3. Control Structures

Bohm, Jacopini (1966) showed that only the three following control structures are needed for representing an algorithm:

Sequence

2.1. Selection
Choosing one of several courses of action according to some condition

2.2. Repetition
Repetition of actions or sequences of actions according to some condition

Syntax Constructs:

built in	**if** **if/else** **? :** **switch**	**while** **for** **do/while**

3.1 Selection: **if** and **if/else** statements

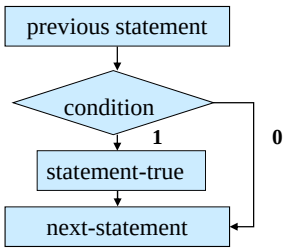
Example 3.1a:

```
float tax, salary;
tax = 0.0;
if (salary > 20000.0)
   tax=0.3*salary;
cout<<"Tax: " << tax << endl;
```

Syntax

```
<previous-statement>
if (<condition>)
   <statement-true>
<next-statement>
```

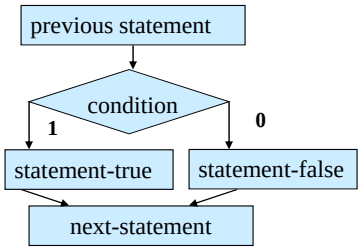
Flowchart:



Example 3.1b:

```
float tax, salary;
if (salary > 20000.0)
   tax=0.3*salary;
else
   tax = 0.0;
cout<<"Tax: " << tax << endl;
```

```
<previous-statement>
if (<condition>)
   <statement-true>
else
   <statement-false>
<next-statement>
```



Examples of **if** and **if/else** statements

Example 3.2: Finding the smallest **min** of three data objects **a**, **b**, **c**

```
// find the smaller of a and b, and assign its value to min
        if (a < b)
           min = a;
        else
           min = b;
// find the smaller of min and c, and assign its value to min
        if (min > c)
           min = c;
```

Example 3.3: Computing the absolute value **abs** of a data object **x**

a.
```
abs = a;
if (a < 0.)
   abs = -a; //a is negative
```

b.
```
if (a < 0.)
   abs = -a; //a is negative
else
   abs = a; //a is positive
```


Compound and Empty Statement

Compound statement

Problem: More than one statement needs to be executed in the "true" or "false" branch of the selection statement. As defined the syntax allows only one single <*statement-true*> or one single <*statement-false*>

enclose in braces **{…}**

Solution: Group somehow a sequence of statements and declarations and define the group as being one single statement, called a *compound* statement

{ <*declaration*>
<*statement*>
<*statement*> …
}

Example 3.3c

```
if (a < 0.) {
   abs = -a; //a is negative
   cout<<  "a is negative "<<endl;
   }
else {
   abs = a; //a is non negative
   cout<<  "a is non negative
"<<endl;
   }
```

Empty statement

Problem: In some cases no action is needed, but the syntax requires a statement

Solution: Define the single semicolon **;** as a statement, called the *empty* statement

Nested if/else

Example 3.4: Given the score a student has achieved in a class, print his/her letter grade according to the following grading scheme

score ≥ 90% : "A"

80% ≤ score < 90% : "B"

70% ≤ score < 80% : "C"

60% ≤ score < 70% : "D"

score < 60% : "F"

```
If score ≥ 90% then
   "The grade is A"
Else
   If score ≥ 80% then
      "The grade is B"
   Else
      If score ≥ 70% then
         "The grade is C"
      Else
         If score ≥ 60% then
            "The grade is D"
         Else
            "The grade is F"
```


Nested if/else: Indentation Style

If indentation follows the logic literally, the code runs across the page without having the benefit of conveying additional information:
AVOID!!!

```
if (score >=90 )
   cout << "The grade is A";
else
   if (score >= 80)
      cout << "The grade is B";
   else
      if (score >= 70 )
         cout << "The grade is C";
      else
         if (score >= 60)
            cout << "The grade is D";
         else
            cout << "The grade is F";
```

