

Control Structure

Normally, a program is executed in a *sequential* manner. However, in many cases, a program has to choose among alternative statements. C++ provides constructs that enable the programmer to select which statement in the program to be executed next.

This is called *transfer of control*.

Flowcharts:

- A flowchart is a graphical representation of an algorithm.
- Flowcharts are drawn using special-purpose symbols, such as ovals, rectangles, diamonds and small circles.
- These symbols are connected by arrows called *flowlines*.

Example:

-Write a program that reads a number and prints out “even” if the number is even or “odd” if the number is odd.

Pseudocode:

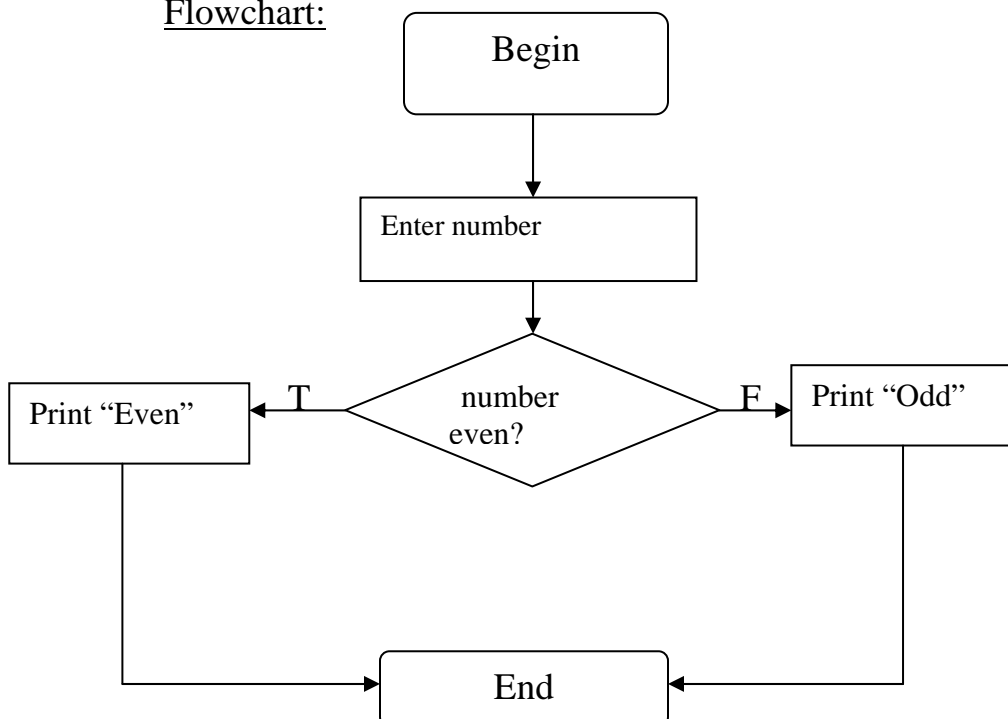
Input number

If number is even

Then Print “even”

Else print “odd”

Flowchart:



Problem:

Write the flowchart for the following problem:

Write a program that prompts the user to enter a number and writes:

- “A negative number has been entered” if the number is negative.
- “0 has been entered” if the number is 0
- “A positive number has been entered” if the number is positive.

Selection statements:

Conditions:

A selection in a program allows us to choose between two or more actions depending on whether a condition is true or false. This condition is based on a comparison of two items, and is usually expressed with one of the following relational operators:

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

In the if statement, a choice is made between two alternative paths, based on a decision about whether the “condition” is true or false.

For example, we want to write a program that determines the service charge for a bank account based on the account balance. When the account balance is less than \$1000, then there is a fee of \$5 on the balance.

Pseudocode:

```
if (account_balance < 1000)
```

```
    Fee = 5;
```

Such an expression is called a **condition** because it establishes a criterion for either executing or skipping a group of statements.

Most conditions used to perform comparisons have this form: variable relational-operator variable/constant.

Where relational operators are: <, <=, >, >=, ==, !=

The if statement:

In C++, the if statement is the primary selection control structure, It is used when we want to check the value of an expression before selecting a course of action.

There are basically two forms of if statements:

If statement with One alternative:

if (condition)

statement_T;

statement_N;

The **if** selection structure performs an indicated action (*statement_T*) only when the condition (*condition*) **is true**.

Otherwise the action (*statement_T*) is skipped.

In either case, *statement_N* is executed.

For example:

```
if (account_balance < 1000)
    service_charge = 5.00;
```

the condition `account_balance < 1000` compares the value of the variable `account_balance` to 1000.

```
/*Add sum to nonzero x */
if (x != 0.0)
    sum = sum + x;
```

The statement:

if (x != 0.0) compares the value of `x` to 0.0 and has one alternative, which is executed when `x` is not equal to 0.0.

