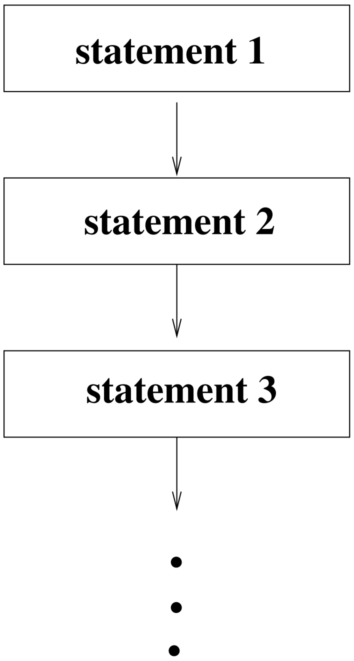
Selection Control Structures

- Flow of control

  - The order in which the computer executes statements in a program

- Sequential control

  - Statements are executed one after the other



- Control structure

  - A statement used to alter the sequential flow of control (see our discussion in Chapter 1)

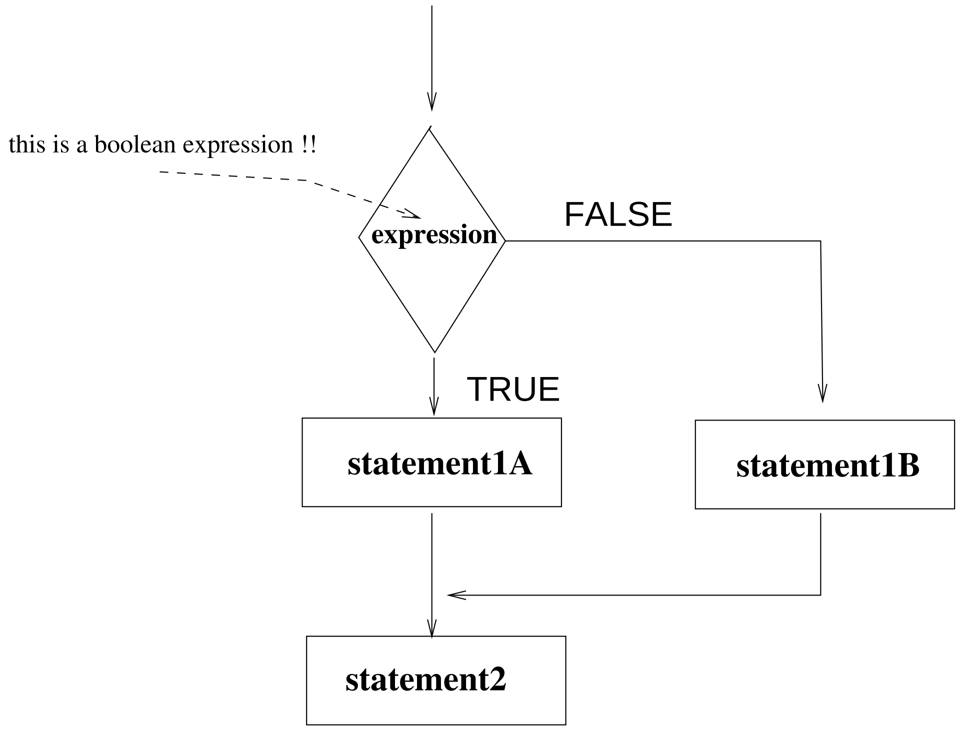
  - The *selection structure*, and the *loop structure* are examples of control strucures

Selection structures

- The *if-then-else* form

```
if ( Expression )
   Statement1A
else
   Statement1B
```

  - If the condition in parentheses (expression) is true, then perform one action, else perform a different action



- Boolean expressions are either true (which is represented by any nonzero number) or false (which is represented by the number 0)

- *Statement1* is called the *then-clause* while *Statement2* is called the *else-clause*

```
if (number > 0)
  cout << "The number is positive" << endl;
else
  cout << "The number is either negative or zero" << endl;
```

- Blocks (compound statements)

