Decision making structures require that the programmer specify one or more conditions to be evaluated by the program, along with statement(s) to be executed if the condition is determined to be `true`, and optionally, another set of statements to be executed if the condition is `false`.

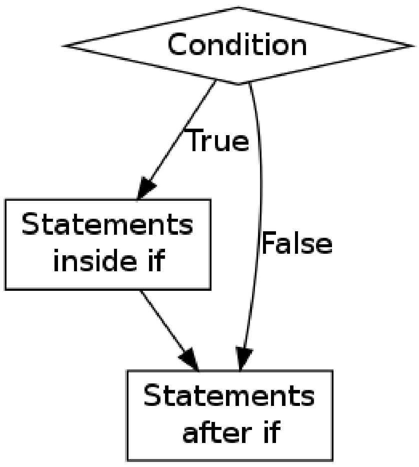


Figure 1: Control flow: if

1 If Statement

C uses the keyword **if** to execute one or more statements when the condition is `true`. The **if** statement controls conditional execution. The body of an **if** statement is executed if the value of the expression is `true` (non-zero).

Syntax

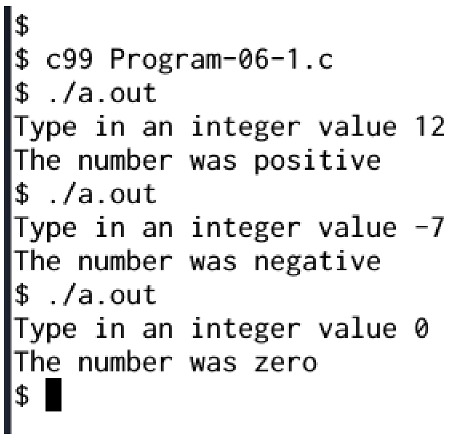
```
if (boolean_expression) {
  statement(s);
}
```

If the boolean expression evaluates to `true`, then the block of code inside the **if** statement will be executed. If the boolean expression is `false`, then the code after the end of the **if** statement will be executed.

Example

The next program checks whether the given number is positive, negative or zero. The output can be seen in Figure 2.

```c
#include <stdio.h>
int main() {
    int num;
    printf("Type in an integer value ");
    scanf("%d", &num);
    if (num == 0) {
        printf("The number was zero\n");
    }
    if (num > 0) {
        printf("The number was positive\n");
    }
    if (num < 0) {
        printf("The number was negative\n");
    }
    return 0;
}
```



```
$
$ c99 Program-06-1.c
$ ./a.out
Type in an integer value 12
The number was positive
$ ./a.out
Type in an integer value -7
The number was negative
$ ./a.out
Type in an integer value 0
The number was zero
$
```

Figure 2: Simple if

True/False in C

Originally C did not have a boolean data type. C programming language treats any `non-zero` and `non-null` values as `true.zero` and `null` are synonymous with `false`. Even `c99` adds a macro which essentially is a wrapper around the fact boolean is integral value.

So you may see code like **while** (1) to set up an infinite loop. Modern programmers should use the symbolic constants `true` and `false` but should be prepared to read code where 1 (rarely some other non-zero value) stands for `true` and 0 for `false`.

If . . . Else Statement

C implements two-way selection with the **if** ... **else** statement. It is a composite statement used to make a decision between two atlernatives. In other words, if the condition is true one set of statements is executed and if false, different piece of code is executed.

Syntax

```
if (boolean_expression) {
  A-statement(s);
} else {
  B-statement(s);
}
```

In the above A-statement(s) are executed if the boolean_expression is true; otherwise B-statement(s) are executed.

To repeat, the **if else** statement is a two way branch, it means do one thing or the other. When it is executed, the condition is evaluated and if it has the value 'true' (i.e. not zero) then **if** block statements will execute. If the condition is 'false' (zero) then **else** block statements will execute.

```
if (yy % 4 == 0) {
  nDays = 366;
} else {
  nDays = 365;
}
```