If `x` is equal to 0.0, the addition is not performed.

The statement:

```
sum = sum + x;
```

causes the value of `x` to be added to the value of `sum` and the new value obtained to be saved in the variable `sum`.

Full program:

```
#include <iostream>
int main(void) {
    float x, sum;
    sum = 100.0;
```

```

cout << "Please enter a real number \n";
cin >> x;
if (x != 0.0)
sum = sum + x;
cout << "This is the sum " << sum;
return 0;
}

```

Program Output:

Run 1:

```

Please enter a real number
23.0
This is the sum 123.00000

```

Run 2:

```

Please enter a real number
0.0
This is the sum 100.00000

```

If Statement with two alternative OR if-else selection structure:

Form:

```

if (condition)
    statementT;
else
    statementF;

```

The **if/else** selection structure allows the programmer to specify the actions to be performed when the condition is **true** and the actions to be performed when the condition is **false**. When the **if** condition evaluates to **true**, the statement *statement_T* is executed. When the **if** condition evaluated to **false**, the statement *statement_F* that is the target of **else** will be executed.

Only the code associated with **if** condition **OR** the code associated with **else** executes, **never both**.

Example:

Suppose the passing grade on an exam is 60. Write a program that prints "Passed" if a grade entered is greater than 60 and "Failed" otherwise.

Algorithm:

Begin

Read student_grade

IF student_grade \geq 60 THEN

print "Passed"

ELSE

print "Failed"

Indentation is very important in making a program more readable and understanding the logic.

Full Program:

```
#include <iostream>
using namespace std;
int main(void) {
    int grade;
    cout << "Please enter the grade \n";
    cin>> grade;
    if (grade >= 60)
        cout<< "Passed";
    else
        cout<< "Failed";
    return 0;
}
```

if statements with compound or block statements:

In many cases, the target statement after the condition **if** or the keyword **else** is not a single statement, but a **group of statements**. In this case, we use braces **{ }** to delimit the block of statements for each case.

When the C++ compiler sees the symbol **{** after the condition or the keyword **else**, it executes (or skips) all statements through the matching **}**.

For example:

We want to write a program that determines the service charge for a bank account based on the account balance. When the account balance is less than \$1000, then there is a fee of \$5 on the balance and a

message is printed informing us about it. If the balance is above the amount, another message is sent.

Algorithm:

Begin

Read account balance

Set fee to 0

IF account balance <= 1000 THEN

print “Fee is due”

fee =5

ELSE

print “No fee is due”

End

account balance= account balance – fee

Full Program:

```
#include <iostream>
using namespace std;
int main(void) {
double account_balance;
int fee=0;
cout << "Please enter the account_balance: \n";
cin>> account_balance;
if (account_balance <=1000.00)
    {cout << "Fee is due";
    fee=5;}
else
cout << “No fee is due”;
account_balance= account_balance - fee;
cout << “The account balance is ”<< account_balance;
return 0;}
```

Nested ifs:

Nested **if/else** structures test for multiple cases by placing **if/else** structures inside **if/else** structures.

In C++, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and not already associated with an **if**.

```
if (expri) {  
    if (exprj)  
        statement1;  
    if (exprk)  
        statement2; /*this if is */  
    else  
        statement3; /*associated with this else*/  
} /*end if  
else  
    statement4;
```

The if-else-if Ladder

This is a common programming construct, also called the if-else-if staircase because of its appearance:

```
if (expri)  
    statement1;  
else  
    if (exprj)  
        statement2;  
    else  
        if (exprk)  
            statement3;  
        else  
            statement4;
```

The conditions are evaluated in a top down fashion. When a condition returns true, the rest of the ladder is bypassed. If none of the conditions are true, the final else statement is executed.

Example:

The following algorithm will print:

- A** for exam grades greater than or equal to 90.
- B** for grades greater than or equal 80.
- C** for grades greater than or equal to 70.
- D** for grades greater than or equal to 60 and
- F** for all other grades.

```

Begin
Read grade
if grade ≥ 90
    print “A”
else
    if grade ≥ 80
        print “B”
    else
        if grade ≥ 70
            print “C”
        else
            if grade ≥ 60
                print “D”
            else
                print “F”

```

The C++ code snippet:

```

if (grade >= 90)
    cout<< “A\n”;

else if (grade >= 80)
    cout << “B\n”;
else
    if (grade >= 70)
        cout << “C\n”;
    else
        if (grade >= 60)
            cout << “D\n”;
        else    cout << “F\n”;

```

The Switch Multiple-Selection Structure:

Sometimes an algorithm contains a series of decisions in which a variable or expression is tested separately. We used an if-else-if ladder to solve it.

