop *doesn't end*
  (unless we use special instructions to exit the loop - more on that later)

```
while (1) {
    ...    // infinite loop
}
```

Example cycle - table of squares

- `squares.c` (source code) - a program to print a table of squares

```c
#include <stdio.h>

int main(void) {
  int i, n;

  printf("Upper limit: ");
  scanf("%d", &n);
  i = 1;
  while (i <= n) {
    printf("%d\t%d\n", i, i*i); // is a tab
    i ++;
  }

  return 0;
}
```

```
Upper limit: 4
1         1
2         4
3         9
4        16
```

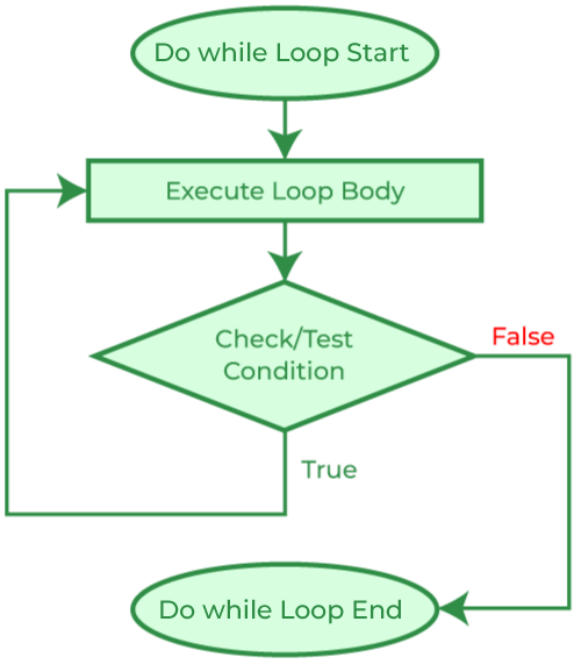
Example cycle - summing numbers

- `sum.c` (source code) - a program to add up a sequence of numbers
- The length of the sequence is not known in advance. Idea:
  - read each value within a cycle
  - accumulate the total in an auxiliary variable
  - terminate when we read a special value (zero)

```c
#include <stdio.h>

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

  printf("Enter values; 0 ends.\n");
  scanf("%d", &n);        // first value
  while (n != 0) {        // while not finished
    sum += n;           // accumulate
    scanf("%d", &n); // read next value
  }
  printf("The sum is: %d\n", sum);

  return 0;
}
```

do...while statement

- `do statement while ( expression );`
- Execution
  1. First executes the instruction
  2. Then evaluates the expression
  3. If it is zero (*false*), the cycle ends;
     If non-zero (*true*), repeat step 1.



(image source: geeksforgeeks)

do...while example - summing numbers revisited

- Let's rewrite the program to add numbers using a `do...while` loop.

  `sum2.c` (source code)

```c
#include <stdio.h>

int main(void) {
    int n, sum = 0;
    printf("Enter values; 0 ends.\n");
    do {
        scanf("%d", &n); // next value
        sum += n;        // accumulate
    } while (n != 0);    // while not finished
    printf("The sum is: %d\n", sum);
    return 0;
}
```

- Remarks:
  - As the condition is tested *after* execution, we don't need to read the first value out of the loop
  - Adding `0` doesn't change the result: we can always accumulate

do...while example - number of digits

- `digits.c` (source code) - compute nr of digits in a positive integer
  - Let's use a cycle to do integer divisions by 10;
  - We finish when it reaches zero
  - The number of iterations performed gives us the digit count
  - The `do...while` loop is more convenient than while because any positive number has at least one digit

```c
#include <stdio.h>

int main(void) {
  int digits = 0, n;
  printf("Enter a positive integer: ");
  scanf("%d", &n);
  do {
    n /= 10;    // quotient of division by 10
    digits++;   // one more digit
  } while (n > 0);
  printf("%d digit(s)\n", digits);
  return 0;
}
```

```
Enter a positive integer: 5633
4 digit(s)
```

for statement

- `for ( expr1; expr2; expr3 ) statement`
  - `expr1` is the *initialisation*
  - `expr2` is the *repeat* condition
  - `expr3` is the *update* after each iteration
  - `statement` is the *body* of the loop

- Example:

```c
for (i = 0; i < 5; i++)
  printf("%d\n", i);
```

```
0
1
2
3
4
```

for statement

- We could use `while` instead of `for`, but `for` is clearer/simpler.

