Decisions and Loops

Overview

- Conditional Statements
- Loops
- Modifying Loops

2

Objectives

In this section we continue with a number of fundamental topics about the C language. The topics are essentially 'procedural' in nature and if you have experience in Pascal, FORTRAN or some other high level language you should not have many problems. The emphasis is on bringing control and structure into the language.

The topics discussed in this chapter:

- The compound or block statement
- The if statement
- The for loop
- The while loop
- The do-while loop
- Break and continue statements
- Multiple choices, the switch statement
- Goto and labels

Blocks and Statements

- Blocks are pieces of discrete code
- Blocks are bounded with braces {}
- Variables must be declared in the beginning of blocks
- Indent the statements inside blocks

3

The Compound or Block Statement

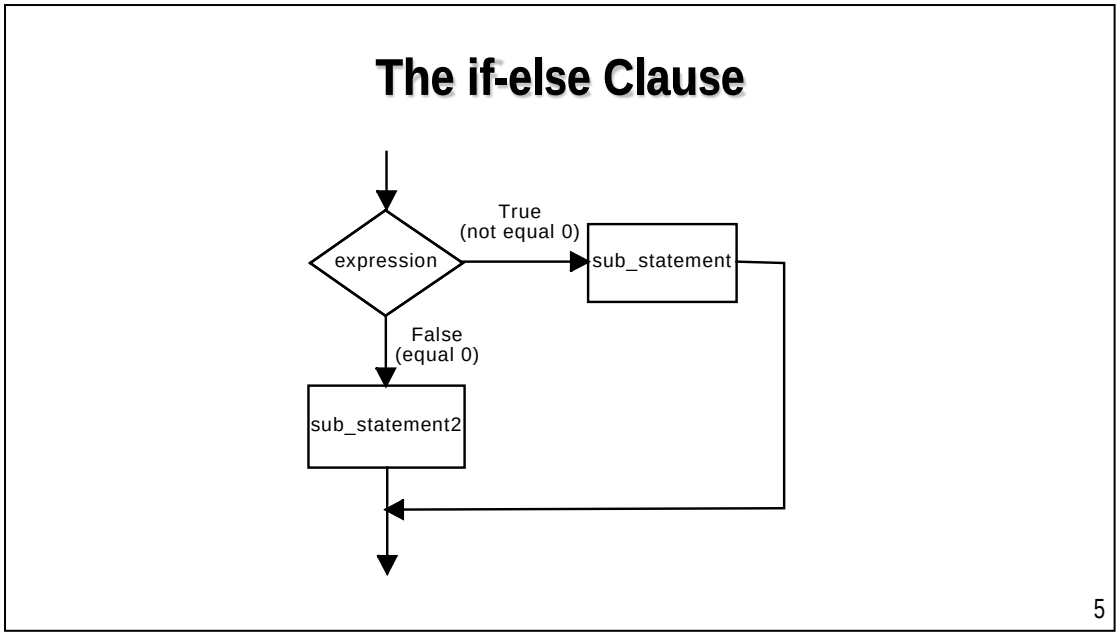
The compound statement can be seen as a number of valid C statements which are related in some way. To be more specific, separate statements which comprise a compound statement are included in braces {}. A compound statement is also known as a block statement.
Local variables which have been defined in a block are destroyed when the block is exited. A declaration is a statement.

Indent new blocks:

```
void main()
{
    int a;
    {
           int b;
           ...
           {
                  int c...
                  ...
           }
           ...
    }
}
```

Conditional Statements

- Conditional statements are used to branch the execution of a program
- Two types of conditional statements
  - if - else
  - switch



The if Statement

The *if* statement determines if a certain block of code needs to be executed or not. If the *if* expression, which is a Boolean expression, evaluates to true the block is executed. If false another block can be executed which is marked with the *else* statement.

