If Control Construct

A mechanism for deciding whether an action should be taken

A Boolean Type

- C++ contains a type named **`bool`**
- Type **`bool`** has two symbolic constants
  - **`true`**
  - **`false`**
- Boolean operators
  - The AND operator is **`&&`**
  - The OR operator is **`||`**
  - The NOT operator is **`!`**
- Warning
  - **`&`** and **`|`** are also operators

A Boolean Type

- Example logical expressions

```
bool P = true;
bool Q = false;
bool R = true;
bool S = (P && Q);
bool T = ((!Q) || R);
bool U = !(R && (!Q));
```

Relational Operators

- Equality operators
  - `==`
  - `!=`
- Examples
  - `int i = 32;`
  - `int k = 45;`
  - `bool q = (i == k);`
  - `bool r = (i != k);`

Relational Operators

- Ordering operators
  - `<`
  - `>`
  - `>=`
  - `<=`
- Examples
  - `int i = 5;`
  - `int k = 12;`
  - `bool p = (i < 10);`
  - `bool q = (k > i);`
  - `bool r = (i >= k);`
  - `bool s = (k <= 12);`

Operator Precedence

- Precedence of operators (from highest to lowest)
  - Parentheses
  - Unary operators
  - Multiplicative operators
  - Additive operators
  - Relational ordering
  - Relational equality
  - Logical and
  - Logical or
  - Assignment

Operator Precedence

- Consider

  ```
  5 * 15 + 4 == 13 && 12 < 19 || !false == 5 < 24
  ```

  It's not a good practice to write such code.
- However, it is equivalent to

  ```
  ((((5 *15) + 4) == 13) && (12 < 19))
      ||
  ((!false) == (5 < 24))
  ```

Conditional Constructs

- Provide
  - Ability to control whether a statement list is executed
- Two constructs
  - If statement
    - if
    - if-else
    - if-else-if
  - Switch statement
    - Multi-way selection

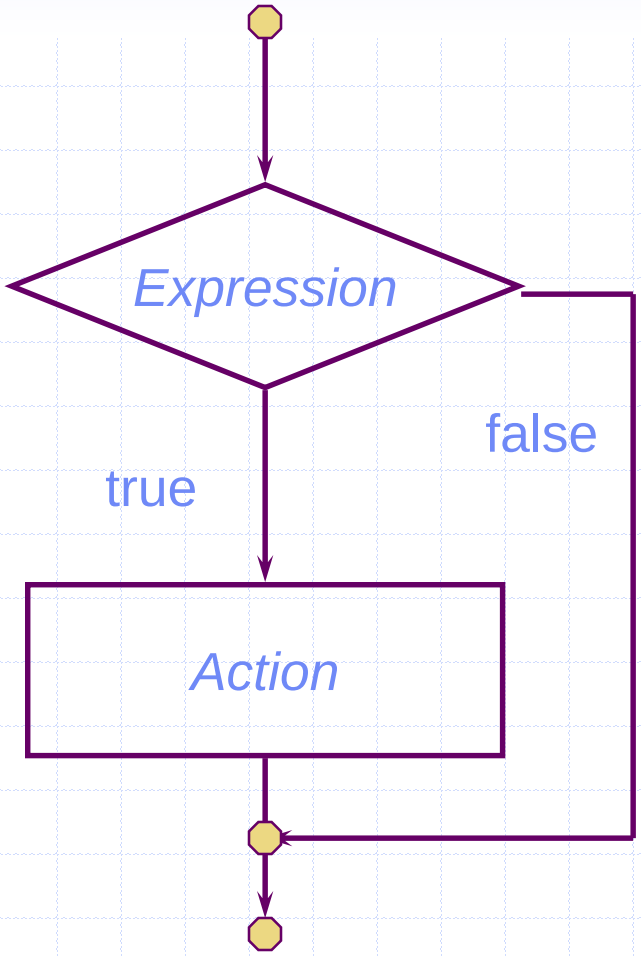
The Basic If Statement

- Syntax

  `if` (*Expression*)
  Action

- If the *Expression* is true then execute *Action*

- *Action* is either a single statement or a group of statements within braces



Sorting Two Numbers

```
cout << "Enter two integers: ";
int Value1;
int Value2;
cin >> Value1 >> Value2;
if (Value1 > Value2) {
   int RememberValue1 = Value1;
   Value1 = Value2;
   Value2 = RememberValue1;
}
cout << "The input in sorted order: "
 << Value1 << " " << Value2 << endl;
```