The if-else-if Ladder

if (expression == value1) statement

else

 if (expression == value2) statement

 else

 if (expression == value3) statement

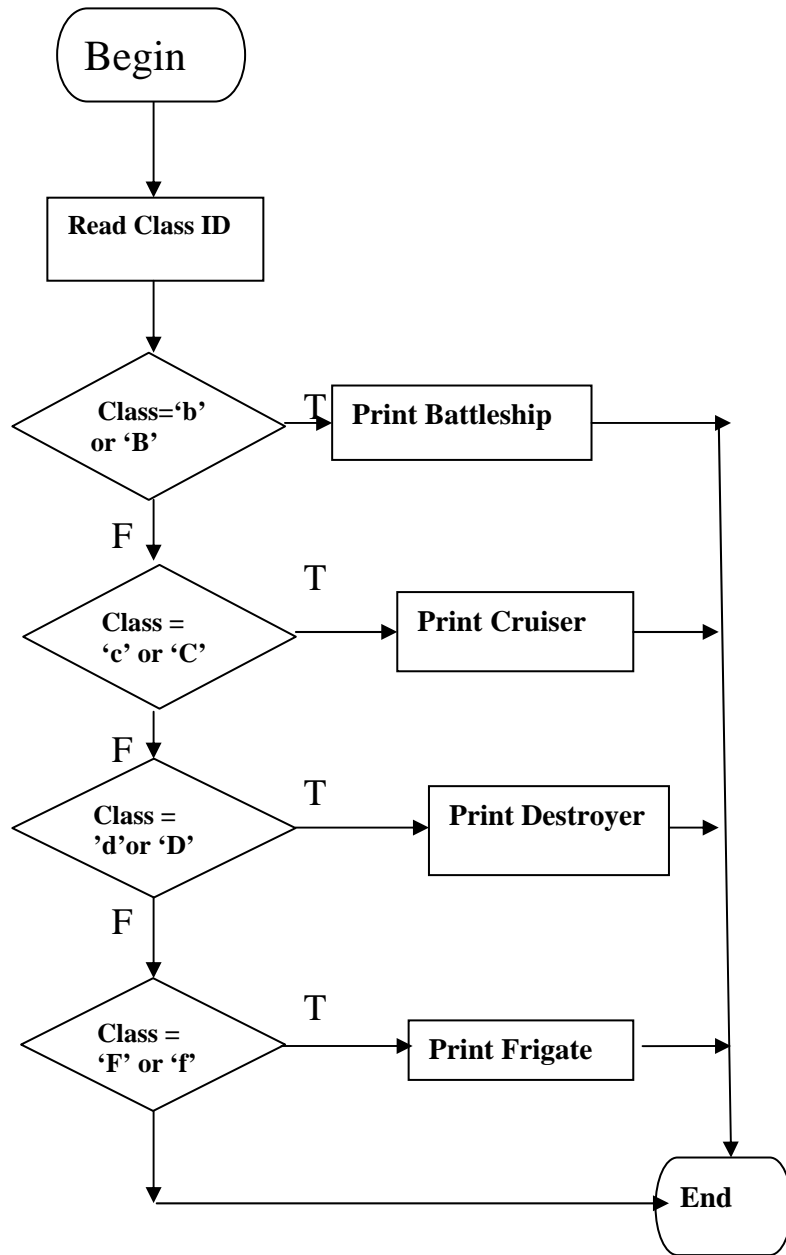
.....

C++ also provides the switch multiple selection structure to handle this kind of tasks.

The *switch case* statement is a better way of writing a program when a series of *if elses* occurs.

For example, we want to implement the following decision table, that classifies a ship depending on its class ID

Class ID	Ship Class
B or b	Battleship
C or c	Cruiser
D or d	Destroyer
F or f	Frigate