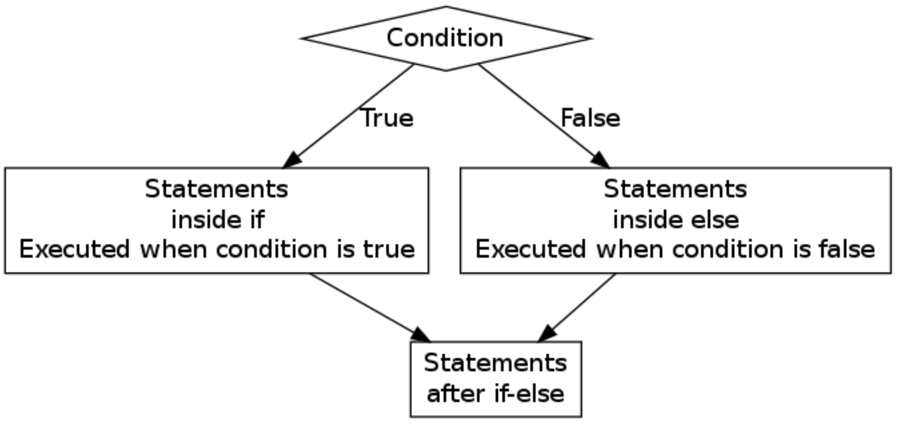
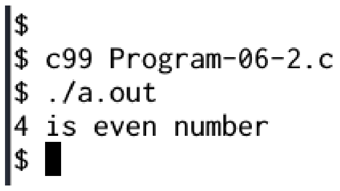

Condition
True
False
Statements inside if Executed when condition is true
Statements inside else Executed when condition is false
Statements after if-else

Figure 3: Control flow: if else

It is unnecessary to test whether yy is not divisible by 4 in the else block; since it is implied by the *if-else* structure. That is, the block would be executed *only* when the remainder is NOT zero.

Example

```
#include <stdio.h>
int main() {
    int a = 4;
    if (a % 2 == 0) {
        printf("%d is even number\n", a);
    } else {
        printf("%d is odd number\n", a);
    }
    return 0;
}
```



```
$
$ c99 Program-06-2.c
$ ./a.out
4 is even number
$ █
```

Figure 4: If else output

Complex Conditions

So far we have written **if** statements with simple conditions. They were simple boolean expressions. Similar to combining many operands with arithmetic operators for complex expressions, we can combine boolean expressions to create complex boolean expressions.

We saw an example where we tested yy % 4 == 0 to check if yy is a leap year. But as you probably know, the rule is more than that. A year is a leap year if it is divisible by 400 or is not divisible by 100 but divisible by 4.

To express the above in code we need to use logical operators.

Logical Operators

C has three logical operators for combining logical expressions and creating new logical expressions: *not*, *and*, and *or*. A common way to show logical relationships is using truth tables. Truth tables list the values that each operand can assume and the result of the expression.

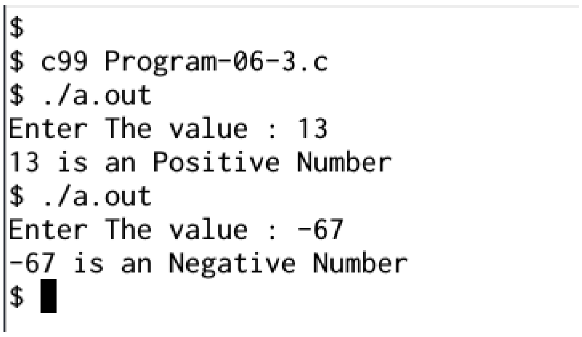
Logical NOT

The *not* operator is **!**. It is an unary operator. It changes a `true` value to `false` and `false` value to `true`. That is, it toggles the value of the boolean expression it operates on.

Example

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter The value : ");
    scanf("%d", &num);
    if (!(num >= 0))
        printf("%d is an Negative Number \n", num);
    else
        printf("%d is an Positive Number \n", num);
    return 0;
}
```

Of course the above example can be simplified. Line 6 can be rewritten as: `if (num < 0)` ...