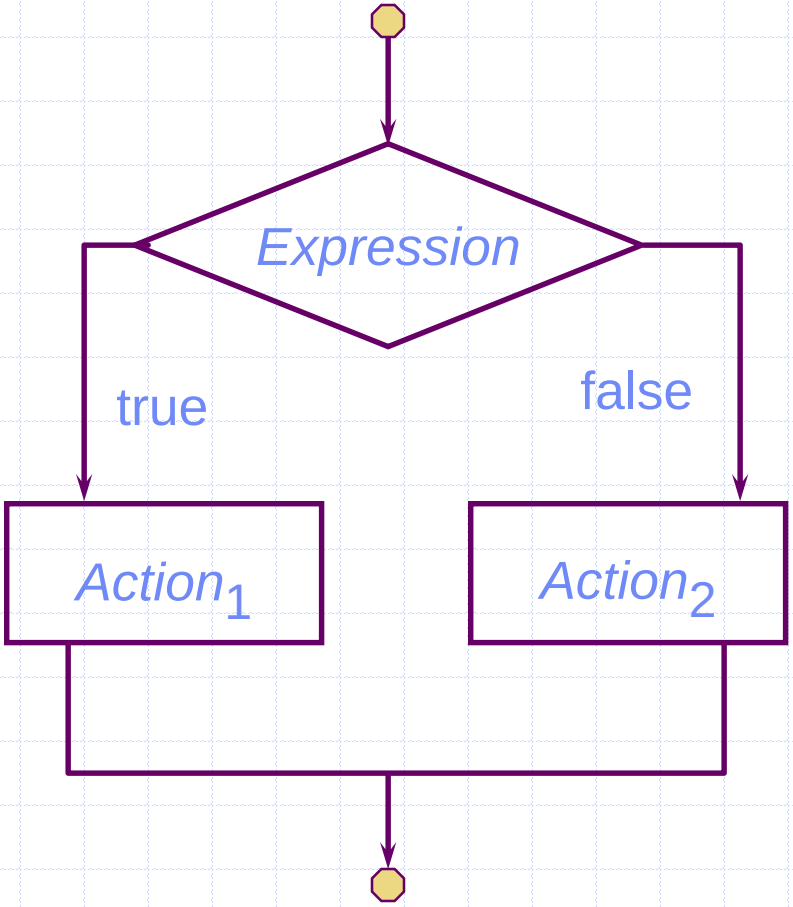
The If-Else Statement

- Syntax

```
if (Expression)
        Action1
    else
        Action2
```

- If *Expression* is true then execute $Action_1$ otherwise execute $Action_2$

```
if (v == 0) {
    cout << "v is 0";
  }
  else {
    cout << "v is not 0";
  }
```



Finding the Max

```
cout << "Enter two integers: ";
int Value1;
int Value2;
cin >> Value1 >> Value2;
int Max;
if (Value1 < Value2) {
    Max = Value2;
}
else {
    Max = Value1;
}
cout << "Maximum of inputs is: " << Max << endl;
```

Conditional Operator (?:)

- Conditional operator (`?:`) takes three arguments
  - Ternary operator

- Syntax for using the conditional operator:

```
expression1 ? expression2 : expression3
```

  - If `expression1` is `true`, the result of the conditional expression is `expression2`
  - Otherwise, the result is `expression3`

Multi-way Selection

- It is often the case that depending upon the value of an expression we want to perform a particular action
- Two major ways of accomplishing this choice
  - if-else-if statement
    - if-else statements “glued” together
  - Switch statement
    - An advanced construct

An If-Else-If Statement

```
if ( nbr < 0 ){
   cout << nbr << " is negative" << endl;
}
else if ( nbr > 0 ) {
   cout << nbr << " is positive" << endl;
}
else {
   cout << nbr << " is zero" << endl;
}
```

A Switch Statement

```
switch (ch) {
   case 'a': case 'A':
   case 'e': case 'E':
   case 'i': case 'I':
   case 'o': case 'O':
   case 'u': case 'U':
       cout << ch << " is a vowel" << endl;
       break;
   default:
       cout << ch << " is not a vowel" << endl;
}
```