A better style: ADOPT!!!

```
if (score >= 90 )
   cout << "The grade is A";
else if (score >= 80)
   cout << "The grade is B";
else if (score >= 70)
   cout << "The grade is C";
else if (score >= 60 )
   cout << "The grade is D";
else cout << "The grade is F";
```

Note that each box represents a single statement

Dangling else

In some cases the interpretation of the nested if/else is ambiguous

Example 3.5: An insurance company uses the following table for car insurance

Age	Male	Female
Under 21	$800	$300
21 and older	$400	$250

The corresponding code

```
if (gender == 'M')
if (age < 21)
   rate = 800;
else
   rate = 400;
```

is ambiguous and can be interpreted as either:

Yes!

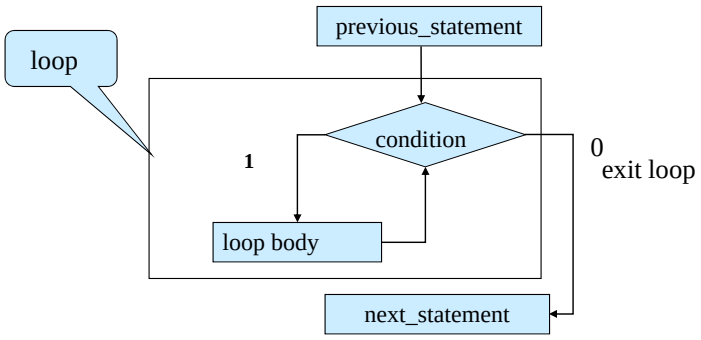
```
if (gender == 'M'){
if (age < 21)
   rate = 800;
else
   rate = 400;
}
```

```
if (gender == 'M'){
   if (age < 21)
      rate = 800;
}
else
   rate = 400;
```

Interpretation Rule: Every **else** is paired/matched with the ***closest preceding if***.
(Of course braces can change this as they have precedence.)

3.2. Repetition: **while, for, while/do**

General Idea



Repetition: Two Basic Repetition Types

Counter Controlled: The loop is controlled by a variable, called *counter*, that keeps track how many times the loop is executed. Repetition ends when the counter variable reaches a predetermined value, e.g. 10 numbers are added, largest of 200 numbers is found, etc.

Sentinel Controlled: The loop is controlled by a special value, placed at the end of the input data, which semantically cannot belong to the input data and referred to as *sentinel*, *signal*, *flag* or *dummy* value. A sentinel can be

(a) Defined by the programmer, e.g. negative number for $amounts, 99 for integers representing the months of the year, etc.; or

(b) Special symbol, e.g. end-of-file character (EOF), end-of-line (EOL) character, etc.

// Example 3.6: Counter controlled loop for **Adding the integers from 1 to 10**

```
// Listing 3-1-sum.cpp
# include <iostream>
using namespace std;
int main()
{
   int sum = 0;        // always initialize sum to 0
   int count =1;       // always initialize count (either to 0 or 1)

   cout << "This program computes the sum of" <<endl;
   cout << "the numbers from 1 to 10 \n" <<endl;

   while (count<=10){
      sum = sum + count;          // equivalently sum += count;
      count = count + 1;          // equivalently count++;
   }

   cout << "The sum is: " << sum << endl;

   return 0;
}
```

The loop is in the box. The loop body is in the braces.

```
This program computes the sum of
the numbers from 1 to 10

The sum is: 55
Press any key to continue
```