The syntax of an *if* statement is:

```
if (<expression>)
  <sub_statement>
else
  <sub_statement2>
```

The <expression> must be of arithmetic or pointer type or of class type which can be converted to one of the first two types. If the <expression> is nonzero the <sub_statement> is executed otherwise the second sub statement is executed. The else block is not required.

```
if (a > b)
{
  printf("a is larger than b\n");
}
else
{
  printf("a is not larger than b");
}
```

The compound block brackets can be left out when the sub_statement has only one expression. The above example could be written as:

```
if (a > b)
  printf("a is larger than b\n");
else
  printf("a is not larger than b");
```

We advise however to always use the braces. Leaving out these brackets is not advisable because of future modifications to one of the sub statements of the *if-else* construction.

Inside one of the sub statements, which is a block, another *if-else* construction can be created. This nesting of *if-else* constructions can make the code very unreadable. This is because every time a new block begins the code is indented. Indenting each time may cause to the text to flow of the screen. Another way of writing the *if-else* clause may prevent this.

```
if (notmarried)
{
    group = 0;
}
else
{
    if (married)
    {
          group = 1;
    }
    else
    {
          if (livingtogether)
          {
                group = 2;
          }
          else
          {
                group = 3;
          }
    }
}
```

Checking these different possible paths can be difficult, the code is starting to become unreadable. One solution for this nesting is writing this code in the *else-if* construction. It is not a different syntax only another way of writing the *if* statement.

```
if (notmarried)
{
    group = 0;
}
else if (married)
{
    group = 1;
}
else if (livingtogether)
{
    group = 2;
}
else
{
    group = 3;
}
```

Example
if ... else

```
void main()
{
  int a = 10;
  int b = 20;

  if ( a < b)
  {
    printf("a is greater than b");
  }
  else
  {
    printf("a is smaller than b");
  }
}
```

6

Conditional Cases

- The switch clause is used for situations where there is a single variable or expression which can have a number of values
- Each value is a different branch
- The “value” expression needs to be a constant expression

7

The switch Statement

The switch statement allows the programmer to transfer control to one of several statements depending on the value of an expression. The switch statement is often used when according to the value of an integer, a different action needs to take place.
The switch statement is almost equal to the *if-else* construction.

The syntax is:

```
switch(<expression>)
{
    case <const_expression1>:
           <sub_statement> [break;]
    case <const_expression2>:
           <sub_statement> [break;]
    case <const_expression3>:
           <sub_statement> [break;]
    default:
           <sub_statement>
}
```

When none of the const_expressions equals the value of the expression, the default sub_statement is executed.

The break statement is optional in the different cases. When it is left out all the following sub_statements are executed until a *break* statement or the end of the *switch* is reached.

The following program shows how the *switch* statement works; it allows an integer to be entered and the value given is printed on the screen.

Example
Switch

```
void main()
{
  int a = 10;

  switch(a)
  {
    case 10:
      printf("value is 10"); break;
    case 11:
      printf("value is 11"); break;
    case 12:
      printf("value is 12"); break;
    default:
      printf("Unexpected value"); break;
  };
}
```

8

```
/*
   Program to show how the switch statement works. Incidentally,
   the use of the break statement is also shown in this example.
   (C) Datasim BV 1995
*/

#include <stdio.h>
#include <stdlib.h>

void main()
{
     char c;
     c = getchar(); /* Gets a character from the keyboard */

     /* Now we test for c using the switch statement */
     switch(c)
     {
            case 'a':
            case 'b':
            case 'c':
                   printf("You have chosen a, b or c\n");
                   break;       /* Exit the switch */
            case 'd':
                   printf("You have chosen the d character\n");
                   break;       /* Exit the switch */
            default:
            /* When no condition is met */
            printf("You choose something that is not a, b, c or d\n");
            break;
     }
}
```

There can be only one *default* label in a *switch* statement.