  - *Statement1A* and *Statement1B* may correspond to a block of statements

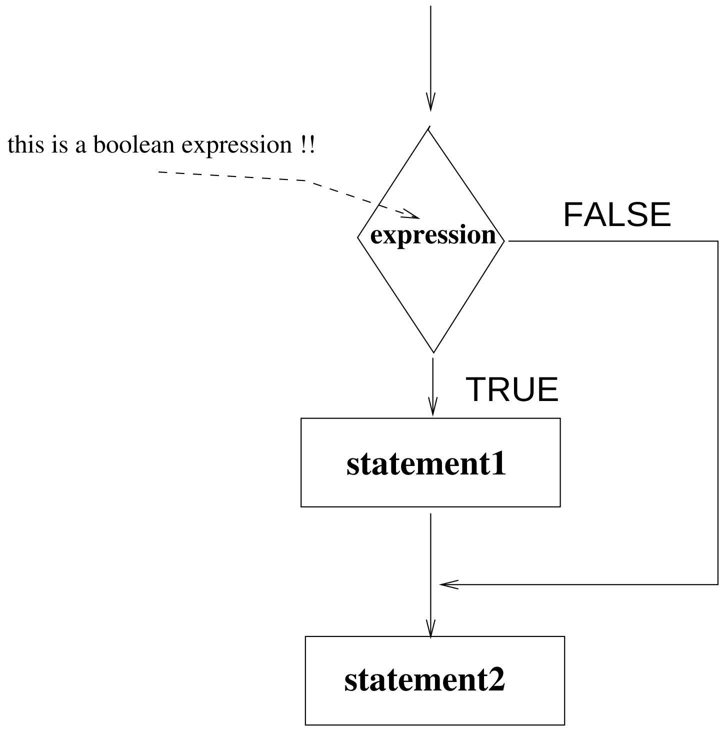
  - Block of statements must be encloded in { }

```
if (number > 0)
{
  cout << "The number is : " << number << endl;
  cout << "The number is positive" << endl;
}
else
{
  cout << "Not a positive number" << endl;
  cout << "The number is either negative or zero" << endl;
}
```

- The *if-then* form

  - If the condition in parentheses (expression) is true, then perform some action; otherwise, don't do anything

if (Expression **)**
Statement




```
if (age < 18)
   cout << "Not an eligible ";
cout << "voter." << endl;
```

