

Looping Constructs

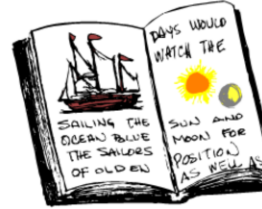
- **Objective**
 - To implement iteration
- **Contents**
 - The *while* statement
 - The *do while* statement
 - The *for* statement
 - The *break* statement
 - The *continue* statement
- **Summary**
- **Practical**
 - Simple implementation of traditional iteration



This is the first of two chapters that cover C's flow-of-control constructs. The objective of this chapter is to cover all of C's looping constructs. Emphasis is placed on the C 'quirks', rather than on programming techniques.

The Story So Far...

- The forms of flow control we seen so far are...
 - Sequential
 - Selective
- Useful programs also need loops
 - Iteration



```
int main(void)
{
    ...
    ...
    return 0;
}
```

Sequential

```
int main(void)
{
    ...
    ...
    return 0;
}
```

Selective



By the end of this chapter, the story will be complete. C's decision-making constructs are fairly conventional. There are two oddities: a dedicated decision operator and a statement that controls loops in an unorthodox manner.

QACPROG

The *while* Statement

The simplest iterative statement

test is at start of iteration

continuation condition

executed while result of expression remains non-zero (true)

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

The fundamental looping construct in C is the `while` loop. In this structure a statement (called the *while body*) is repeatedly executed while a control expression evaluates to true. The repetition ends when this condition evaluates to false. Because the condition is checked before the loop body it is possible for the body to never be executed, i.e. the condition is false initially.

A classic mistake with a `while` loop is to forget to update the variable or variables used to the control expression in the loop body. This has the effect of creating an infinite loop: one that never terminates. Many programmers prefer to use the `for` loop (described later) in preference to the `while` loop as it is not as easy to make these kind of mistakes.

The output created by the code on this slide is: 0 1 2 3 4 (all on their own lines)

The code in the slide can be slightly improved by noticing that `printf("\n")` can be replaced with the equivalent but safer and more specific `putchar('\n')` or most explicitly of all `fputc('\n', stdout)`.

The *do while* Statement

The second iterative statement

executed at
least once

test is at end
of iteration

```
do  
    statement  
while (expression);
```

executed while result
of expression
remains non-zero (true)



must be semi-colon
terminated

```
int x;  
do  
{  
    puts("Enter non-zero integer");  
    scanf("%d", &x);  
}  
while (x == 0);
```

The C `do while` loop involves a test on exit, which follows the statement. This statement can be a simple statement or a compound statement. The statement is repeatedly executed only if the test is true.

Note the syntax: the construct starts with the `do` keyword (usually on a line of its own). This is followed by a simple statement or a compound statement. The test is in parentheses following the `while` keyword. The final `;` signals the end of the construct.

A common error is to leave the parentheses out of the compound statement. This is picked up by the compiler.

The output from the code in this slide is the same as for the previous `while` loop:
0 1 2 3 4 (all on their own line).

Exercise: Spot the Bugs

①

```
int x;
do
{
    puts("Enter non-zero integer");
    scanf("%d", &x);
}
while (x = 0);
```

②

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



③

```
int x;
scanf("%d", &x);
while (x < 5)
    printf("%d\n", x);
    x++;
```

The first example illustrates a very common bug! The intention was to terminate the do while loop when x has a non-zero value; loop while x equals zero. Unfortunately, the programmer has written an assignment statement! However, assignment is an expression, which means it has a value. The value of the assignment (which must be zero) is used as the continuation expression. The integer zero is implicitly converted to a boolean false and the loop terminates. The correct code uses == instead of =.

```
do
{
    printf("Enter a non-zero integer\n");
    scanf("%d", &x);
}
while (x == 0);
```

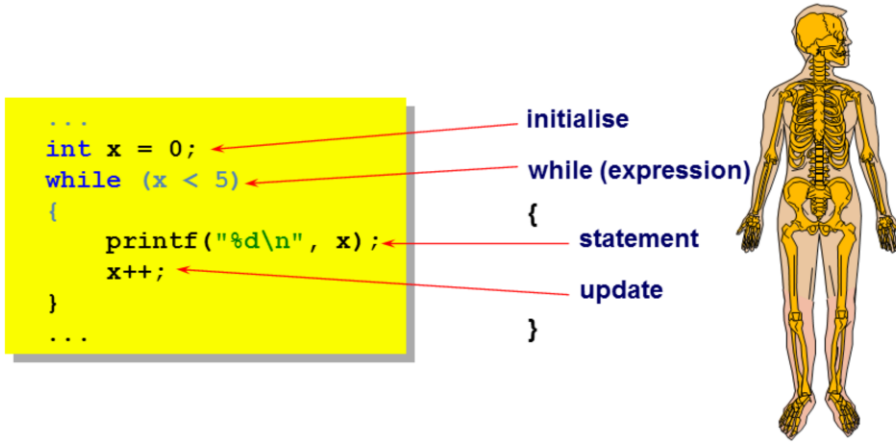
The second example is very subtle. You can stare at it for hours and not see the problem. Look very closely. There is a semi-colon immediately after the closing parentheses of the while statement's continuation expression. To the compiler, the code is formatted as if:

