

## Overview: Loops

- **while and for loops**
- **infinite and unbounded loops, nested loops**
- **do-while loops**
- **multiple-alternative decisions: switch**

1

## Why are loops needed?

- Because sometimes we want to repeat some processing step(s) many times.
  - Example: we saw earlier a C program to compute the area of a circle, given its diameter. Suppose we wanted to find the areas of 50 circles, each with a given diameter.
    - We could run our earlier program 50 times, once for each circle.
    - Or we could re-write our earlier program to sequentially find the area of each of the 50 circles (using `diameter1`, `diameter2`, ... `diameter50`, `area1`, `area2`, ... `area50`) – but this new program has 50 times as much code as our earlier program (what if we wanted the areas of 1,000 circles?! or 1,000,000, or...)
    - Better: re-write our earlier program to repeat the relevant section 50 times.
- A **loop** is the fundamental programming concept that allows this repetition (also called *iteration*).

2

## The while loop

```
/* crack a dozen eggs for a large omelette */
int n_eggs = 1;
while (n_eggs <= 12) {
    printf("cracked egg number %d\n", n_eggs);
    n_eggs = n_eggs + 1; /* or: n_eggs++; */
}
printf("Finished cracking the dozen eggs\n");
```

termination condition

loop body

Produces the screen output:

```
cracked egg number 1
cracked egg number 2
cracked egg number 3
cracked egg number 4
cracked egg number 5
cracked egg number 6
cracked egg number 7
cracked egg number 8
cracked egg number 9
cracked egg number 10
cracked egg number 11
cracked egg number 12
Finished cracking the dozen eggs
```

3

## The while loop: details

- Loops are another form of conditional statement. In the case of a loop, the computer uses the *termination condition* to determine whether the statements associated with the loop should be repeated or not.
- These statements are referred to as the *loop body*. The loop body can be a single or a compound statement.
- The general form of a **while** loop is as follows:

```
while (condition) {
    statements;
}
```

What happens: The loop body is executed only if the condition is **true**. After the loop body is executed, the condition is re-evaluated; if it is still **true** the loop body is executed again; and so on, and so on, until the condition is **false**.

*Note: If a condition evaluates to false before executing the loop body, the loop body is not executed at all.*

4

## The while loop: details (contd.)

[Aside: in certain situations the condition will never evaluate to **false** and the loop will continue to execute endlessly. Such a loop (usually, though not always, a programming error!) is called an *infinite* loop. An infinite loop may be terminated by interrupting the program or by switching off the computer, both of which terminate the program as well. To interrupt a program press Ctrl/C (and it might work!)]

while loop example: sum up the integers from 1 to 6

```
int i = 1, sum = 0;
while (i < 7) { /* or: while (i <= 6) */
    sum = sum + i;
    i = i + 1; /* or: i++; */
}
printf("Sum is %d\n", sum);
```

which produces the screen output:      **Sum is 21**

5

## The while loop: Example (contd.)

- The loop body is executed only if the condition is **true**. Since we have initialised **i** to 1, the condition (**i < 7**) evaluates to **true** and the loop body is executed. In the loop body, a running total for **sum** is calculated by adding the value of **i** to **sum**. The first time through, the variable **sum** is assigned the value 1. The variable **i** is then increased by 1. We then test the condition again.
- The variable **i** now has the value 2 so the condition is still **true**. Therefore the loop body is executed again, assigning **sum** the value 3 (i.e. 1+2) and increasing **i** to 3.
- Next time around **sum** becomes 6 and **i** becomes 4; next time **sum** becomes 10 and **i** becomes 5; next time **sum** becomes 15, **i** becomes 6; next time **sum** becomes 21, **i** becomes 7.
- Since **i** has reached 7, the condition is now **false**, and the loop exits, so the output statement is executed.

6

## The while loop: Another example

```
/* C code to sum 10 pairs of integers */
int num1, num2, result;
int count = 0;
while (count < 10) {
    printf("Enter first integer: ");
    scanf("%d", &num1);
    printf("Enter second integer: ");
    scanf("%d", &num2);
    result = num1 + num2;
    printf("Result is %d\n", result);
    count++;
}
```

If we omitted the **count++** statement, what would happen?

Answer: **infinite loop** ("logical error" – not a syntax error. The program will compile, but will execute incorrectly).

7

## The while loop: unbounded loop

- in the 2 previous examples, it is very important that the *loop control variables* **i** and **count** are initialised to appropriate values in order that the loops operate correctly.
- however, often you will not know in advance how many times to execute the loop body:
  - suppose that we want to generalise the previous program so that it continues calculating for as long as the user requires. The user may wish to sum one pair of numbers, or may wish to sum 100 pairs of numbers. We can code it so that the user indicates that they are finished by entering -1 as the first number.
  - this type of loop is known as an **unbounded loop** – it will terminate at some point, but we don't know exactly how many iterations are going to be required.