// Example 3.7: Counter controlled loopfor **Multiplying the odd integers from 1 to n**

```
// Listing 3-2-product.cpp

# include <iostream>
using namespace std;

int main()
{
    int product = 1;  // always initialize product to 1
    int count=1;      // always initialize count to either 0 or 1
    int n;            //upper bound of interval for factors

    cout << "This program computes the product of" <<endl;
    cout << "the odd numbers from 1 to n \n" <<endl;
    cout << "Enter a value for n:  ";
    cin >> n;

    int iteration = 0;
    cout<<"Iteration\n";
    while (count <=n){
        product *= count;       // product = product * count;
        cout << ++iteration << "\tMultiplying by "<< count;
        cout <<" yields current product of  " << product << endl;
        count += 2;                 // count = count + 2;
    }

    cout << "\nThe final product is: " << product << endl;
    return 0;
}
```

```
This program computes the product of
the odd numbers from 1 to n

Enter a value for n:  10
Iteration
1       Multiplying by 1 yields current product of  1
2       Multiplying by 3 yields current product of  3
3       Multiplying by 5 yields current product of  1
4       Multiplying by 7 yields current product of  1
5       Multiplying by 9 yields current product of  9

The final product is: 945
```

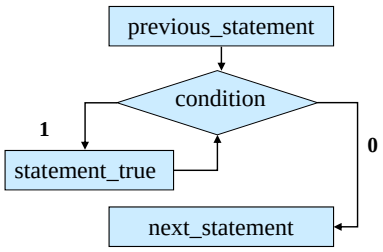

Repetition: while statement

<previous-statement>

while (*<condition>* **)**

<statement-true>

<next-statement>



---

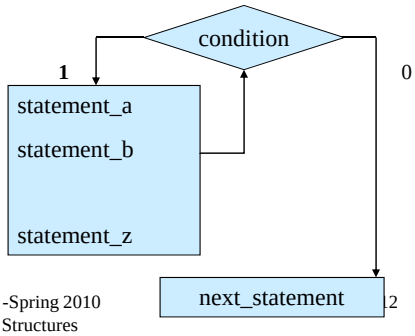
while (*<condition>* **) {**

<statement-a>
<statement -b>

<statement -z>

}

<next_statement>



Unary Operators

Unary Operator: an operator that is applied to ONE SINGLE argument.

Examples of unary operators are:

(i) The *sign operators* **+** and **-** that determine the sign of their argument; their precedence is higher than the precedence of the arithmetic operators * , / , %; and their associativity is right-to-left

i.e. a * -b + c is the same as a * (-b) + c

Note: The symbols for the unary sign operators are the same as for addition and subtraction. However, their precedence (higher) and their associativity (right-to-left instead of left-to-right) are different from the ones of the addition and subtraction operator. In fact, from a formal point of view they are different operators.

(ii) Logical *not* operator !

(iii) The *increment* ++ and *decrement* -- operators.

Increment and Decrement Operators: ++ and --

Prefix or Pre-increment / decrement
++a / --a

1. Value of variable is a increment by 1
2. Expression ++a/ --a uses the new INCREMENTED value

Postfix or Post-increment / decrement
a++ / a--

1. Expression a++/ a– uses the current NONincremented value of a
2. Value of variable a is incremented by 1

precedence is higher than * , / , %

with postincrement BEFORE preincrement

associativity is from right to left

(See Savitch Appendix 2: Precedence of Operators p.857)

Increment and Decrement: Examples

Example	values of		
declaration/statement	a	b	c
int a, b, c=0	-	-	0
a = ++c	1		1
b = c++		1	2

1. The value of the variable is incremented in BOTH cases. It is the expression that has different values
2. Associativity is always right-to-left, and both forms have higher precedence than the arithmetic *, /, %. There is a difference in precedence among the unary operators with
   postfix increment/decrement BEFORE unary !, +, -, as well as their prefix forms;
   and
   prefix increment/decrement having the same precedence as unary !, +, -.