```
$
$ c99 Program-06-3.c
$ ./a.out
Enter The value : 13
13 is an Positive Number
$ ./a.out
Enter The value : -67
-67 is an Negative Number
$ █
```

Figure 5: Using NOT

Logical AND

The *and* operator is &&. It is a binary operator, used to combine multiple expresions, the result is `true` when *all* expressions are `true`; `false` otherwise.

Example

```c
#include <stdio.h>
int main() {
    int T1_marks, T2_marks;
    printf("Enter Two Test Marks : ");
    scanf("%d %d", &T1_marks, &T2_marks);
    if (T1_marks > 50 && T2_marks > 50)
        printf("Qualified \n");
    else
        printf("Not Qualified \n");
    return 0;
}
```

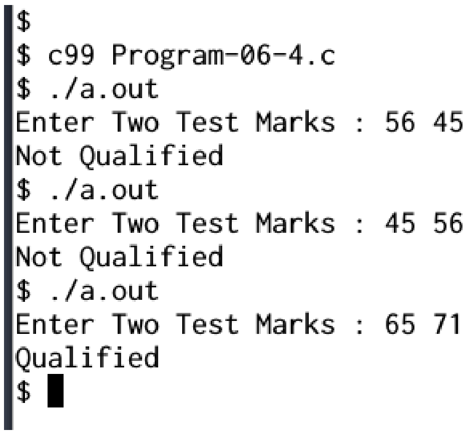
The behaviour and output of this program is identical to the next code, where we rewrite the condition using ||.

Logical OR

The *or* operator is ||. It is a binary operator, used to combine multiple expressions. The result is `false` if all expressions are `false`, it is `true` in all other cases. Stated another way, it returns `true` if *any one* of the component expressions is `true`.

Example

```
#include <stdio.h>
int main() {
    int T1_marks;
    int T2_marks;
    printf("Enter Two Test Marks : ");
    scanf("%d %d", &T1_marks, &T2_marks);
    if (T1_marks < 50 || T2_marks < 50)
        printf("Not Qualified \n");
    else
        printf("Qualified \n");
    return 0;
}
```



```
$
$ c99 Program-06-4.c
$ ./a.out
Enter Two Test Marks : 56 45
Not Qualified
$ ./a.out
Enter Two Test Marks : 45 56
Not Qualified
$ ./a.out
Enter Two Test Marks : 65 71
Qualified
$ 
```

Figure 6: Using Logical Operators

Short-circuit Evaluation

Computer languages use short-circut method to evaluate the binary logical relationships, it does not need to complete the evaluation. In other words, it operates in a short-circuit fashion and stops the evaluation as soon asit knows the final result for sure.

So if the first operand of a logical *and* expression is `false`, the second half of the expression is not evaluated because it is apparent that the result must be `false`. Again, with the *or* expression, if the first operand is `true`, then there is no need to evaluate the second half of the expression, so the resulting value is set to `true` immediately.

Recall the example of the leap year condition: a year is a leap year if it is divisible by 400 or it is not divisible by 100 but divisible by 4.

```
isleap = (yy % 400 == 0) || ((yy % 100 != 0) && (yy % 4 == 0));
```

yy is 2000 The first condition is `true`. So we need not evaluate what follows the ||, as (`true` || any) is `true`. That is 2000 is a leap year.

yy is 1900 The first condition is `false`. So what follows the OR needs to be evaluated. The first part of that is `false`. So the part following AND is not evaluated as (`false` && any) is `false`. The whole expression simplifies to (`false` || `false`) ⇒ `false`. ⇒ 1900 is NOT a leap year.

yy is 2012 The first condition is `false`, but as it is followed by an OR the rest needs to be evaluated. That is also a complex condition. The first part is `true`. Still the part following the AND needs to be evaluated; that part is `true`. So the whole expressions simplifies to

`false` || (`true` && `true`)

which is `true`. That is 2012 is a leap year.

yy is 2014 The first condition is `false`, but as it is followed by an OR the rest needs to be evaluated. That is also a complex condition. The first part is `true`. Still the part following the AND needs to be evaluated; that part is `false`. So the whole expression simplifies to

`false` || (`true` && `false`)

which is `false`. That is 2014 is NOT a leap year.

Nested If Statement

An **if** statement within an another **if** statement is termed as *nested-if* statement. When the outer **if** statement is satisfied then it will check for the inner **if** statement and execute the block of code.

Syntax