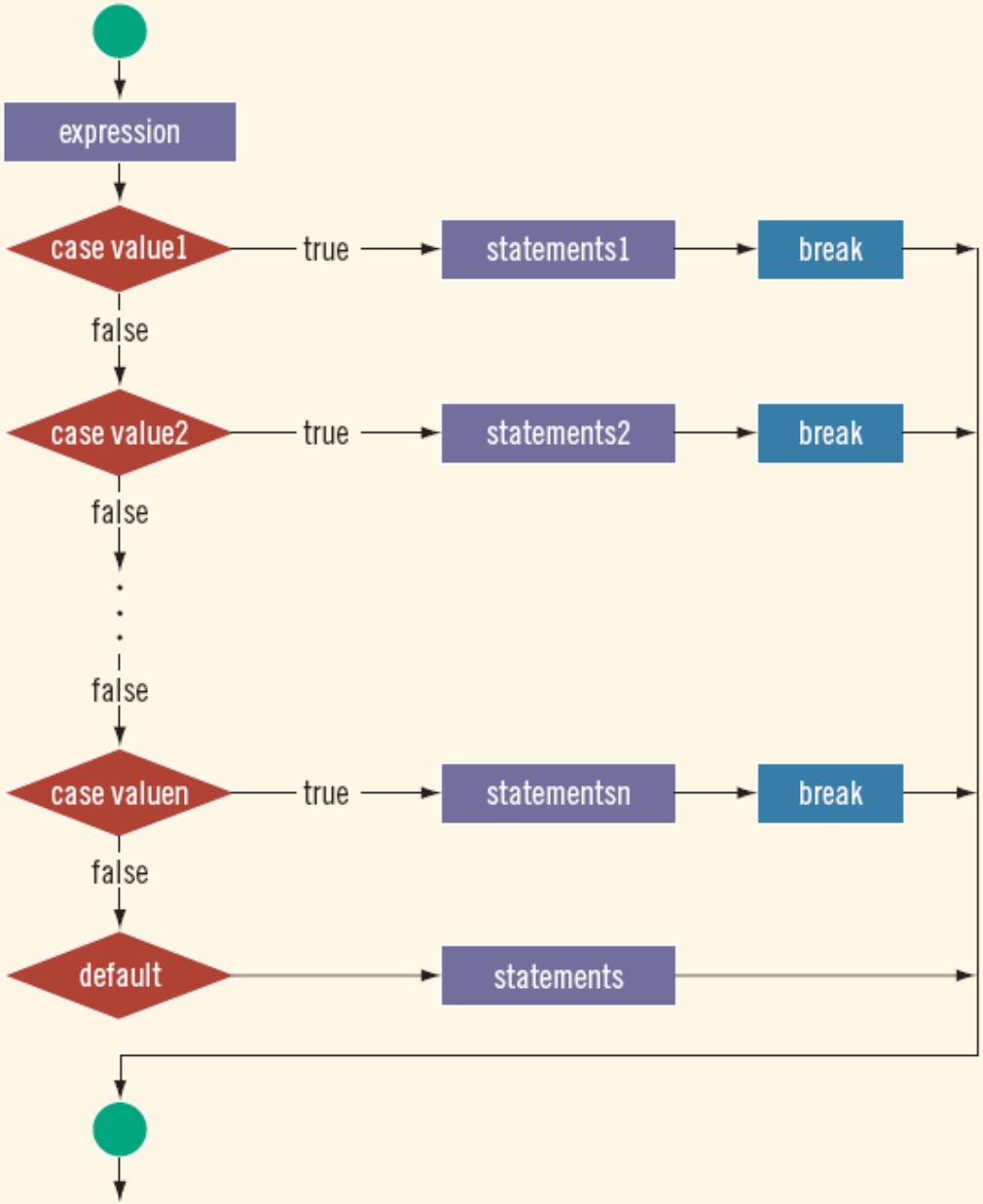


FIGURE 4-4 `switch` statement

`switch` Statement

- One or more statements may follow a case label
- Braces are not needed to turn multiple statements into a single compound statement
- The `break` statement may or may not appear after each statement
- `switch`, `case`, `break`, and `default` are reserved words

```
cout << "Enter simple expression: ";
int Left;
int Right;
char Operator;
cin >> Left >> Operator >> Right;
cout << Left << " " << Operator << " " << Right
 << " = ";
switch (Operator) {
   case '+' : cout << Left + Right << endl; break;
   case '-' : cout << Left - Right << endl; break;
   case '*' : cout << Left * Right << endl; break;
   case '/' : cout << Left / Right << endl; break;
   default: cout << "Illegal operation" << endl;
}
```