Assignment Operators for Shorthand Notation

operator	example	equivalent C/C++ expression
+=	a += 3	a = a + 3
-=	a -= 7	a = a – 7
*=	a *= 5	a = a * 5
/=	a /= 7	a = a / 7

Generally: ✱= (Some binary operator)

A typical form of abbreviated assignment statement :

<identifier> ✱= *<expression>;*

Associativity: from right to left

Precedence: lowest

// Example 3.8 (Listing 3-3-average.cpp): Sentinel controlled loop for computing the Average number of employees per department in an company

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  //declaration and initialization
  int total=0,     // total number of employees
      number,    // number of employees entered ofr some department
      count=0;   // number of departments
  float average;   // average number of employees per department

  // sentinel is -1, as no negative number employees possible
  cout << "Enter number of employees in department , -1 to end: ";
  cin >> number;

  // Computing total number of employees and keeping count of departments
  while ( number != -1 ) {
    total += number;
    count++;
    cout << "Enter number of employees in department , -1 to end: ";
    cin >> number;
  }

  // Computing average
  if ( count != 0 ) { // to avoid division by 0
    average = static_cast< float >( total ) / count;
    cout << "Employee numbers for " << count << " departments were entered.\n"
         <<"The average number of employees per department is " << setprecision( 2 )
         << setiosflags( ios::fixed | ios::showpoint )
         << average << endl;
  }
  else
    cout << "No input data were given" << endl;

  return 0;
}
```

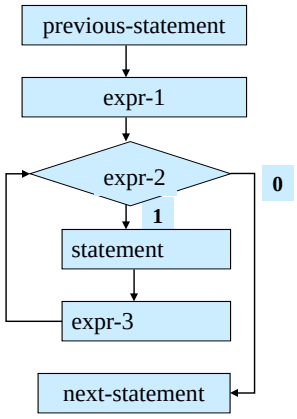
```
Enter number of employees in department , -1 to end: 17
Enter number of employees in department , -1 to end: 31
Enter number of employees in department , -1 to end: 27
Enter number of employees in department , -1 to end: 24
Enter number of employees in department , -1 to end: -1
Employee numbers for 4 departments were entered.
The average number of employees per department is 24.75
```

Repetition: for statement

Example 3.6 with for loop can be rewritten as:

```
for    (count = 1; count <= 10;   ++ count )
            sum  += count ;
```

Flowchart:



for Syntax:

<previous-statement>

for (*<expr-1>* **;** *<expr-2>* **;** *<expr-3>* **)**
    <statement>

<next-statement>

Equivalent **while** Syntax

```
<previous-statement>

<expr-1>;
while ( <expr-2> ){
        <statement>
        <expr-3>;
}

<next-statement>
```

for-loops: Examples and Notes

1. Typically the three expressions of the for statement are used as follows

   <expr-1> for **initialization** of the control variable;

   <expr-2> for stating the **end value** or **looping condition**;

   <expr-3> for defining the **stepwise increment**;

2. The increment can be negative, e.g. adding the integers 1 to 10 can be done in descending order

```
for (count = 10; count >= 1;  -- count )
        sum += count;
```

3. All three expressions in the for-statement are optional, they can be missing. However, the two semicolons must remain for syntactical reasons.

   a) if *<expr-1>* **is missing** initialization is not performed and must be done before the loop:

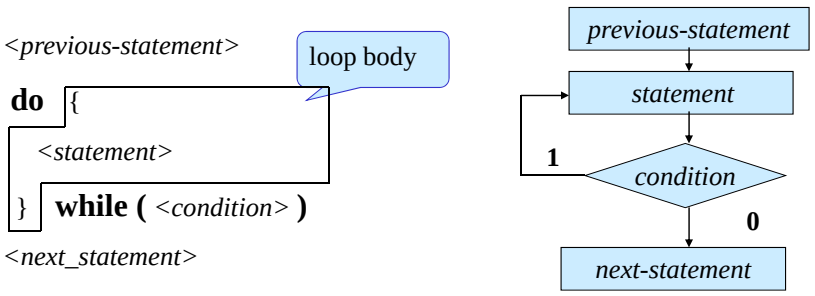
```
count = 1;
for ( ; count <= 10; ++ count)
        sum += count;
```

   b) if *<expr-3>* **is missing** incrementation is not performed and must be done within the loop body