```
if (A_boolean_expression) {    // Outer if
   if (B_boolean_expression) { // Inner if
      Alpha_statement;
   } else {
      Beta_statement;
   }
} else {
   Gamma_statement;
}
```

There is nothing special about a nested-if. An **if** statement has a set of statements to be executed when the condition is true and optionally another set of statements to be executed when the condition is false. One or more of these statements in question can be an **if** statement itself.

In above, the condition A_boolean_expression is checked first. If it is true, then the program control goes inside the braces and executes the next statement: which checks the next condition (inner if) namely B_boolean_expression. If it is also true then it executes the block of statements associated with it, namely Alpha_statement; if not the Beta_statement is executed; if the outer if is itself false, execution continues directly with Gamma_statement.

We can build a Truth table to explain the flow:

A_boolean_expression	B_boolean_expression	Action
True	True	Alpha_statement
True	False	Beta_statement
False	Don't care	Gamma_statement

Example

Let us revisit the complex condition we wrote for the leap year. Let us rewrite it as a nested-conditional statement.

```
isleap = (yy % 400 == 0) || ((yy % 100 != 0) && (yy % 4 == 0));
```

```
    if (yy % 400 == 0)
        isleap = true;
    else
        if (yy % 100 == 0)
            isleap = false;
        else
            if (yy % 4 == 0)
                isleap = true;
            else
                isleap = false;
```

Trace the above for the four values, 2000, 1900, 2012, 2014 and convince yourself they are equivalent. Note that we can simplify the above code somewhat.

```
    if (yy % 400 == 0)
        isleap = true;
    else
        if (yy % 100 == 0)
            isleap = false;
        else
            isleap =  (yy % 4 == 0);
```

Such use of boolean expressions is a good idea.

Another example

```
#include <stdio.h>
int main() {
    int num;
    printf("\n Enter Number :");
    scanf("%d", &num);
    if (num >= 0) {
```

```c
        printf("%d is an Positive Number \n", num);
        if (num % 2 == 0)
            printf("%d is an Even Number \n", num);
        else
            printf("%d is an Odd Number \n", num);
    } else {
        printf("%d is an Negative Number \n", num);
    }
    return 0;
}
```

If . . . Else Ladder

It is a conditional statement which is used to check a series of conditions. Each condition is associated with a set of statements. The conditions are checked one after another. The code associated with the first true condition is executed. And then control goes out of the ladder.

Syntax

```c
    if (boolean_expression1) {
        aStatement;
    } else if (boolean_expression2 ) {
        bStatement;
    } else if (boolean_expression3) {
        cStatement;
    } else {
        catchall_statement;
    }
```

Example

```c
#include <stdio.h>
int main() {
    int avg;
    printf("Enter The Average Percentage : ");
    scanf("%d", &avg);
```

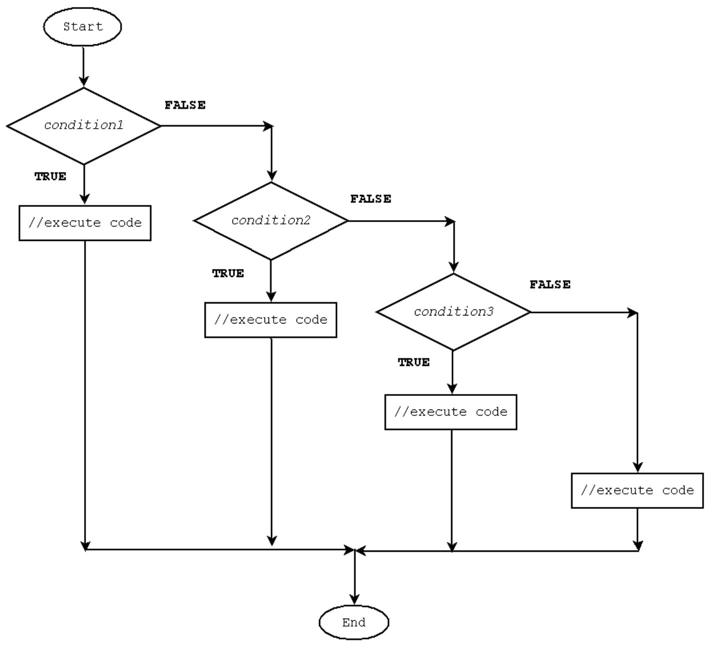


Figure 7: Flowchart if else ladder