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

The third bug is that the printf is inside the while loop, but the increment is not. A compound statement is required. Notice that if a value less than five was read by scanf the loop would iterate forever. Debug time.

while Statement Anatomy

- Typically, a while loop can be broken into *four parts*
 - Identifying these *parts* significantly increases *comprehension*



The four parts of the loop, as described above, are the essential and only ingredients. Good design should bring these items out quite clearly. In C, once these four items have been identified, you can choose the most appropriate construct.

One problem with the while loop is that if the statement body contains many statements, one per line, the top of the loop can end up a long way from the end of the loop. This textual separation can cause bugs. Psychologically, you write a `while` statement (or an iterative statement) because you want to repeatedly execute the loop body. It is very easy, being mentally focused on the `while` body, to forget the final update. The effect of this will be a loop that loops forever. Debug time. One way of avoiding this trap is to write the initialisation, the while (expression), the opening and closing brace of the compound statement, and the update *before* writing the body. Another way is to use the `for` statement.

QACPROG

The *for* Statement


The most common iterative statement

executed while result of
expression remains
non-zero (true)
↓

continuation
condition
↓

executed after
statement
↓

`for (initialise; expression; update)
 statement`



```
int x;  
...  
for (x = 0; x < 5; x++)  
{  
    printf("%d\n", x);  
}
```

The `for` loop makes use of the four items mentioned on the previous page, i.e. initialisation, test, body and update. The syntax of the `for` loop groups three of the items in parentheses and separates them with `;`. Incidentally, this is the only place in C where semicolons are usually found inside parentheses. The body is in the same position as the other looping constructs.

The flow of control adheres strictly to that of the `while` loop, i.e. the initialisation takes place (once only), then we go into the test ...

The `for` loop is far more flexible than the corresponding `while` loop. We shall see, in the following pages, that the initialisation and update items can be quite complex.

The `for` statement in the slide could be:

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

However, many C style guides recommend *always* using a compound statement as the body of a `while`/`do`/`for` statement. This makes maintenance much easier - if we need to add another 'inner' statement we can do so without having to also add the braces to form a compound statement (which we might forget). It also presents a uniform visual syntax for the reader.

for Statement Notes

- Multiple comma-separated expressions may appear in the initialise and update parts of a *for* statement
- The initialise, test and update parts may be empty

```
...  
int count, sum;  
  
for (sum = 0, count = 1; count <= 7; count++)  
{  
    sum += count;  
    printf("%d %d\n", count, sum);  
}
```

```
for (;;)   
{  
    ...  
}
```

The initialisation and update items can involve the evaluation of more than one comma separated expression. For instance, the above example has the expression `sum = 0, count = 1` as the initialisation. Both assignments take place. The technique can equally be applied to the update item, where one can envisage expressions such as `count++`, `x += 2`.

The comma operator is a binary operator of the lowest precedence.

The expression `x , y` is evaluated as follows:

Evaluate `x`

Evaluate `y`

`y` is the value of the operation

Used in the `for` loop, the value of the operation is ignored; only the evaluations of the operands are used.

The code in the slide shows the comma syntax inside a `for` loop initialisation, but many programmers would consider the following to exhibit better style:

```
int sum = 0;  
int count;  
for (count = 0; count < 8; count++)  
{  
    sum += count;  
    printf("%d %d\n", count, sum);  
}
```

The *break* Statement

Can be used to jump out of an iteration

```
int guess, try;
for (try = 0; try < max_tries; try++)
{
    scanf("%d", &guess);
    if (guess == answer)
        break;
    puts("Sorry, not right");
}
if (guess == answer)
    puts("Congratulations!");
else
    puts("Hard luck");
```

 *try* is a C++ keyword



A break statement often indicates poor logic. See notes for rewrite.

The effect of the break statements can be shown using labels:

```
try = 0;
beginloop:
    if (!(try < max_tries)) goto endloop;
    scanf("%d", &guess);
    if (guess == answer) goto endloop; /* break */
    puts("Sorry, not right");
    try++;
    goto beginloop;
endloop:
```

The non-continuous control flow does not suit structured programming theory and many programmers advise against their use. Just look how unreadable the previous examples are!