```
count = 1;
for ( ; count <= 10;  )
        sum += count++;
```

   c) if *<expr-2>* **is missing**, the looping condition it is **always** true, and the result is in an infinite loop.

Another loop construct: **do/while**



Note:

- The braces { } of the loop body are not mandatory, but are good programming practice;
- do/while is equivalent to while,
- the difference is that in do/while loop body is executed the first time before condition checked in the

Software Design Issues

Top-Down Design:

The process of developing a program solution by starting at the coarsest level and working out the details gradually or by **stepwise refinement**.

Software Design Criteria:

correctness: program gives correct results for any set of valid data; typically achieved through extensive testing.

user-friendliness: easy to use; typically achieved through messages prompting for input, explaining errors or giving information about the workings of the program.

robustness: program is shielded from undesired events, e.g. wrong input type, division by zero, etc; typically achieved by checking the data at critical points, and, if necessary, preventing program continuation.

readability: easy to read by programmer; typically achieved through inclusion of appropriate comments.

Design **Example: Find the largest of n numbers (Version 1&2)**

Version 1:

```
read in number n of entries to be
    compared;
read in 1st number;
set largest to 1st number;
while (entry remains to be read in)
    read in number;
    compare number to largest,
    change if necessary;
output largest;
```

Version 2: making it more user-friendly by displaying program information and prompting for input

```
display program information;
prompt for number of entries to be
    compared;
read in number n of entries to be compared;
prompt for number entry;
read in 1st number;
set largest to 1st number;
while (entry remains to be read in)
    read in number;
    compare number to largest,
    change if  necessary;
output largest;
```

Design Example: Find the largest of n numbers (version 3 & 4)

Version 3: Making it robust by checking and not accepting a negative value as number entry, and asking to repeat the input until a positive value has been entered;

```
display program information;
prompt for number of entries to be compared;
read in number n of entries to be compared;
prompt for number entry;
while (input value <= 0)
  error message;
  prompt for input: how many numbers
                    will be entered;
  read in number n of entries to be
     compared;
read in 1st number;
set largest to 1st number;
while (entry remains to be read in)
    read in number
    compare number to largest,
    change if  necessary;
output largest;
```

Version 4: Work out remaining details

```
display program information;
prompt for number of entries to be compared;
read in number n of entries to be compared;
prompt for number entry;
while (input value <= 0)
  error message;
  prompt for input: how many numbers
                    will be entered;
  read in number n of entries to be compared;
read in number;
largest = number;
while  (++cnt < n)
    read in number;
    if (largest < number)
        largest = number;
output largest;
```


3.3. More on Selection and Repetition

- Mutliway decision: **switch**
  - The type **char, cin.get()**
- Unconditional jumps: **break**, **continue, goto**
- Shorthand for two-way decision: **?:** - the conditional operator

Multiway decision: **switch**

Gives a list of alternative actions, based on a single control expression

```
// Example 3.9: Converting score ranges to lettergrades
//Listing L3-4-switchGrades.cpp
#include <iostream>
using namespace std;

int main()
{
  int score;

  cout << "Enter the semester score, -1 to end" << endl;
  cin >> score;

  while ( score != -1 ) {
     Determines and Prints appropriate grade
       using the switch structure
     cin >> score;
  }

  return 0;
}
```


switch Example

control-expression: must have **integer** value

keyword

keyword followed by label= value of conrol-expression :

default label executed if no case label matched

```
switch (score/10) {    // switch nested in while
   case 10:    //perfect score 100
   case  9:    //scores >=90
      cout << "Lettergrade is A\n"; break;

   case  8:   // 90 > scores >= 80
      cout << "Lettergrade is B\n"; break;

   case  7:   // 80 > scores >= 70
      cout << "Lettergrade is C\n"; break;

   case  6:   // 70 > scores >= 60
       cout << "Lettergrade is D\n"; break;

   default:   // catch all other scores
       cout <<"Lettergrade is F\n";  break;
  }
```