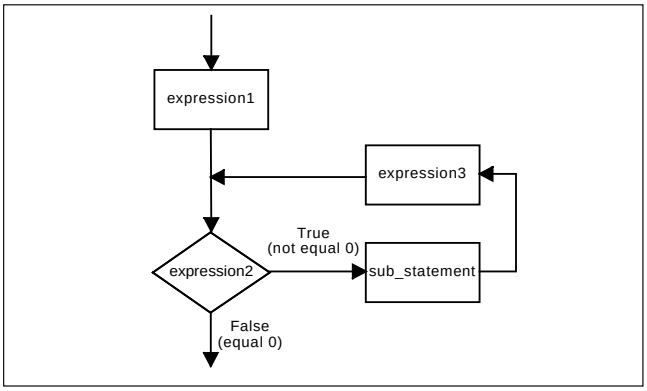
Loop Constructions

- Loops are used for iterations in the code
- Single piece of code is executed several times
- Three types of loops
  - for-next
  - while
  - do-while

9

for Loop

- The for loop is often used when the number of iterations is known



10

For Loop

The syntax of the *for* statement is

```
for (expression1; expression2; expression3)
      <sub_statement>
```

The different expressions serve the following purposes:
expression1: Used for initialising the loop variable
expression2: The stop-criteria of the *for* loop. While this expression yields true, the sub-statement is executed.
expression3: The post iteration expression. After the sub_statement is executed this expression is evaluated. Mostly used for incrementing the loop variable.

Any of the three expressions can be left out.

Examples of for statements

1. Iterating from 0 to 10 and printing the corresponding number.

```
for (j = 0; j <= 10; j++) printf("%d\n", j);
```

2. Creating an infinite loop

```
for (;;)
```

Example
for loop

```
void main()
{
  int cnt;

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

initialization expression

conditional expression

iteration expression

11

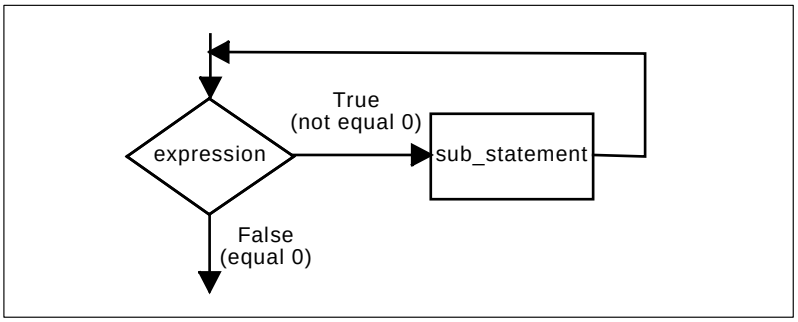
for Loop

- Any of the expressions can be left out
- Smallest expression possible (endless loop):
  - for (;;)

12

while Loop

- while is used when the number of iterations is not known
- while is executed zero or more times



13

While Loop

The *while* statement has a syntax in the form:

```
while (<expression>)
      <sub_statement>
```

In this case the <sub_statement> is executed repeatedly until the value of the <expression> becomes zero. The test takes place before each execution of the sub_statement.

```
while (a > 10)
{
  a--;
}
```

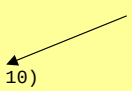
When creating a *do-while* loop do not place a semi-colon ';' after the *while* expression. Placing a semicolon can create an infinite loop.

```
int a = 80;
while (a < 90); /* Can cause problems */
{
  a++;
}
```

The *while* loop has become an infinitive loop. The variable ‘a’ equals 80, the loop checks to see if the value of ‘a’ is smaller than 90. If so it executes the empty statement following the *while* expression (only a semicolon). The loop has become infinitive only because there is an extra semicolon after the while expression. The compiler does not give a warning or error.

Example
while loop

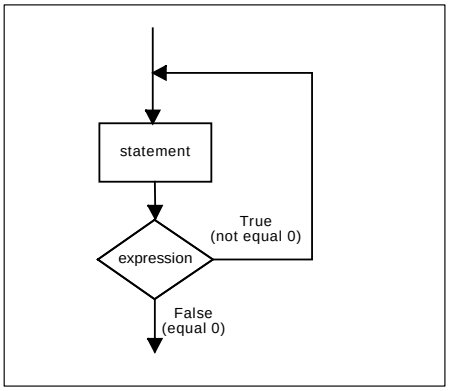
```
void main()
{
  int cnt=0;

  while (cnt < 10)
  {
     printf("value is %d\n", cnt);
     cnt++;
  }
}
```



14

do-while Loop

- The do-while loop is executed one or more times
- Much like the while construction



15

Do While Loop

The *do-while* statement is similar to the REPEAT UNTIL statement in Pascal. The syntax is:

```
do
        <sub_statement>
while (<expression>); /* do not forget the semicolon */
```

The sub statement is executed at least once. After the sub statement is executed the expression is evaluated. If it yields to true the sub statement is executed again until the expression yields to false. The big difference between the normal *while* and the *do-while* loops is that the *while* construction first checks the expression and the *do-while* executes the statement once before checking the expression. In the do-*while* construction the semi-colon is mandatory after the *while* expression.

```
do
{
     a--;
}
while (a >10);
```

Example
do-while loop

```
void main()
{
  int cnt=1;

  do
  {
    printf("value is %d\n", cnt);
    cnt++;
  }
  while (cnt < 10);
}
```

16

Break and Continue

- `break` Used for breaking out of a loop
- `continue` Causes next iteration to take place

17

Break and continue

All the loop constructions discussed before can be stopped even before the expression yields to false by using the *break* statement. When the *break* statement is executed, it breaks the loop.

```
for (counter = 1; counter < 11; counter++)
{
    printf("%d\n",i);
}
```

The *for* loop exits when the expression 'counter < 11' yields to false. Another way of exiting the loop would be by using the *break* statement.

The next example exits the loop when the counter reaches 5:

```
for (counter = 1; counter < 11; counter++)
{
    if(counter == 5)
           break;

    printf("%d\n",i);
}
```

The output:

```
1
2
3
4
```

The loop now prints the numbers of one to four and exits. The *break* statement causes the *for* loop to exit. Execution continues after the *for* sub statement. This construction can be used in the *while* and *do-while* as well.

The *continue* statement causes the next iteration to take place. Take a look at the *for* loop again, only slightly modified:

```
for (counter = 1; counter < 11; counter++)
{
    if(counter == 5)
           continue;

    printf("%d\n",i);
}
```

The output:

```
1
2
3
4
6
7
8
9
10
```

The *break* statement has been changed to *continue*. The loop does not exit at 5 but forces the *for* loop to go to the next iteration. This means that the counter is increased and the stop-criteria of the *for* loop (counter < 11) is executed again. The number 5 is not printed. On the screen a list of the numbers one to four and six to ten will be displayed.

The construction with *break* is often used in message processing systems, where you create an infinite loop that needs to be exited when a quit message is received.

For example:

```
for (;;)
{
   ...
   if ( /* Quit message */ ) break; /* Causes exit the for loop */
   ...
   if ( /* User wants to end */ ) break;
   ...
}
```

Infinite loops are useful when creating event driven software (for example, graphical user interfaces and CAD software).

Goto and Labels

- The goto statement jumps to a certain label
- It should not be used

18

Goto and Labels

A lot of languages have the possibility to jump to a certain location in the code. In most procedural languages this possibility is unavialable. In C it is possible to just jump to a certain position in the code. It is however not recommended to do so because it is breaking up your procedural construction.

The statement that can cause a jump is the *goto* statement. With a *goto* statement it is possible to jump to a certain label. A label is marked by a name followed by a colon. For the name of the colon the same rules apply as for identifiers. After the colon it is mandatory to place at least one statement. There are however a few restrictions to the jump. The label the *goto* statement jumps to must reside in the same function. Again watch out using this statement. It creates spaghetti syntax.

```
{
    ...
    ...
    for(...)
    {
           ...
           ...
           while(...)
           {
                  ...
                  ...
                  if(...)
                  {
                         goto end;
                  }
           }
    }
    ...
    ...
end:;
}
```