8

## The while loop: example of an unbounded loop

```
#include <stdio.h>
main() {
    int num1 = 0; /* can be anything except -1 */
    int num2, result;
    printf("To exit, enter -1 as value for first number\n");
    while (num1 != -1) {
        printf("Enter first number: ");
        scanf("%d", &num1);
        if (num1 != -1) { /* did user wish to stop? */
            printf("Enter second number: ");
            scanf("%d", &num2);
            result = num1 + num2;
            printf("The sum of the numbers is %d\n", result);
        } /* closes the "if" set of statements */
    } /* closes the "while" set of statements */
} /* closes the main() function */
```

The loop body will continue to be executed as long as the user enters something other than -1 for `num1`. The loop body will **always** be executed **at least once** – *why?*

9

## The while loop: debugging with loops, nested loops

### Debugging with Loops

- Sometimes it can be difficult to understand what is going on when writing code involving loops.
- To aid understanding, it is often a good idea to insert **output statements** inside the body of the loop to print out values of variables. For example, in the “integer summing” code (pg. 5) we could place the following line of code inside the loop body during code development:

```
printf("Sum is %d and i is %d\n", sum, i);
```

### Nested Loops

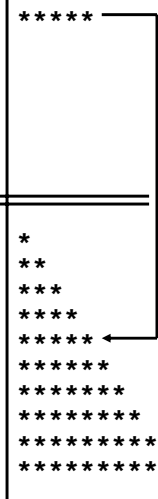
- A loop may contain, as part of its loop body, a statement which constitutes another loop. This is called an *inner loop* or *nested loop*. This may in turn contain its own nested loop, and so on...

10

## The while loop: example of a nested loop

```
int nstars = 5, stars = 1;
while (stars <= nstars) {
    printf("**");
    stars++;
}

int nstars = 1, stars;
while (nstars <= 10) { /* outer loop */
    stars = 1; /* reset to 1 each time */
    while (stars <= nstars) { /* inner loop */
        printf("**");
        stars++;
    }
    printf("\n");
    nstars++;
}
```



11

## The while loop: Exercise

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

### Produces the screen output:

12

### 3 for loops and their while equivalents

0+1+2 +...+9	<pre>int total = 0, i; for (i = 0; i &lt; 10; i++) {     total += i; }</pre>	<pre>int total = 0, i = 0; while (i &lt; 10) {     total += i;     i++; }</pre>
ten sums	<pre>int count, num1, num2, result; for (count=0; count&lt;10; count++) {     printf("Enter 1st number: ");     scanf("%d",&amp;num1);     printf("Enter 2nd number: ");     scanf("%d",&amp;num2);     result = num1 + num2;     printf("The sum is %d\n",result); }</pre>	<pre>int count = 0, num1, num2, result; while (count &lt; 10) {     printf("Enter 1st number: ");     scanf("%d",&amp;num1);     printf("Enter 2nd number: ");     scanf("%d",&amp;num2);     result = num1 + num2;     printf("The sum is %d\n",result);     count++; }</pre>
*	<pre>int nstars, stars; for (nstars=1; nstars&lt;=10; nstars++) {     for (stars=1; stars&lt;=nstars; stars++) {         printf("*");     }     printf("\n"); }</pre>	<pre>int nstars = 1, stars; while (nstars &lt;= 10) {     stars = 1;     while (stars&lt;=nstars) {         printf("*");         stars++;     }     printf("\n");     nstars++; }</pre>

### The for loop

- The **for** loop forces the programmer to explicitly initialise the counter, to provide a condition for loop continuation, and to update the counter.

<p><b>general syntax</b></p> <pre>for (initialise; condition; update) {     statements; }</pre>
---

```
int stars;
for (stars = 0; stars < 10; stars++) {
    printf("***");
}
```

```
int stars = 0;
while (stars < 10) {
    printf("***");
    stars++;
}
```

← **Equivalent: both produce the screen output \*\*\*\*\***

- initialise** executed only once: when the loop is first encountered. Usually this sets a variable to some initial value (often called a "loop counter").
- condition** is then tested. This usually tests the value of the counter.
- If **true**, then the loop body is executed. If **false**, the **for** loop exits.
- update** called *after* each time around the loop, and we start over.