```
Enter the semester score, -
1 to end
67
Lettergrade is D
89
Lettergrade is B
46
Lettergrade is F
98
Lettergrade is A
-1
```

switch Example - Flow Chart

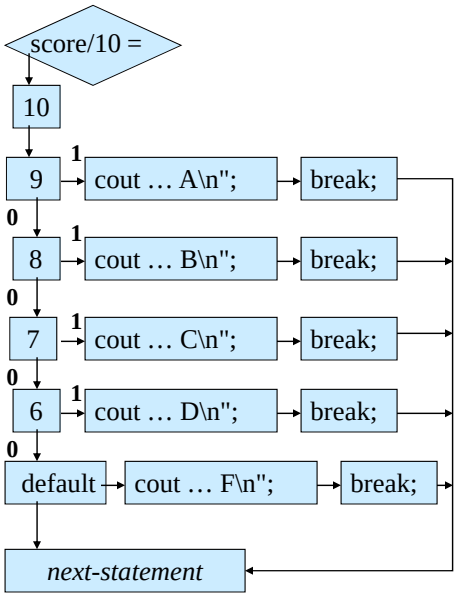
```
switch (score/10) {    // switch nested in while
   case 10:    //perfect score 100
   case  9:    //scores >=90
      cout << "Lettergrade is A\n"; break;

   case  8:   // 90 > scores >= 80
      cout << "Lettergrade is B\n"; break;

   case  7:   // 80 > scores >= 70
      cout << "Lettergrade is C\n"; break;

   case  6:   // 70 > scores >= 60
       cout << "Lettergrade is D\n"; break;

   default:   // catch all other scores
       cout <<"Lettergrade is F\n";  break;
    }
```



switch Syntax

Formal Syntax:

switch (<*control-expression*>)
{
  case <*label*> **:**
    <*statement*> <*statement*>…
  case <*label*> **:**
    <*statement*> <*statement*>…
  default:
    <*statement*> <*statement*>…
}
<*next_statement*>

Actions:

1. Evaluate *control-expression*;
2. **If** (value of *control-expression* matches one of the labels)
   go to **case** with matching *label;*
   execute statements between **':'** and next **case***;*
   if no break encountered fall through next case;
   Else, (i.e. there is no match)
   Go to **default** case;

3. Exit switch when **break** encountered, or by "falling through" the default.

Note: *control-expression* and all *labels* must have integer values

switch Examples: "Falling Through"

```
// Example 3.10: switch - Falling through
//Listing 3-5-switchFallThrough.cpp
#include <iostream>
using namespace std;

int main()
{
  char c;     // one letter
  cout << "Enter e digit from 1 to 9 : ";
  cin >> c;
  switch ( c ){
    case '9': cout << "9 9 9 9 9 9 9 9 9 \n";
    case '8': cout <<" 8 8 8 8 8 8 8 8 \n";
    case '7': cout <<" 7 7 7 7 7 7 7 \n";
    case '6': cout <<"   6 6 6 6 6 6 \n";
    case '5': cout <<"    5 5 5 5 5 \n";
    case '4': cout <<"     4 4 4 4 \n";
    case '3': cout <<"      3 3 3 \n";
    case '2': cout <<"       2 2 \n";
    case '1': cout <<"        1 \n";
    default: cout <<" - - - - - - - - - - - - - - - - \n";
  }
  return 0;
}
```

```
Enter a digit from 1 to 9 : 9
9 9 9 9 9 9 9 9 9
 8 8 8 8 8 8 8 8
 7 7 7 7 7 7 7
  6 6 6 6 6 6
   5 5 5 5 5
    4 4 4 4
     3 3 3
      2 2
       1
 - - - - - - - - - - - - - - -
```

```
Enter a digit from 1 to 9 : 4
     4 4 4 4
      3 3 3
       2 2
        1
 - - - - - - - - - - - - - - -
```


// Example 3.11a: Counting a-s and b-s in text; recognizing the EOF character