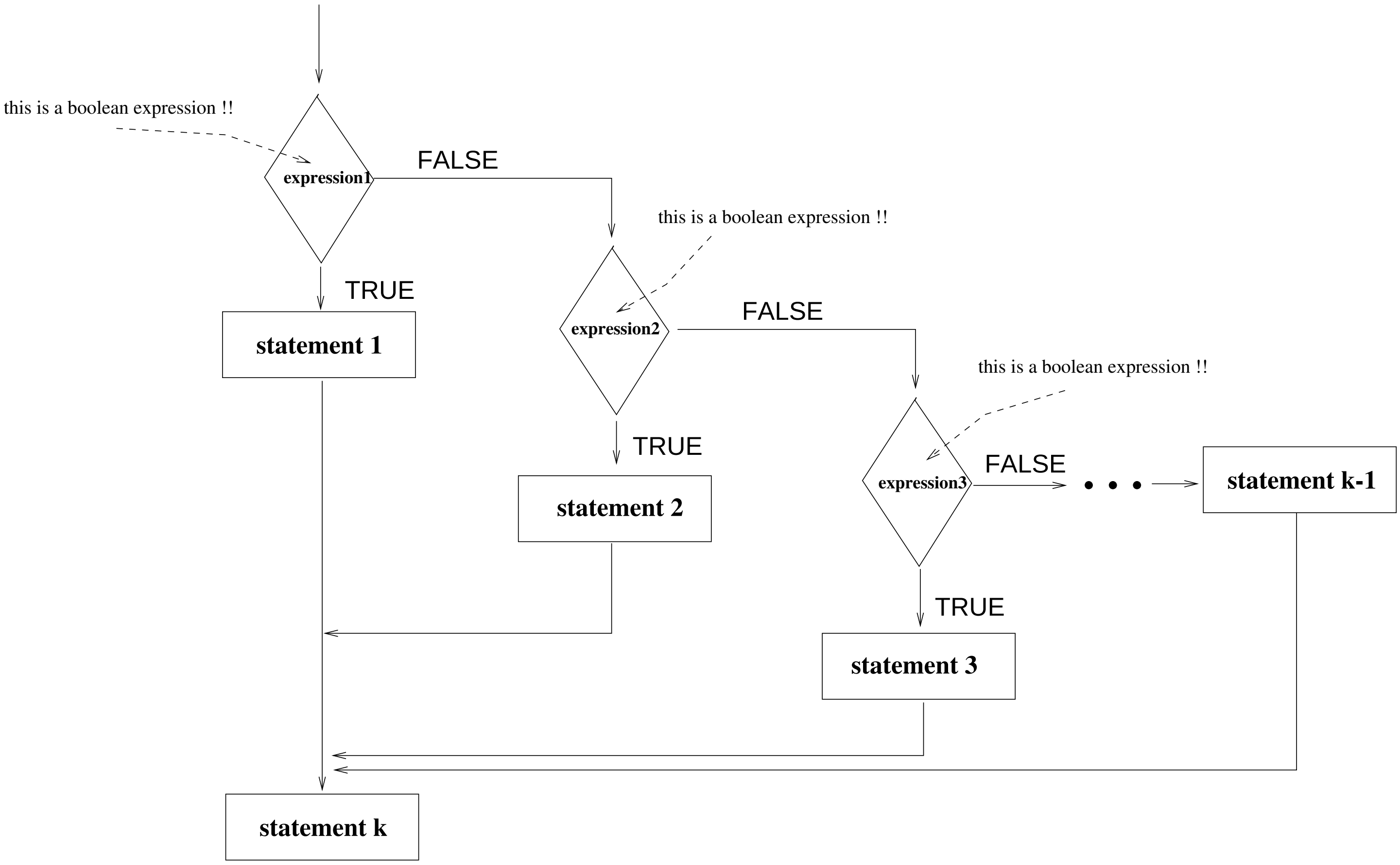
- *Statement* may correspond to a block of statements (must be enclosed in { })

```
if (age < 18)
{
  n = 18 - age;
  cout << "Not an eligible ";
}
cout << "voter." << endl;
```

- Nested *if* statements

  - Problems which involve multiple decisions can be coded using nested *if* statements

```
if ( Expression1 )
   Statement1
else if Expression2 )
   Statement2
else if Expression3 )
   Statement3
  .
  .
  .
else
   Statementk
```



```
if (number > 0)
{
  cout << "The number is : " << number << endl;
  cout << "The number is positive" << endl;
}
else if (number < 0)
{
  cout << "The number is : " << number << endl;
  cout << "The number is negative" << endl;
}
else
{
  cout << "The number is : " << number << endl;
  cout << "The number is equal to zero" << endl;
}
```

- Nested *if* structures yield less comparisons (more efficient code)

- Consider the following alternative implementation
(you need to make 3 comparisons no matter what value is stored in "number")

```
if (number > 0)
{
  cout << "The number is : " << number << endl;
  cout << "The number is positive" << endl;
}
if (number < 0)
{
  cout << "The number is : " << number << endl;
  cout << "The number is negative" << endl;
}
if (number == 0)   // the symbol '==' means equal to
{
  cout << "The number is : " << number << endl;
  cout << "The number is equal to zero" << endl;
}
```

- The body of an *if* statement can contain other *if* statements

```
if (sum > 40)
    if (average < 20)
        cout << "The sum is > 40 and the average is < 20" << endl;
    else
        cout << The sum is > 40 and the average is <= 20" << endl;
else
  cout << "The sum is <= 40" << endl;
```

- The dangling else

  - When *if* statements are nested, it is often confusing to associate an *if* with its corresponding *else*

```
if (sum > 40)
  if (average < 20)
    cout << "The sum is > 40 and the average is < 20" << endl;
  else
    cout << "The sum is <= 40"  << endl;
```

  - *Rule:* in the absence of braces, an *else* is always paired with the closest preceding *if* that doesn't already have an *else* paired with it !!