Terminating a Program with the `assert` Function

- Certain types of errors that are very difficult to catch can occur in a program
  - Example: division by zero can be difficult to catch using any of the programming techniques examined so far
- The predefined function, `assert`, is useful in stopping program execution when certain elusive errors occur

The `assert` Function (continued)

- Syntax:

```
assert(expression);
```

  `expression` is any logical expression
- If `expression` evaluates to `true`, the next statement executes
- If `expression` evaluates to `false`, the program terminates and indicates where in the program the error occurred
- To use `assert`, include `cassert` header file

The `assert` Function (continued)

- `assert` is useful for enforcing programming constraints during program development
- After developing and testing a program, remove or disable assert statements
- The preprocessor directive `#define NDEBUG` must be placed before the directive `#include <cassert>` to disable the assert statement

Example

```
#include <iostream>
// uncomment to disable assert()
// #define NDEBUG
#include <cassert>

int main() {
  assert(2+2==4);
  std::cout << "Execution continues  past the first assert\n";
  assert(2+2==5);
  std::cout << "Execution continues past the second assert\n";
}
```

Output:

Execution continues past the first assert
test: test.cc:10: int main(): Assertion `2+2==5' failed.
Aborted

Iterative Constructs

Mechanisms for deciding under what conditions an action should be repeated

C++ Iterative Constructs

- Three constructs
  - while statement
  - for statement
  - do-while statement

While Syntax

Logical expression that determines whether the action is to be executed

Action to be iteratively performed until logical expression is false

while (*Expression*) *Action*

Computing the Average

```
int listSize = 4;
int numberProcessed = 0;
double sum = 0;
while (numberProcessed < listSize) {
   double value;
   cin >> value;
   sum += value;
   ++numberProcessed;
}
double average = sum / numberProcessed ;
cout  << "Average: " << average << endl;
```

Suppose input contains: 1 5 3 1 6

Execution Trace

listSize	4

```
int listSize = 4;
int numberProcessed = 0;
double sum = 0;
while (numberProcessed < listSize) {
   double value;
   cin >> value;
   sum += value;
   ++numberProcessed;
}
double average = sum / numberProcessed ;
cout  << "Average: " << average << endl;
```

Suppose input contains: 1 5 3 1 6

Execution Trace

listSize	4
numberProcessed	4
sum	10
average	2.5

```
int listSize = 4;
int numberProcessed = 0;
double sum = 0;
while (numberProcessed < listSize) {
   double value;
   cin >> value;
   sum += value;
   ++numberProcessed;
}
double average = sum / numberProcessed ;
cout  << "Average: " << average << endl;
```

Suppose input contains: 1 5 3 1 6

Execution Trace

Stays in stream until extracted

```
int listSize = 4;
int numberProcessed = 0;
double sum = 0;
while (numberProcessed < listSize) {
   double value;
   cin >> value;
   sum += value;
   ++numberProcessed;
}
double average = sum / numberProcessed ;
cout  << "Average: " << average << endl;
```

Better Way of Averaging

```
int numberProcessed = 0;
double sum = 0;
double value;
while (  cin >> value  ) {
   sum += value;
   ++numberProcessed;
}
double average = sum / numberProcessed ;
cout  << "Average: " << average << endl;
```

The value of the input operation corresponds to true only if a successful extraction was made

What if list is empty?

Even Better Way of Averaging

```
int numberProcessed = 0;
double sum = 0;
double value;
while (  cin >> value  ) {
   sum += value;
   ++numberProcessed;
}
if ( numberProcessed > 0 ) {
   double average = sum / numberProcessed ;
   cout  << "Average: " << average << endl;
}
else {
   cout << "No list to average" << endl;
}
```