### Specifying number of iterations

Each of these loops produces the same screen output: \*\*\*\*\*

```
int i;
for (i=0; i <= 4; i++){ /* i increases by 1 each iteration */
/* condition True when i is 0,1,2,3,4 */
/* then False when i is 5 */
    printf("***");
}
```

```
int i;
for (i=0; i < 10; i=i+2){ /* i increases by 2 each iteration */
/* condition True when i is 0,2,4,6,8 */
/* then False when i is 10 */
    printf("***");
}
```

```
int i;
for (i=100; i > 95; i--){ /* i decreases by 1 each iteration */
/* condition True when i is 100,99,98,97,96 */
/* then False when i is 95 */
    printf("***");
}
```

**Exercise: re-write the above as while loops**

### for loop: syntax details

```
int j;
for (j = 0; j < 10; j++) {
    printf("***");
}
```

**produces the screen output \*\*\*\*\***

What would happen if the for statement was changed as follows:

```
for (j = 0; j < 10; ) {
```

```
for (j = 0; -1; j++) {
```

```
for (j = 0; ; j++) {
```

```
for (j; j < 10; j++) {
```

```
int j = 0; for ( ; j < 10; j++) {
```

## Nested for loop: multiplication table

We want a C program which prints out the following multiplication table:

```
    1  2  3  4  5  6  7  8  9
1  1
2  2  4
3  3  6  9
4  4  8 12 16
5  5 10 15 20 25
6  6 12 18 24 30 36
7  7 14 21 28 35 42 49
8  8 16 24 32 40 48 56 64
9  9 18 27 36 45 54 63 72 81
```

Recall the *Engineering Problem-Solving Methodology* from the Introductory Topics lecture: let's assume that the first 3 steps (clear problem statement, describe inputs & outputs, work a simple example) have been done – now we have to develop an algorithm and then implement it in C...

17

## Nested for loop: multiplication table (contd.)

- Have to compute table row-by-row.
- The first row prints out the headings for the table. This is straightforward.
- For each subsequent row, we will print out the row number and then print out the list of multiples. So an initial attempt is:
  1. Print out row of integers from 1 to 9
  2. Loop for Counter = 1 to 9
    3. Print out Counter multiples of Counter

*e.g. if Counter is 6, print 6 multiples of 6: 1\*6,2\*6,...,6\*6*
- Step 3 clearly requires further refinement:
  1. Print out row of integers from 1 to 9
  2. Loop for Counter = 1 to 9
    - 3.1 Move onto new line
    - 3.2 Print value of Counter
    - 3.2 Loop for Counter2 = 1 to Counter
      - 3.3 Print out Counter\*Counter2
- Step 1 also requires a bit more work:
  - 1.1 Loop for Counter = 1 to 9
    - 1.2 Print out Counter

18

## Nested for loop: multiplication table (contd.)

```
int counter, counter2;
for (counter = 1; counter <= 9; counter++) {
    printf("\t%d", counter); /* \t means horizontal tab */
}
for (counter = 1; counter <= 9; counter++) {
    printf("\n");
    printf("%d", counter);
    for (counter2 = 1; counter2 <= counter; counter2++) {
        printf("\t%d", counter*counter2);
    }
}
```

Things for you to think about: how would you modify this code so that

- the user can choose the final value of **counter** (e.g. 7 instead of 9)
- the user can choose the operation to be performed (e.g. division or addition, rather than multiplication)

19

## The for loop: Exercise

```
int i;
for (i=0; i<=10; i++){
    i += 2;
    printf("value of i is %d\n", i);
}
```

Produces the screen output:

20

## do-while loop

• The only difference between this and a **while** loop is that the condition is checked at the *end* of the loop body, rather than at the beginning.

• General syntax of a **do-while** loop:

```
do {
    statements;
} while (condition);
```

### Example:

produces the screen output:

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

j is 0  
j is 1  
j is 2  
j is 3  
j is 4  
j is 5

The main difference between a **while** and a **do-while** is that the body of the **do-while** will **always** be executed **at least once**, whereas the body of the **while** may not be executed at all if the condition is found to be **false** first time around.

21

## Be careful with loops!

• what do these loops do?

```
for (stars = 0; stars <= 100; stars--) {
    printf("*");
}

for (j = 0; j > 0; j++) {
    printf("*");
}
```

• Note that all loops have 3 parts:

(1) initialisation, (2) condition, (3) update

(**for**: part of syntax; **while**, **do-while**: up to you)

**DO NOT START CODING A LOOP UNTIL YOU KNOW EXACTLY WHAT EACH SHOULD BE!!!**

if not careful: loop won't terminate, won't do anything, won't do what you expect...

22

## Loops are not just for numbers...

```
char ch = 'y';
while (ch == 'y') {
    /* something interesting */
    printf("Go again (y/n)?");
    scanf(" %c",&ch);
}
```

*Note blank space before %c – needed to fix a problem with scanf when making multiple reads...*

```
char ch;
for (ch = 'y'; ch == 'y'; scanf(" %c",&ch)) {
    /* something interesting */
    printf("Go again (y/n)?");
}
```

23

## Multiple-alternative decisions

Sometimes there are multiple possible paths the program could follow.