  - So, does it make sense to rewrite the above segment of code as follows ?

```
if (sum > 40)
  if (average < 20)
    cout << "The sum is > 40 and the average is < 20" << endl;
else
  cout << "The sum is <= 40" << endl;
```

  - *No* !!! (changing the indentation does not make any difference !!!)

  - So, what is the solution ? Use braces !!

```
if (sum > 40)
{
  if (average < 20)
    cout << "The sum is > 40 and the average is < 20" << endl;
}
else
  cout << "The sum is <= 40" << endl; average
```

  - The braces indicate that the *else* statement belongs to the outer *if* statement

Logical (or Boolean) expressions

- It is an expression whose result is either *true* or *false*

sum > 40, *average < 20*, *(sum-average) < 10*

- Creating logical expressions using relational operators

Relational Operators	
==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

if (denom ! = 0)
result = num / denom;

<Table from page 192>

- Common mistakes using the '==' operator

- Do not confuse the assignment operator '=' with the '==' operator

* *x=3* means assign the value *3* to the variable *x*
* *x==3* means check if *x* is equal to *3*

* Which message do you think will be printed on the screen if the following segment of code is executed ?

```
cin >> n;
if(n=3)
  cout << "n equals 3";
else
  cout << "n doesn't equal 3";
```

* This segment of code will always print the message "n equals 3" !! Why ??

* When assignments are treated as logical expressions, they have the value of the expression at the right hand side (nonzero values correspond to *true* and zero values correspond to *false*)

- Comparing numbers for equality

* if *x,y* are integers, you check equality using the statement: *if (x == y)*

* if *w,z* are floats, you **never** check equality using the statement: *if (w == z)*

* In this case, you need to compare the absolute value of the difference of the two numbers with a very small number (e.g., *if (fabs(w-z) < 0.00001)*)

* This is because real numbers cannot be represented exactly in the memory of the computer (truncation or rounding take place during computations)

```
oneThird = 1.0 / 3.0;
x = f2oneThird + f2oneThird + f2oneThird;
```

* What is the value of *x* ? (*x=0.999999*)

- Involving characters in a logical expression

  - Many times, it is useful to compare characters in an expression (e.g., suppose that you want to sort a list of names alphabetically)

  - In each language, there is a predefined ordering for all the characters (each character is associated with an integer value; try *cout << int('A');* to print the integer value associated with 'A')

  - The following ordering is true:

'0' '1' ... '9' 'A' 'B' ... 'Z' 'a' 'b' ... 'z'

- According to this ordering:

’0’ < ’1’ < ... < ’9’ < ’A’ < ’B’ < ... < ’Z’ < ’a’ < ’b’ < ... ’z’

- The following are valid logical expressions

’M’ < ’R’ (true), *’m’ < ’r’* (true), *’m’ < ’R’* (false), *’0’ < ’1’* (true)

- Creating logical expressions using logical operators

  - Logical operators are used to construct more complex logical expressions by combining a number of logical expressions together

- Fundamental logical operators

<table>
<tr><th colspan="2">Logical Operators</th></tr>
<tr><td>&&</td><td>logical AND</td></tr>
<tr><td>||</td><td>logical OR</td></tr>
<tr><td>!</td><td>logical NOT</td></tr>
</table>

  - Truth table for the logical AND operator

    * The logical AND of two logical expressions is true if both expressions are true

<Table from page 194>

(finalScore > 90) && (midtermScore > 70), (x - y < 1) && (y > 10)

- Truth table for the logical OR operator

  * The logical OR of two logical expressions is true if at least one of the expressions is true

<Table from page 195>

(midtermGrade == ’A’) || *(finalGrade == ’A’)*, *(x < 10)* || *(y - 2 < 20)*

- Truth table for the logical NOT operator

  * The logical NOT of a logical expression is true if the logical expression is false

<Table from page 195>

!(hours > 40) is equivalent to *hours <= 40*

- Note that the same logical expression can be written in many different ways !!

!(a == b) is equivalent to *a != b*
!(a == b || *a == c)* is equivalent to *(a != b) && (a != c)*
!(a == b && c > d) is equivalent to *(a != b)* || *(c <= d)*