

Conditional Execution

Objectives:

Having read this section you should be able to:

1. Program control with if, if-else and switch structures
2. have a better idea of what C understands as true and false.

Program Control:

It is time to turn our attention to a different problem - conditional execution. We often need to be able to choose which set of instructions are obeyed according to a condition. For example, if you're keeping a total and you need to display the message 'OK' if the value is greater than zero you would need to write something like:

```
if (total>0) printf("OK");
```

This is perfectly reasonable English, if somewhat terse, but it is also perfectly good C. The if statement allows you to evaluate a >condition and only carry out the statement, or compound statement, that follows if the condition is true. In other words the printf will only be obeyed if the condition total > 0 is true.

If the condition is false then the program continues with the next instruction. In general the if statement is of the following form:

```
if (condition) statement;
```

and of course the statement can be a compound statement.

Here's an example program using two if statements:

```
#include <stdio.h>

main()
{
    int a , b;

    do {

        printf("\nEnter first number: ");
        scanf("%d" , &a);

        printf("\nEnter second number: ");
        scanf("%d" , &b);

        if (a<b) printf("\n\nFirst number is less than second\n\n");
        if (b<a) printf("\n\nSecond number is less than first\n\n");

    } while (a < 999);
}
```

Here's another program using an if keyword and a compound statement or a block:

```
#include <stdio.h>

main()
{
    int a , b;

    do {

        printf("\nEnter first number: ");
        scanf("%d" , &a);

        printf("\nEnter second number: ");
        scanf("%d" , &b);
```

```

    if (a<b) {
        printf("\n\nFirst number is less than second\n");
        printf("Their difference is : %d\n" , b-a);
        printf("\n");
    }

    printf("\n");

} while (a < 999);
}

```

The if statement lets you execute or skip an instruction depending on the value of the condition. Another possibility is that you might want to select one of two possible statements - one to be obeyed when the condition is true and one to be obeyed when the condition is false. You can do this using the

```

if (condition) statement1;
else statement2;

```

form of the if statement.

In this case statement1 is carried out if the condition is true and statement2 if the condition is false.

Notice that it is certain that one of the two statements will be obeyed because the condition has to be either true or false! You may be puzzled by the semicolon at the end of the if part of the statement. The if (condition) statement1 part is one statement and the else statement2 part behaves like a second separate statement, so there has to be semi-colon terminating the first statement.

Logical Expressions:

So far we have assumed that the way to write the conditions used in loops and if statements is so obvious that we don't need to look more closely. In fact there are a number of deviations from what you might expect. To compare two values you can use the standard symbols:

```

> (greater than)
< (less than)
>= (for greater than or equal to )
<= (for less than or equal to)
== (to test for equality).

```

The reason for using two equal signs for equality is that the single equals sign always means store a value in a variable - i.e. it is the assignment operator. This causes beginners lots of problems because they tend to write:

```

if (a = 10) instead of if (a == 10)

```

The situation is made worse by the fact that the statement if (a = 10) is legal and causes no compiler error messages! It may even appear to work at first because, due to a logical quirk of C, the assignment actually evaluates to the value being assigned and a non-zero value is treated as true (see below). Confused? I agree it is confusing, but it gets easier. . .

Just as the equals condition is written differently from what you might expect so the non-equals sign looks a little odd. You write not equals as !=. For example:

```

if (a != 0)

```

is 'if a is not equal to zero'.

An example program showing the if else construction now follows:

```

#include <stdio.h>

main ()
{

```

```

int num1, num2;

printf("\nEnter first number ");
scanf("%d",&num1);

printf("\nEnter second number ");
scanf("%d",&num2);

if (num2 ==0) printf("\n\nCannot divide by zero\n\n");
else      printf("\n\nAnswer is %d\n\n",num1/num2);
}

```

This program uses an if and else statement to prevent division by 0 from occurring.

True and False in C:

Now we come to an advanced trick which you do need to know about, but if it only confuses you, come back to this bit later. Most experienced C programmers would wince at the expression `if(a!=0)`.

The reason is that in the C programming language doesn't have a concept of a Boolean variable, i.e. a type class that can be either true or false. Why bother when we can use numerical values. In C true is represented by any numeric value not equal to 0 and false is represented by 0. This fact is usually well hidden and can be ignored, but it does allow you to write

`if(a != 0)` just as `if(a)`

because if a isn't zero then this also acts as the value true. It is debatable if this sort of shortcut is worth the three characters it saves. Reading something like

`if(!done)`

as 'if not done' is clear, but `if(!total)` is more dubious.

Using break and continue Within Loops:

The break statement allows you to exit a loop from any point within its body, bypassing its normal termination expression. When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. The break statement can be used with all three of C's loops. You can have as many statements within a loop as you desire. It is generally best to use the break for special purposes, not as your normal loop exit. break is also used in conjunction with functions and >case statements which will be covered in later sections.

The continue statement is somewhat the opposite of the break statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. In while and do-while loops, a continue statement will cause control to go directly to the test condition and then continue the looping process. In the case of the for loop, the increment part of the loop continues. One good use of continue is to restart a statement sequence when an error occurs.

```

#include <stdio.h>

main()
{
    int x ;

    for ( x=0 ; x<=100 ; x++) {
        if (x%2) continue;
        printf("%d\n" , x);
    }
}

```

Here we have used C's modulus operator: %. A expression:

`a % b`

produces the remainder when a is divided by b; and zero when there is no remainder.

Here's an example of a use for the break statement:

```
#include <stdio.h>

main()
{
    int t;

    for ( ;; ) {
        scanf("%d", &t) ;
        if ( t==10 ) break ;
    }
    printf("End of an infinite loop...\n");
}
```

Select Paths with switch:

While if is good for choosing between two alternatives, it quickly becomes cumbersome when several alternatives are needed. C's solution to this problem is the switch statement. The switch statement is C's multiple selection statement. It is used to select one of several alternative paths in program execution and works like this: A variable is successively tested against a list of integer or character constants. When a match is found, the statement sequence associated with the match is executed. The general form of the switch statement is:

```
switch(expression)
{
    case constant1: statement sequence; break;
    case constant2: statement sequence; break;
    case constant3: statement sequence; break;
    .
    .
    .
    default: statement sequence; break;
}
```

Each case is labelled by one, or more, constant expressions (or integer-valued constants). The default statement sequence is performed if no matches are found. The default is optional. If all matches fail and default is absent, no action takes place.

When a match is found, the statement sequence associated with that case are executed until break is encountered.

An example program follows:

```
#include <stdio.h>

main()
{
    int i;

    printf("Enter a number between 1 and 4");
    scanf("%d",&i);

    switch (i)
    {
        case 1:
            printf("one");
            break;
        case 2:
            printf("two");
            break;
        case 3:
            printf("three");
    }
```

```
    break;
    case 4:
        printf("four");
        break;
    default:
        printf("unrecognized number");
} /* end of switch */

}
```

This simple program recognizes the numbers 1 to 4 and prints the name of the one you enter. The switch statement differs from if, in that switch can only test for equality, whereas the if conditional expression can be of any type. Also switch will work with only int and char types. You cannot for example, use floating-point numbers. If the statement sequence includes more than one statement they will have to be enclosed with {} to form a compound statement.