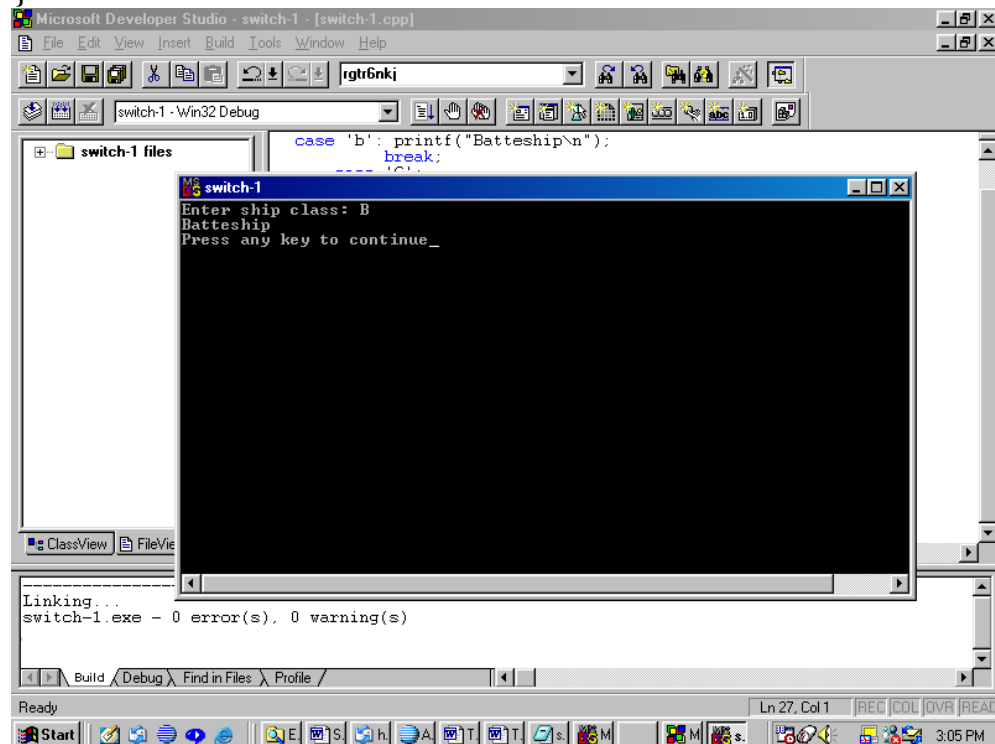


```

#include <iostream>
using namespace std;

int main(void) {
char Class_id;
cout<< "Enter ship class: ";
cin>> Class_id;
switch (Class_id) {
    case 'B': cout<<"Battleship\n"; break;
    case 'b': cout <<"Battleship\n"; break;
    case 'C':
    case 'c': cout<<"Cruiser\n"; break;
    case 'D':
    case 'd': cout<<" Destroyer \n"; break;
    case 'F':
    case 'f': cout<<"Fregate\n";
                break;
    default : cout<<"Unknown ship class<< Class_id<<endl;
            }
    return 0;
}

```



The general format for the switch statement is:

```
switch (variable) {  
    case  value1:  
        statement;  
        statement;  
        .....  
        break;  
    case  value2:  
        statement;  
        .....  
        break;  
    default:  
        .....  
        .....  
        break;  
}
```

The **switch** structure consists of a series of **case** labels and an optional **default** case.

The keyword **break** must be included at the end of each case statement.

The default clause is optional, and is executed if the cases are not met.

The right brace at the end signifies the end of the case selections.

Rules for switch statements

- values for 'case' must be integer,boolean or character constants.
- the order of the 'case' statements is unimportant
- the default clause may occur first (convention places it last)
- you cannot use expressions or ranges for the variable.

-The value of expression is tested, in order, against that value specified in the case.

-When a match is found the sequence associated with that case is executed until the break statement or the end of the switch statement is reached.

-The default statement is executed if no matches are found.

-If the default statement is not present, no action takes place.

```

#include <iostream>
using namespace std;
int main(void)
{
    int plusminus, num1, num2, output;
    cout<<"enter in two numbers ";
    cin >> num1 >> num2 ;
    cout<< "select an operator\n";
    cout<<"a=addition\n";
    cout <<"s=subtraction\n";
    cin >> plusminus ;
    switch(plusminus) {
        case 'a':
            output= num1 + num2;
            break;
        case 's':
            output = num1 - num2;
            break;
        default:
            cout<<"Invalid option selected\n";
    }

    if(plusminus == 'a' )
        cout << num1 << "plus"<< num2 <<"is"
        <<output;
    else
        if( plusminus == 's' )
            cout<< num1<< "minus"<< num2<<"is"
            <<output;
        return 0;
    }
}

```

Sample program output:

```

enter in two numbers  37 23
select an operator
a=addition
s=subtraction
s

```

37 minus 23 is 14

The **break** statement is one of C's jump statements.