```
    if (avg >= 60)
        printf("Passed in First Division\n");
    else if (avg >= 50)
        printf("Passed in Second Division\n");
    else if (avg >= 40)
        printf("Passed in Third Division \n");
    else
        printf("Failed");
    return 0;
}
```

Switch case

The switch statement is similar to the **if** . . . **else** ladder statement. The switch makes one selection when there are several choices to be made.

But the major difference between switch . . . case, and a ladder is that the conditional expression must evaluate to an integer.

The general form of switch statement is shown below:

Syntax

```
switch (expression) {
  case constant1:
    statement 1;
    break;

  case constant2:
    statement 2;
    break;

  case constantn:
    statementn;
    break;

  default:
    statement;
}
```

The key word case is followed by an integer or a character constant. Each constant in each case must be different from all others, that is, each case constant must be distinct.

The **break** statement is used inside each case of the switch, causes an immediate exit from the switch statement; and execution continues from the statement after the switch statement.

When we execute the switch statement, first the expression is evaluated. The value of expression is then compared one-by-one, with constant1, constant2 . . . constantn. When a match is found, the program executes the statements corresponding to that case.

The execution continues from that point till either a **break** statement is found or the switch statement is completed. The **break** causes an immediate exit from the switch construct.

Example

```
#include <stdio.h>
int main() {
    int i = 1;
```

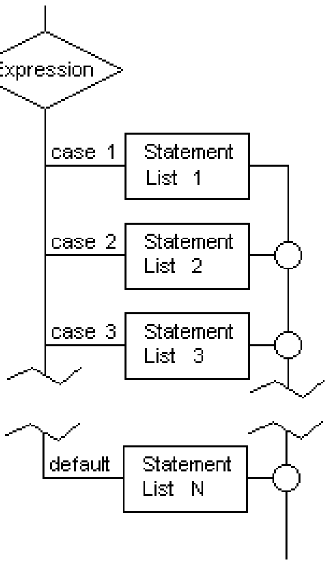


Figure 8: Flowchart: Switch Case

```
    printf("1. choice 1\n");
    printf("2. choice 2\n");
    printf("3. choice 3\n\n");
    printf("select option between 1 and 3: ");
    scanf("%d", &i);
    printf("\n");
    switch(i) {
        case 1 :
            printf("you selected first choice");
            printf("\n");
        case 2 :
            printf("you selected second choice");
            printf("\n");
        case 3 :
            printf("you selected third choice");
            printf("\n");
            break;
        default:
            printf("1 -- 3 please\n");
            break;
    }
}
```

If the option selected is 1, then the output of this program is . . .

you selected first choice

you selected second choice

you selected third choice

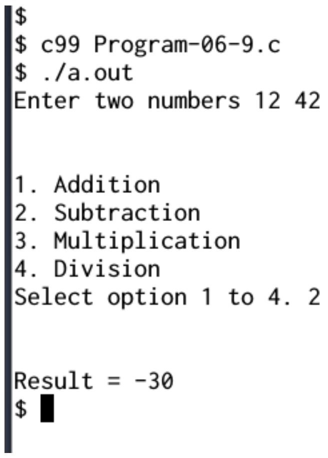
Missing Break in Switch

Because there is no **break** statement in the corresponding cases, the execution of the switch statement continues to all subsequent cases (including default if exist).

Example

The following program reads two numbers and performs selected arithmetic operation.

```
#include <stdio.h>
int main() {
    int a;
    int b;
    int c;
    int opt;
    printf("Enter two numbers ");
    scanf("%d %d", &a, &b);
    printf("\n\n");
    printf("1. Addition\n");
    printf("2. Subtraction\n");
    printf("3. Multiplication\n");
    printf("4. Division\n");
    printf("Select option 1 to 4. ");
    scanf("%d", &opt);
    printf("\n\n");
    switch (opt) {
        case 1:
            c = a + b;
            printf("Result = %d\n", c);
            break;
        case 2:
            c = a - b;
            printf("Result = %d\n", c);
            break;
        case 3:
            c = a * b;
            printf("Result = %d\n", c);
            break;
        case 4:
            c = a / b; /* Integer Division */
            printf("Result = %d\n", c);
            break;
        default:
            printf("Selected wrong option \n");
            break;
    }
    return 0;
}
```



```
$
$ c99 Program-06-9.c
$ ./a.out
Enter two numbers 12 42


1. Addition
2. Subtraction
3. Multiplication
4. Division
Select option 1 to 4. 2


Result = -30
$ █
```

Figure 9: switch case calculator