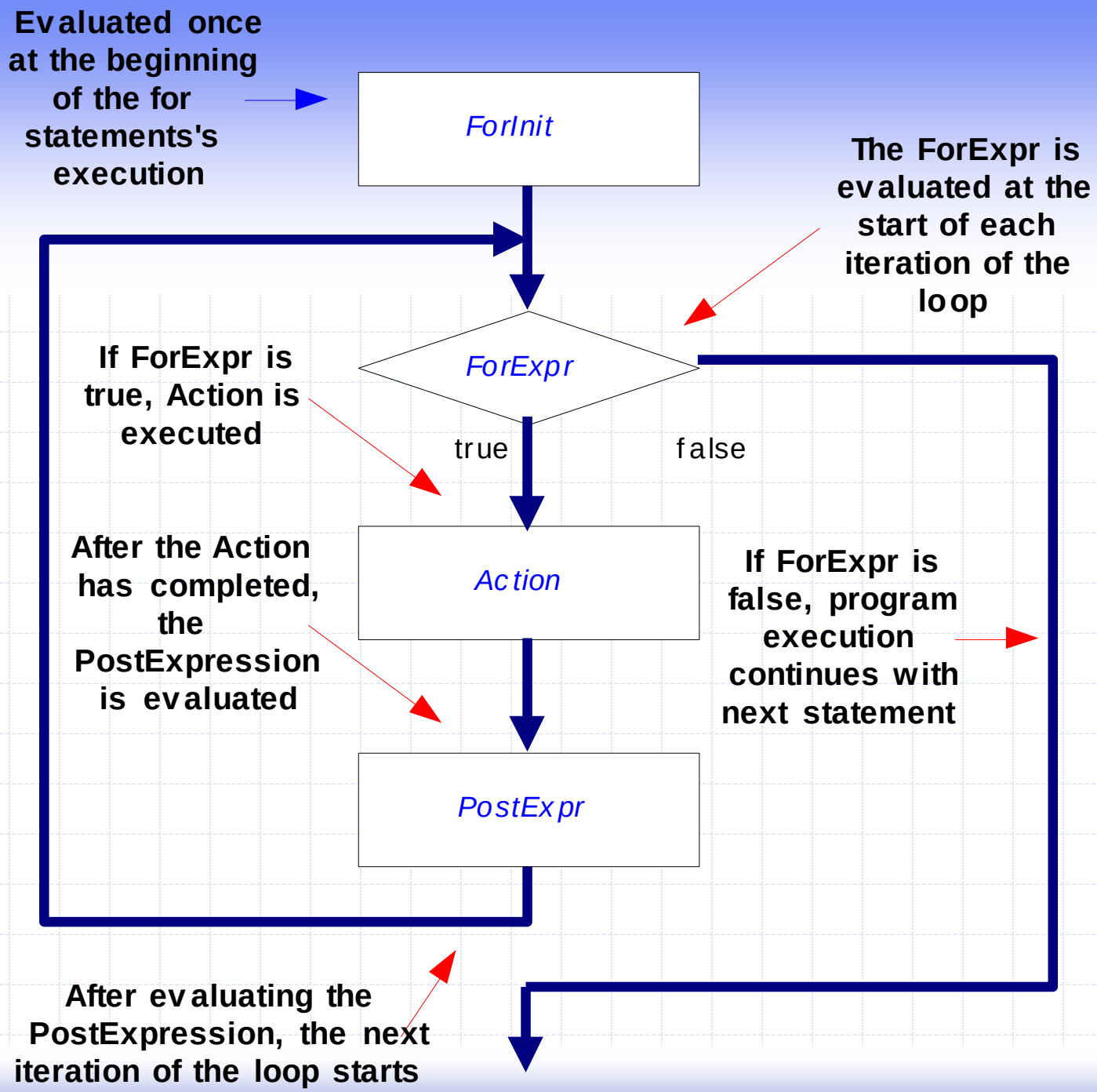
The For Statement

- Syntax

  for (*ForInit* ; *ForExpression*; *PostExpression*)
  Action

- Example

```
for (int i = 0; i < 3; ++i) {
    cout << "i is " << i << endl;
}
```


Evaluated once at the beginning of the for statements's execution
ForInit
The ForExpr is evaluated at the start of each iteration of the loop
ForExpr
If ForExpr is true, Action is executed
true
false
Action
After the Action has completed, the PostExpression is evaluated
If ForExpr is false, program execution continues with next statement
PostExpr
After evaluating the PostExpression, the next iteration of the loop starts

For Into While

- Observation
  - The for statement is equivalent to

```
{
  ForInit;
  while (ForExpression) {
        Action;
        PostExpression;
  }
}
```

Counting Characters

```
int NumberOfNonBlanks = 0;
int NumberOfUpperCase = 0;
char c;
while (cin >> c) {
   ++NumberOfNonBlanks;
   if ((c >= 'A') && (c <= 'Z')) {
       ++NumberOfUpperCase;
   }
}
cout << "Nonblank characters: " << NumberOfNonBlanks
     << endl << "Uppercase characters: "
     << NumberOfUpperCase << endl;
```