Example: mark  $\geq 70$ , First Honours;  $60 \leq$  mark  $< 70$ , Second Honours;  $40 \leq$  mark  $< 60$ , Pass; mark  $< 40$ , Fail.

You could write a C program to do this using just **if** statements, or just **if-else** statements (nested). But there is another way:

```
if (mark >= 70) {
    printf("First Honours\n");
} else if (mark >= 60) { /* know mark < 70 here */
    printf("Second Honours\n");
} else if (mark >= 40) { /* know mark < 60 here */
    printf("Pass\n");
} else { /* know mark < 40 here */
    printf("Fail\n");
}
```

24

## Multiple-alternative decisions (contd.)

As usual, there is more than one correct solution:

```
if (mark<40){
    printf("Fail\n");
} else if (mark<60){ /* know mark≥40 here */
    printf("Pass\n");
} else if (mark<70){ /* know mark≥60 here */
    printf("Second Honours\n");
} else{ /* know mark≥70 here */
    printf("First Honours\n");
}
```

25

## Multiple-alternative decisions: switch

This is another flow of control statement. It is not essential as anything that can be done with a **switch** can be done with **if-else** statements.

However it is easier to use and to read.

**switch** general syntax :

```
switch (expression) {
    case const1: statements;
    case const2: statements;
    . . .
    default : statements;
}
```

There can be as many **const**'s as you like. **expression** is evaluated and its value is compared to **const1**, **const2**... If any of them have the same value as **expression**, then the **statements** associated with this **const** (and all subsequent **statements** in the body of the **switch**) are executed. If there is a **default** present, then its **statements** are executed when no matching value is found. If there is no **default** and no matching value, then the **switch** statement does nothing.

26

## Multiple-alternative decisions: switch (contd.)

The cases are not exclusive as this example shows:

```
int i;
for (i = 0; i <= 5; i++) {
    switch(i) {
        case 1:
        case 2: printf("1 or 2\n");
        case 4: printf("4\n");
        default: printf("default\n");
    }
}
```

} default  
1 or 2  
4  
default  
1 or 2  
4  
default  
default  
4  
default  
default

The loop cycles with values of **i** from 0 to 5. 0 gives the default. 1 however triggers **case 1** and rather than this having no effect, this causes all the **printf** statements to be executed. This is because the **switch** selects one **entry point** and then executes all statements following that entry point. When **i** is 4, the entry point is **case 4** and so the last 2 messages are printed.

27

## Multiple-alternative decisions: switch (contd.)

Usually we want to express *mutually exclusive* courses of action, and this can be done by using **break** statements in the **switch**.

```
int i;
for (i = 0; i <= 5; i++) {
    switch(i) {
        case 1:
        case 2: printf("1 or 2\n");
                break;
        case 4: printf("4\n");
                break;
        default: printf("default\n");
                break; /* good practice */
    }
}
```

} default  
1 or 2  
1 or 2  
default  
4  
default

Note: **break** means “immediately halt execution and exit this **compound statement** (loop or switch)”, so it can also be used with loops.

28

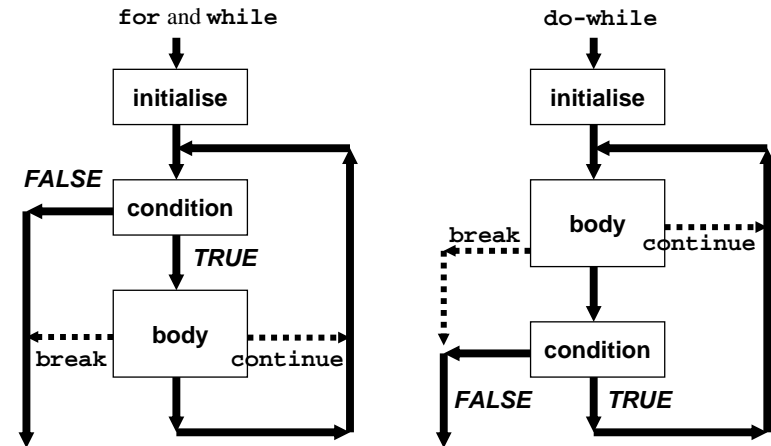
## Multiple-alternative decisions: switch (contd.)

The expression controlling the **switch** can be of any integral type.  
This includes **char**... For example:

```
char ch;
printf("enter your letter: ");
scanf("%c", &ch);
switch(ch) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    printf("A vowel\n");
    break;
  default:
    printf("A Consonant\n");
    break;
}
```

29

## break vs continue, for/while vs do-while



**break** means "immediately halt execution of this loop"

– often useful for complicated/multiple/exceptional loop exit tests

**continue** means "skip to next iteration of this loop"

– occasionally useful when loop body is very complicated

30