When the **break** statement is encountered in a **switch** statement, program execution "jumps" to the line following the switch statement.

```
int main(void)
{
    int plusminus, num1, num2, output;
    cout<< "enter in two numbers ";
    cin>> num1>> num2 ;
    cout << "select an operator\n";
    cout << "a=addition\n";
    cout << "s=subtraction\n";
    cin>>plusminus ;
    switch(plusminus) {
        default: cout<<"Invalid option selected\n";
        case 'a': output= num1 + num2;
                    cout << num1<<"plus"<< num2<< "is" <<output;
                    break;
        case 's': output = num1 - num2;
                    cout<<num1<<" minus"<< num1<<"is"<< output;
    }

    return 0;
}
```

Output of sample program:

Enter two numbers 34 23

Select operator

a=addition

s=subtraction

1

34 plus 23 is 57

34 minus 23 is 11

Things to know about the switch statement:

- The **switch** statement differs from the **if** statement in that **switch** can only test for equality, whereas **if** can evaluate a relational expressions.
- No two case constants in the same **switch** can have identical values.
- The statements can be empty, in which case next case statement is executed.
- No braces are required for multiple statements in a **switch** structure.

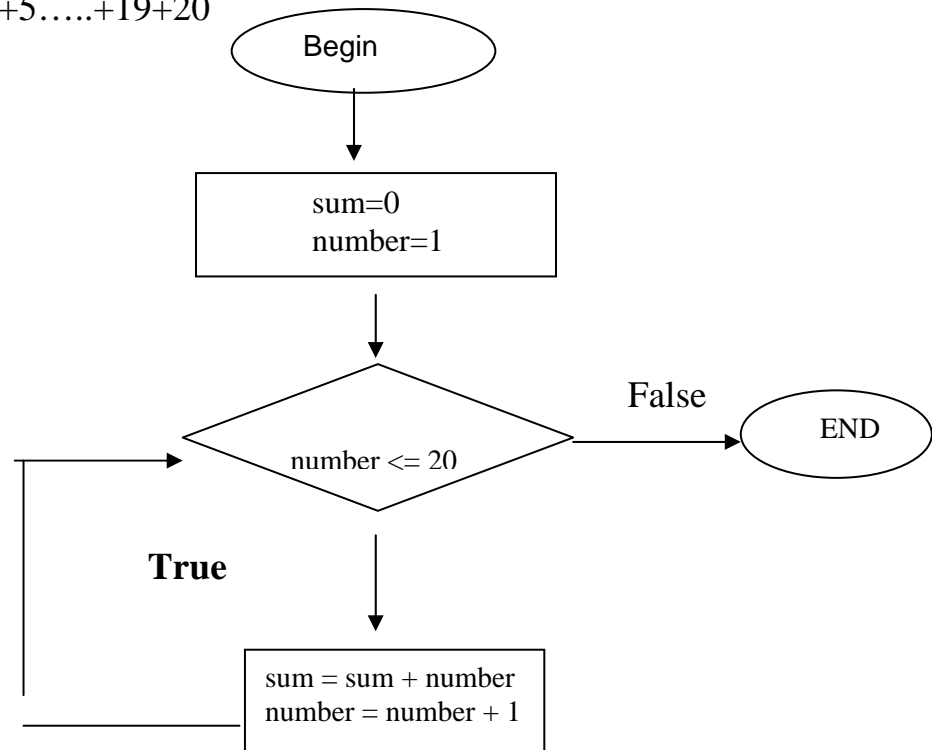
Tracing C++ statements:

A critical step in program design is verify that an algorithm or C statement is correct and that it is performing as it should. For

Example:

Add the 20 first numbers:

$1+2+3+4+5+...+19+20$



Repetition Structures:

A repetition structure also called iteration or loop allows an action or a set of instructions to be performed while some condition remains true.

There are 3 types of loop or repetition structures each one is suitable for a specific type of problems.

Counter-Controlled Repetition:

Problem Statement:

A class of 10 students takes a quiz. The grades (integers in the range 0 to 100) are entered.

Determine the class average.

Input:

-number of students:

-grade for each student

Output

-Class average.

Processing: How do we get the class average ?

We have to sum the grades for the 10 students and then divide the sum by the number of students.

So, we need a variable total that will be used to add up all the grades for the class.

Set total to 0,

1-Read grade_s1

2-Add grade_s1 to total

Read grade_s2

Add grade_s2 to total

Read grade_s3

Add grade_s3 to total

.....

Read grade_s10

Add grade_s10 to total

Set Average = total divided by 10

We notice that there is a repetition of the steps 1 and 2 with the exception that the name of the variable holding the grade changes.

A better approach is to use a **counter controlled loop**:

Set total to zero

Set grade_counter to one

WHILE (grade_counter ≤ 10) DO

 Input next grade

 Add the grade into the total

 Add one to the grade_counter

ENDWHILE

Set the class average to the total divided by ten.