Only extracts nonblank characters

Counting All Characters

```
char c;
int NumberOfCharacters = 0;
int NumberOfLines = 0;
while ( cin.get(c) ) {
   ++NumberOfCharacters;
   if (c == '\n') {
       ++NumberOfLines
   }
}
cout << "Characters: " << NumberOfCharacters
     << endl << "Lines: " << NumberOfLines
     << endl;
```

Extracts all characters

File Processing

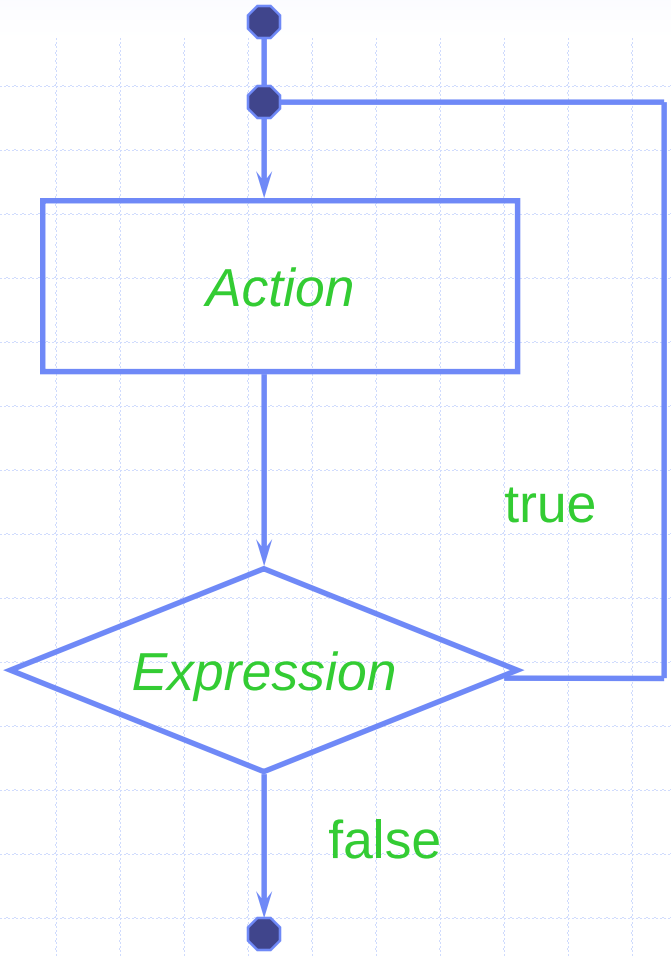
```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   ifstream fin("mydata.txt");
   int ValuesProcessed = 0;
   float ValueSum = 0;
   float Value;
   while ( fin >> Value ) {
        ValueSum += Value;
        ++ValuesProcessed;
   }
   if (ValuesProcessed > 0) {
        ofstream fout("average.txt");
        float Average = ValueSum / ValuesProcessed;
        fout << "Average: " << Average << endl;
        return 0;
   }
   else {
        cerr << "No list to average" << endl;
        return 1;
   }
}
```

The Do-While Statement

- Syntax

  `do` *Action*

  `while` (*Expression*)
- Semantics
  - Execute *Action*
  - If *Expression* is true then execute *Action* again
  - Repeat this process until *Expression* evaluates to false
- *Action* is either a single statement or a group of statements within braces



Waiting for a Proper Reply

```
char Reply;
do {
   cout << "Decision (y, n): ";
   if (cin >> Reply)
      Reply = tolower(Reply);
   else
      Reply = 'n';
} while ((Reply != 'y') && (Reply != 'n'));
```

Iteration Do's

- Key Points
  - Make sure there is a statement that will eventually terminate the iteration criterion
    - The loop must stop!
  - Make sure that initialization of loop counters or iterators is properly performed
  - Have a clear purpose for the loop
    - Document the purpose of the loop
    - Document how the body of the loop advances the purpose of the loop

`break` and `continue` Statements

- `break` and `continue` alter the flow of control
- `break` statement is used for two purposes:
  - To exit early from a loop
    - Can eliminate the use of certain (flag) variables
  - To skip the remainder of the `switch` structure
- After the `break` statement executes, the program continues with the first statement after the structure
- `continue` is used in `while`, `for`, and `do`…`while` structures
- When executed in a loop
  - It skips remaining statements and proceeds with the next iteration of the loop