A better approach is to accept that the continuation condition is compound:

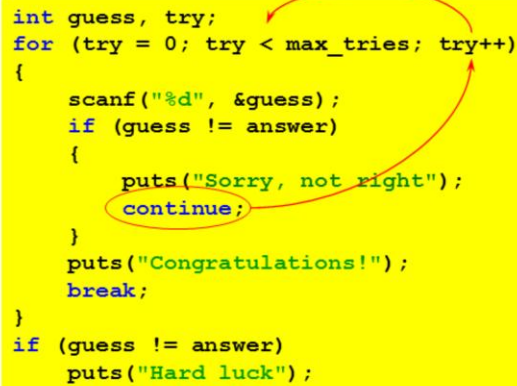
```
int guess, try;
scanf("%d", &guess);
for (try = 0; try < max_tries && guess != answer; try++)
{
    puts("Sorry, not right");
    scanf("%d", &guess);
}
```

Notice that the `scanf` function call is duplicated. This could be eliminated by introducing an input function (see *Functions* chapter) which would then be called as part of the continuation condition.

The *continue* Statement

Can be used to jump to the next step of an iteration

```
int guess, try;
for (try = 0; try < max_tries; try++)
{
    scanf("%d", &guess);
    if (guess != answer)
    {
        puts("Sorry, not right");
        continue;
    }
    puts("Congratulations!");
    break;
}
if (guess != answer)
    puts("Hard luck");
```



A *continue* statement often indicates poor logic. See notes for rewrite.

The effect of the *continue* statements can be shown using labels:

```
try = 0;
beginloop:
    if (!(try < max_tries)) goto endloop;
    scanf("%d", &guess);
    if (guess != answer) goto updateloop; /* continue */
    puts("Congratulations!");
updateloop:
    try++;
    goto beginloop;
endloop:
```

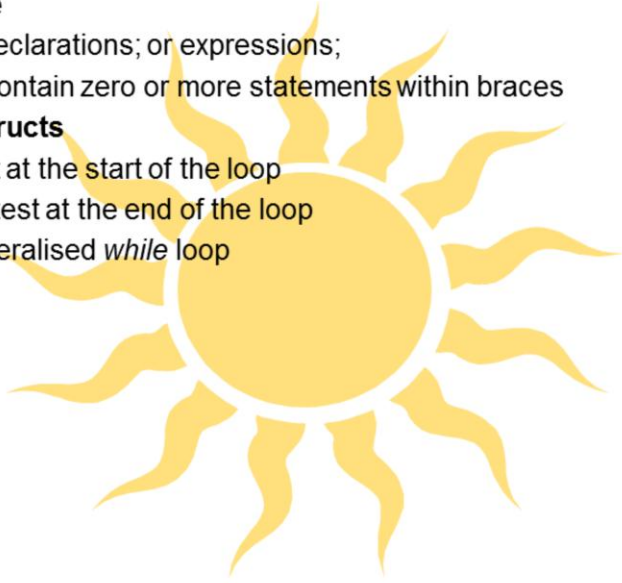
The non-continuous control flow does not suit structured programming theory and many programmers advise against their use. Just look how unreadable the previous examples are! Notice that the *continue* does *not* skip the *try++* update. Many C programming guidelines ban the use of *continue*.

Again, the better approach is to accept that the continuation condition is compound:

```
int guess, try;
scanf("%d", &guess);
for (try = 0; try < max_tries && guess != answer; try++)
{
    puts("Sorry, not right");
    scanf("%d", &guess);
}
```

Summary

- **C is a sequential language**
 - Simple statements are declarations; or expressions;
 - Compound statements contain zero or more statements within braces
- **C has three looping constructs**
 - A *while* loop with the test at the start of the loop
 - A *do while* loop with the test at the end of the loop
 - A *for* loop which is a generalised *while* loop
- **Identify**
 - Initialisation
 - Continuation condition
 - Update
 - Body



Common Pitfalls

- **General**

- Not grouping multiple statements into a compound statement
- Thinking the loop test is a termination condition
- Confusing assignment and equality

```
while (x < 0)
    printf("Enter a +ve number");
    scanf("%d", &x);
...
```



```
for (x = 0; x >= 5; x++)
    ...
```



```
while (x = 0)
    ...
```



- **for**

- Forgetting that semi colons separate the parts of a *for* statement

```
for (x = 0, x < 5, x++)
    ...
```



To group multiple statements together you must use braces { } to form a compound statement. This is one of the reasons that many coding guidelines recommend that *all* if and else bodies should be compound statements, even when they only contain a single statement.

```
while (num1 <= 0 && num2 <= 0)
{
    printf("Enter two +ve numbers");
    scanf("%d %d", &num1, &num2);
}
```

The condition in an for statement is a *continuation* condition

```
for (x = 0; x < 5; x++) ...
```

Everyone's favourite! The programmer intended that x would be compared for equality with zero, which is tested using ==. instead they have assigned zero to x, and then compared the result for truth, as this is zero which is considered false, the while body will never be executed and x will always be zero!

```
while (x == 0) ...
```

The parts of a for loop are separated by semi-colons (not commas).

```
for (x = 0; x < 5; x++) ...
```

Intentionally left blank