Print the class average.

```
/*Program class average */
#include <iostream>
using namespace std;

int main(void) {
int counter, grade, total, average;
/* initialization phase */
total =0;
counter =1;
/*processing phase */
while (counter <= 10) {
    cout<<"Enter grade: " ;
    cin>> grade;
    total = total+ grade;
    counter = counter +1;
}/*end while */
/*termination phase */
average = total /10;
cout<<" Class average is" <<average<<endl;
return 0;

}/*end main */
```

The three color lines control the looping process.

The first statement:

counter =1; stores an initial value of 1 in the variable counter, which represent the count of students processed so far.

The next line:

while (counter <= 10) evaluates the condition (counter <= 10).

If the condition is true (1), the block statement representing the loop body is executed, this cause a new grade to be read and its value to be added to the varaible total.

The last statement in the loop body:

counter = counter +1;

adds 1 to the current value of counter.

After executing the last step in the loop body, control returns to the line beginning with **while** and the condition is re-evaluated.

The loop body is executed for each value of **counter** from 1 to 10.

When the value of **counter** becomes 11, and the condition evaluates to false (0).

At this point, control passes to the statement that follows the loop body.

General form of a while loop:

while (*condition*)

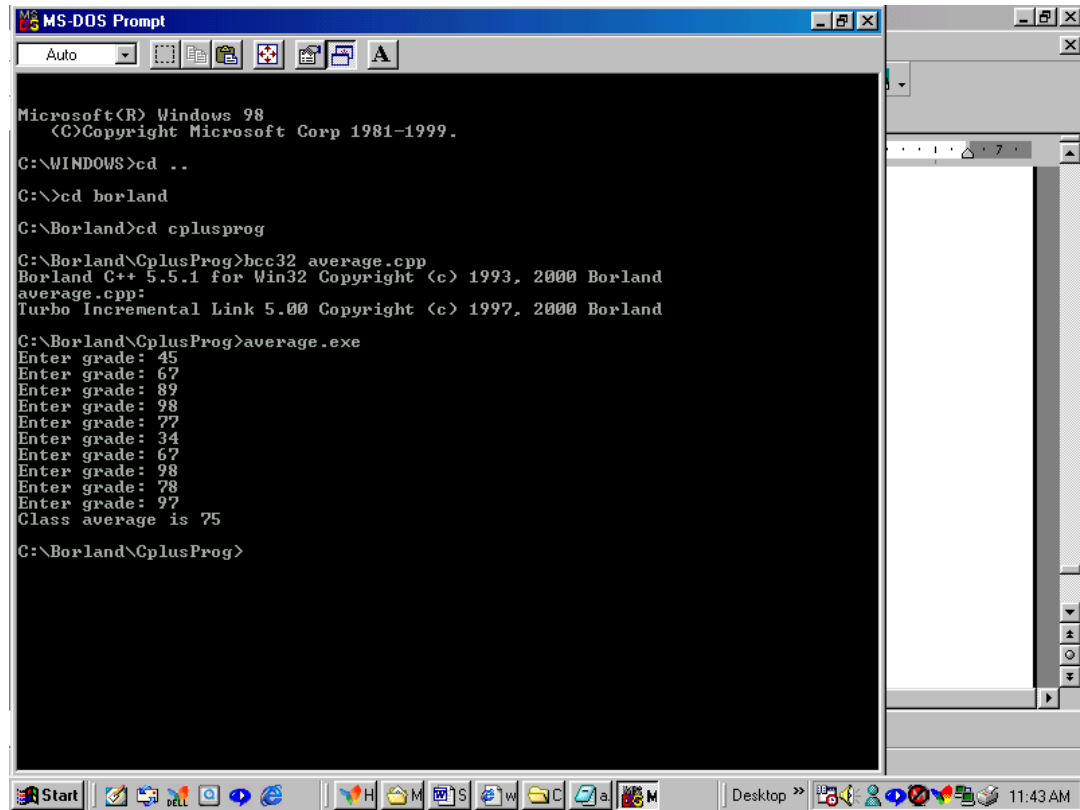
statement;

where:

- The **condition** may be any expression and **true is any nonzero value**.
- *statement* is a single statement or a block of statements that repeats { }.
- The loop iterates while the condition is true.
- When the condition becomes false, program control is returned to the line after the loop code.
- Somewhere within the body of the *while* loop a statement must alter the value of the condition to allow the loop to finish.