  The general form:
  `for (expr1; expr2; expr3) statement`

  is equivalent to:

```
expr1;
while (expr2) {
   statement
   expr3;
}
```

- For example, the following two pieces of code are equivalent:

```
for (i = 0; i < 5; i++)
  printf("%d\n", i);
```

```
i = 0;
while (i < 5) {
  printf("%d\n", i);
  i++;
}
```

(this "translation" may help to understand some details)

Common usages of for

- The for statement is convenient for cycles that need to count from a start value to an end value

- Examples of repeating *n* times:

```
// count up from 0 to n-1
for(i = 0; i < n; i++) ...

// count up from 1 to n
for(i = 1; i <= n; i++) ...

// count down from n-1 to 0
for(i = n-1; i >= 0; i--) ...

// count down from n to 1
for(i = n; i > 0; i--) ...
```

Common errors when using for

- Swapping the order of comparisons
  - ascending counts should use `<` or `<=`
  - descending counts should use `>` or `>=`
- Use `==` instead of `<`, `<=`, `>`, `>=`
  - we can accidentally *"skip"* the termination
- *"Miss by one"* the termination condition
  - e.g. use `i < n` instead of `i <= n` (or *vice-versa*)

Omitting expressions in for

- We can omit one or more expressions in the for loop.
- Omitting the **initialisation**:

```
int i = 5;
for (; i > 0; i--)
   printf("%d\n", i)
```

- Omitting the **update**:

```
int i;
for (i = 5; i > 0;)
   printf("%d\n", i--)
```

- Omitting both **initialisation** and **update**:

```
int i = 10;
for (; i > 0;)
   printf("%d\n", i--)
```

Omitting expressions in for

- If we omit the condition, the `for` loop only ends if we use exit statements in the body (more on this later)

- Some programmers prefer to use an unconditional `for` instead of a `while` for these types of (infinite) cycles

```c
// using a while loop
while (1) {
    ...
}
```

```c
// using for loop
for (;;) {
    ...
}
```

Declaring a variable in the initialization expression

- Since C99, the `for` initialisation expression can be replaced by a declaration.
- This allows you to declare a variable for use within the loop: `for(int i = 0; i < n; i++) ...`
- The variable does not have to be declared first and is **limited in scope** to the cycle
- Declaring the control variable inside the loop is convenient and can help make the program simpler
- However, if we want to use the final value of the variable after the end of the cycle, we need to declare it before the cycle

```
for(int i = 0; i < n; i++) {
   printf("%d\n", i); // OK: i is valid
}
printf("%d\n", i); // ERROR: i out of scope
printf("%d\n", n); // OK: n is valid
```

Break statement

- Usually a loop ends only when the condition is tested
  - before a `while` or `for` iteration
  - after a `do...while` iteration
- We can also use the `break` statement to end a cycle at any time
- Here is an example for finding the first proper divisor of a number n:

```
int i, n;
scanf("%d", &n);
for (i = 2; i < n; i++) {
    if (n%i == 0) break;
}
if (i < n)
    printf("Found a divisor: %d\n", i);
else
    printf("No divisors\n")
```

- This loop can end in one of two ways:
  - if it has exhausted the possible divisors ($i \geq n$)
  - if it has found a divisor ($i < n$)

Break statement

- The `break` statement is useful for writing a loop with a termination test in the middle

- Example: reading a sequence of values and ending with a special value

```
for(;;) {
  scanf("%d", &n);
  if (n == 0) // if zero
     break;   // end the cycle
  ...         // if not:  process the value
}
```

continue statement

- The `continue` statement transfers execution to the point just before the end of the body
- While `break` ends the cycle, `continue` continues in the cycle
- Example: Read and sum 10 non-negative integers.

```
int i = 0, n, sum = 0;
while (i < 10) {
  scanf("%d", &n);
  if (n < 0)
    continue;
  sum += n;
  i++;
  // continue jumps to here
}
```

continue statement

- Sometimes it is simpler to avoid the `continue` by putting statements inside an `if` statement
- The condition is inverted: `n >= 0` instead of `n < 0`

```c
int i = 0, n, sum = 0;
while (i < 10) {
  scanf("%d", &n);
  if (n >= 0) {
     soma += n;
     i++;
  }
}
```

Empty statement

- An instruction can be empty, i.e. just a semicolon with no other symbols:

```
i = 0; ; j = 1; // second empty instruction
```

- An empty instruction does nothing; it is only useful for writing a loop whose body is empty.

- Consider the loop for finding divisors:

```
for (i = 2; i < n; i++) {
    if (n%i == 0) break;
}
```

- We could join the two conditions and remove the break; the body is empty:

```
for (i = 2; i < n && n%i != 0; i++); // empty instruction
```