```
//Listing 3-6-charCount.cpp
//preprocessing directives
int main()
{
  char letter;     // one letter
  int  aCount = 0,  // number of a's
       bCount = 0,  // number of b's
       others = 0;  // number of other characters
  cout << "Enter text \n"
       << "Enter the EOF character to end input." << endl;
  while ( ( letter = cin.get() ) != EOF ) {

    switch structure for computing numbers
    of a-s, b-s and others

}
  cout << "\n\nTotals number of"
       << "\nA: " << aCount
       << "\nB: " << bCount
       << "\nother characters: " << others
       << "\nThe end-of-file character (EOF) is:  ";
  cout << static_cast<int>(letter) << endl; //check alternative cout.put(letter);

  return 0;
}
```

cin.get() reads in **char** that is stored in letter
Note: precedence of ' = ' requires (letter…);
letter is compare to EOF

Conversion needed to print letter as int

// Example 3.11a: switch structure (continued)

```
switch ( letter ) {      // switch nested in while

   case 'A':  // letter was uppercase A
   case 'a':  // or lowercase a
         ++aCount;
         break; // necessary to exit switch

   case 'B':  // letter was uppercase B
   case 'b':  // or lowercase b
         ++bCount;
         break;

   default:   // catch all other characters
         ++others;
         break;  // optional
    }
```

```
Enter text
Enter the EOF character to end input.
a
b
b
x
y
z
^Z

Totals number of
A: 1
B: 2
other characters: 9
The end-of-file character (EOF) is:  -1
```

```
Enter text
Enter the EOF character to end input.
abbxyz
^Z

Totals number of
A: 1
B: 2
other characters: 4
The end-of-file character (EOF)is:  -1
```

What are the other characters? Why is their number different in the two runs?

// Example 3.11b: Counting a-s and b-s in text; recognizing the EOF character
//Including a message that a character is not a or b

```
switch ( letter ) {      // switch nested in while
   case 'A':  // letter was uppercase A
   case 'a':  // or lowercase a
        ++aCount;
        break; // necessary to exit switch

   case 'B':  // letter was uppercase B
   case 'b':  // or lowercase b
        ++bCount;
        break;

   default:   // catch all other characters
        cout << "\nThe character you entered is not an a or b. "<< endl;
        ++others;
        break;  // optional
}
```

```
Enter text
Enter the EOF character to end input.
ab b

The character you entered is not an a or b.

The character you entered is not an a or b.
^Z

Totals number of
A: 1
B: 2
other characters: 2
The end-of-file character (EOF) is:  -1
```

strange claim as we see only a-s and b-s!!

Remedy: Remove white space

// Example 3.11c: Counting a-s and b-s in text; recognizing the EOF character
//Including a message that a character is not a or b, **ignoring white space**

```
int wCount = 0,    //counter for white space characters added

switch ( letter ) {  // switch nested in while
   //same as before
   case ' ':  // ignore space
   case '\t':  // tab
   case '\n':  // newline
        ++wCount;
        break;
   //same as before
}
```

still not perfect as not clear to what character it refers.

```
Enter text
Enter the EOF character to end input.
ab
dmn

The character you entered is not an a or b.

The character you entered is not an a or b.

The character you entered is not an a or b.
abdbb

The character you entered is not an a or b.
^Z


Totals number of
A: 2
B: 4
other characters: 4
The end-of-file character (EOF) is:  -1
```

```
Enter text
Enter the EOF character to end input.
ab b
ab
^Z


Totals number of
A: 2
B: 3
other characters: 0
The end-of-file character (EOF) is:  -1
```


switch Summary

- Provides an alternative to the nested if/else statement for multiple branching.
- Not as versatile as the nested if/else, but
- Gives explicit list of all cases thus increasing readability

Unconditional Jumps: **break** and **continue**

break in a loop or switch causes exiting the control structure; execution continues with next statement.

Example:

```
while (++number <= 6 ){
  if(number == 4)
    break;
  product *= number;
  cout << "Multiplied by "
       <<number << endl;
}
cout << "The product is: "
     << product << endl;
```

This program computes the product of integers from 1 to 6, **breaking loop at 4**
Multiplied by 2
Multiplied by 3
The product is: 6

continue in a loop causes skipping the current iteration; execution continues with evaluation of control expression.

Example:

```
while (++number <= 6 ){
    if(number == 4)
      continue;
    product *= number;
    cout << "Multiplied by "
         <<number << endl;
}
cout << "The product is: "
     << product << endl;
```

This program computes the product of integers from 1 to 6, **skipping 4**
Multiplied by 2
Multiplied by 3
Multiplied by 5
Multiplied by 6
The product is: 180

Loop-and-a-Half

The Loop-and-a-Half: loop termination condition is in the middle instead of at the beginning/end of the loop:

```
bool more = true;
while(more){//loop condition
  cin >> x;
  if( cin.fail() ) //decision point
    more = false;//
  else
    <false-statement>
}
```

Equivalent neater code:

```
while(true){//clear not a loop condition
  cin >> x;
  if( cin.fail() )// loop condition
    break; //decision point
  else
    <false-statement>
}
```

Note: break is a poor way to end a loop

EOF Detection with cin.eof(): safe only after input stream has failed

```
while(more){
  cin >> x;
  if( cin.eof() ) //DON'T
    more = false;
  else
    sum+=x;
}
```

If input fails for another reason EOF cannot be reached

input fails due to non-numeric input; ***EOF not reached***

5	\n	2	\n	t	y	p	0

input ***not failed***; ***EOF not reached***

5	\n	2	\n	EOF

```
while(cin){//DON'T
  sum+=x;
}
```

```
//DO: test for failure, then test for eof
bool more = true;
while(more){
  cin >> x;
  if (cin.fail() ) {
    more = false;
    if  (cin.eof())
      cout << "End of data";
    else
      cout << "Bad input data";
  }
}
```


Unconditional Jumps: the infamous **goto**

"Infamous" as it has been blamed for convoluted and unreadable programs. For this reason it is often outlawed altogether.

Syntax: **goto** label

label is an identifier put in front of the statement.

jumps allowed within the single function scope only;

no jumps allowed over definitions

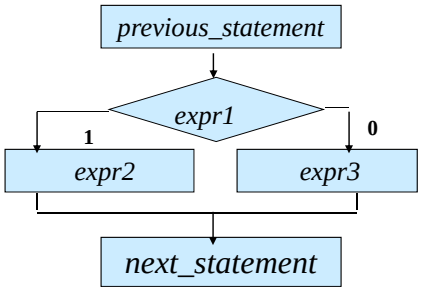
Shorthand for two-way decision: the conditional operator ?:

Syntax:

<*previous_statement*>

<*expr1*> **?** <*expr2*> **:** <*expr3*> ;

<*next_statement*>



Example 1: Largest of two numbers a, b

```
max = ( a > b ) ?  a : b ;
```

equivalent to

```
if ( a > b )
    max = a;
else
    max = b;
```

Example 2: Absolute value of a

```
abs = ( a > 0 ) ?  a:  -a ;
```


Conditional operator ?: Precedence and Associativity

Precedence	logical ||
	conditional ?:
	assignment = , +=, ….
Associativity	**right to left** ←

Note:

1. The value of the conditional expression
<*expr1*> **?** <*expr2*> **:** <*expr3*>
is equal to the value of the expression that is evaluated last , i.e. if <*expr1*> is true the entire conditional expression will have the value of <*expr2*>, and if <*expr1*> is false the conditional expression will take the value of <expr3>.

2. The type of the conditional expression is determined according to the usual conversion rules and does not depend on which of the expressions is evaluated.

Summary: Control Structures

- Selection: **if , if/else, switch** statements; **?:** operator

  Important Special Cases: nested if/else; indentation style, dangling else
- Repetition: **while**, **for, do..while**
- Unary Operators: sign (+, -), increment/decrement (++, --), static_cast
- Abbreviated Assignment Operators (+=, -=, *=, /=, %=, etc.)
- I/O Manipulators: **setprecision(), setiosflags()**
- Design Criteria:

  correctness, user-friendliness, robustness, readability