The three parts of a program are illustrated:

- *Declaration and initialization part.*
- *Processing part.*
- *Termination and result output phase.*



```
Microsoft(R) Windows 98
(C)Copyright Microsoft Corp 1981-1999.

C:\WINDOWS>cd ..
C:\>cd borland
C:\Borland>cd cplusplus
C:\Borland\Cplusplus>bcc32 average.cpp
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
average.cpp:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

C:\Borland\Cplusplus>average.exe
Enter grade: 45
Enter grade: 67
Enter grade: 89
Enter grade: 98
Enter grade: 77
Enter grade: 34
Enter grade: 67
Enter grade: 98
Enter grade: 78
Enter grade: 97
Class average is 75

C:\Borland\Cplusplus>
```

- Initialize the counter prior to entering the while loop.
- Alter the counter value inside the loop so that the loop finishes, otherwise you'll get an *infinite loop bug*.

Sentinel Controlled Repetition:

Problem:

Let's generalize the class average problem in such a way that the number of grades is *not* known in advance.

Problem formulation:

Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

One way to solve this problem is to use a special value called *sentinel value*(*flag value* or *exit value*) that will indicate the “end of data entry.”

This is how it works:

- The user enters all the grades she/he wants computed.
- The user then enters the sentinel value to indicate to the program that there is no more data to be entered.

- Sentinel controlled loops is often called indefinite repetition because the number of repetitions is not known in advance.
- The value of the sentinel or flag must be chosen in such a way that it cannot be confused with a valid value.

How do we modify this Pseudocode to work with our new constraints:

Pseudocode:

Set total to zero

Set student counter to zero

Read grade value

While (grade value is different from sentinel value){

 Add the grade into the total

 Add one to the student counter

 Read grade value }

(End while)

Set the class average to the total divided by the student counter.

Print the class average.

`*Class Average program with sentinel-controlled repetition *\`

`int main(void)`

`{/*initialization phase */`

`float average;`

`int counter, grade, total;`

`counter= 0, total=0;`

`cout<<" Enter grade, -1 to end: ";`

`cin>> grade;`

`while (grade != -1) {`

`total = total+ grade;`

`counter = counter+ 1;`

`cout<< "Enter grade, -1 to end: ";`

`cin>> grade;`

`} /* end while */`

`/*termination phase */`

`if (counter != 0) {`

`average =(float) total /counter;`

`cout<< "Class average is "<< average;`

`}/* end if */`

```

else
    cout<< " No grades were entered \n";
return 0;
} \*end main *

```

The while loop has four statements and they must be enclosed between braces.

```

total = total+ grade;
counter = counter+ 1;
cout<< "Enter grade, -1 to end: ";
cin>> grade ;

```

The line:

```

average =(float) total /counter;

```

uses an operator, we call the *cast operator*

The Cast Operator:

- Casting allows you to convert a value from one data type to another.
- The Cast operator is unary and has the form:

$$(datatype) \text{ value}$$
- The precedence of the cast operator is higher than that of arithmetic operators, except the ().
- When we cast a variable, the cast operator creates a copy of its value with the required type, the original variable ***does not change***.
- If the operands of an expression are of different data types the compiler performs an operation called ***promotion or implicit conversion***.

In the case above, the variable `total` is being “cast” into a float or real value, so that the result of the division is a float rather than an integer.

The operation is executed in this order:

1. The value of `total` is copied into a float representation
2. The division is performed, the result is a float
3. The result is assigned to the variable `average`, which is of type float.

The line: `cout<< "Class average is" << average;`
prints the variable `average`

Case Study 3: Nested Control Structures:

Problem:

You have been given a list of 10 student names with their exam results.

Next to each name is written 1 if the student passed or 2 if the student fails.

Your program should analyze the results of the exam as follow:

- Display the message "Enter result" on the screen each time the program requests a test result.
- Input each test result (i.e. a 1 or 2).
- Summarizes the results and print the number of passes and the number of failures.
- Recommends that tuition be raised if more than 8 students have passed.

Input:

-10 test results

Output:

- Number of passes and failures.
- Recommendation on tuition.

Process:

To process the 10 test results.

A counter-controlled loop has to be used.

Each time the program reads a test result

1. the program must determine if it is a 1 or a 2.
2. Two counters are used, each to use the number of passes and failures.
3. After the program has processed all the results, it must decide if more than 8 students passed the exam.

Pseudocode:

Initialize passes to zero,

Initialize failures to zero

Initialize student counter to 1

While student counter <= 10

Input the next exam result

If student passed

Add one to passes

```

        Else
            Add one to failures
            Add one to student counter
    End While
    Print the number of passes
    Print the number of failures
    If more than eight students passed
        Print "Raise tuition"

```

```

\* Analysis of examination results *\
#include <iostream>
using namespace std;
int main(void) {
    \* initialization variables *\
    int passes =0, failures = 0, student = 1, result;
    while (student <= 10) {
        cout<< "Enter result (1=pass, 2 =fail):\n";
        cin>>result;
        if (result == 1)
            passes = passes + 1;
        else
            failures = failures + 1;
        student = student + 1;
    }/* end while */
    cout<< "Passed"<< passes;
    cout<< " Failed" << failures;

    if (passes > 8)
        cout <<"Raise tuition\n";
    return 0;
}/* end main */

```

Efficiency pointers:

```

\* Analysis of examination results *\
#include <iostream>
main() {
    \* initialization variables *\
    int passes =0, failures = 0, student = 1, result;
    while (student <= 10) {
        cout << "Enter result (1=pass, 2 =fail):\n";

```

```

cin>> result;

if (result == 1)
    passes = passes + 1;
else
    failures = failures + 1;
    student = student + 1;
} \* end while * \
cout<< "Passed"<< passes <<endl;
cout<<"Failed " <<<failures <<endl;
if (passes > 8)
    cout <<"Raise tuition\n";
return 0; } \* end main * \

```

-Declaring and initializing variables in the same statement.

datatype variable = value;

int var=0;

int var;

var=0;

-Abbreviating assignment expressions:

For example:

passes = passes + 1

Can be re-written as:

passes += 1

-The += **operator** adds the value of the expression on the right of the operator += to the value of the variable on the left of the operator and stores the result in that variable.

Format:

variable operator= expression

where operator is one of +, -, *, /, %

So we have that :

variable= variable operator expression

Can be re-written as:

variable operator= expression

total +=1 (adds 1 to total), total *=2 (multiplies total by 2)

int c = 3, d =5, e = 4, f = 6, g = 12;

<i>Assignment Operator</i>	<i>Sample Expression</i>	<i>Explanation</i>	<i>Assigns</i>
<code>+=</code>	<code>c+=7</code>	<code>c=c+7</code>	<i>10 to c</i>
<code>-=</code>	<code>d-=4</code>	<code>d= d-4</code>	<i>1 to d</i>
<code>*=</code>	<code>e*=5</code>	<code>e= e*5</code>	<i>20 to e</i>
<code>/=</code>	<code>f/=3</code>	<code>f= f/ 3</code>	<i>2 to f</i>
<code>%=</code>	<code>g%=9</code>	<code>g=g%9</code>	<i>3 to g</i>

Signed and unsigned numbers:

C++ provides the unary + and – so we can define numbers such as:

`a = -2, b = +3, etc...`

Increment and decrement operators:

C++ also provides the unary increment operator ++ and the unary decrement operator --.

If a variable is incremented by *1* the increment operator ++ can be used instead of the expressions:

variable = variable + 1, OR variable +=1
we can use variable ++ or ++variables

-If the increment or decrement operators are placed before a variable, they are referred to as

preincrement or predecrement operators. ++var, --var

-If the increment or decrement operators are placed after a variable, they are referred to as *postincrement or postdecrement operators. var++, var--*

If a preincrement operator is used in an expression,

For example: `b= ++a,`

Then the operations are executed in the following order:

The value of a is incremented by 1, and the new value is assigned to b.

If a postincrement operator is used in an expression,
For example: $b = a++$,

Then the operations are executed in the following
order: The value of a is assigned to b , then a is incremented
by 1.

$a = 2; b = ++a$

Value of $a = 3$

Value of $b = 3$

$a = 2; c = a++$

Value of $a = 3$

Value of $c = 2$

-Preincrement and postincrement forms have the
same effect, when incrementing or decrementing a variable in a
statement by itself.

$a++$ or $++a$ are equivalent

The $++$ and $--$ operators can only be used with simple variables, so
 $++(x+1)$ is an error.

The Essentials of Repetition

A loop is a group of instruction the computer executes repeatedly
while some condition remains true,

We have seen two types of repetition:

1. Counter-controlled repetition:
 - a. The number of iterations or how many times the loop is going to be executed are known.
 - b. A control variable is used to count the number of iterations and is incremented (usually by 1).
 - c. The loop exits when the control variable value has reached the final count.
2. Sentinel-controlled repetition:
 - a. The precise number of iterations is not known in advance.
 - b. The sentinel value indicates “end of data”
 - c. Sentinel value must be distinct from regular, valid data.

Counter-controlled Repetition:

1. The name of a control variable (or loop counter).
2. The initial value of the control variable
3. The increment (or decrement) by which the control value is modified each time through the loop.
4. The condition that tests for the final value of the control variable, or exit condition.

```
/* example of counter controlled repetition */
#include <iostream>
using namespace std;

int main(void)
{
    int counter = 1;
    while (counter <= 10) {
        cout << counter;
        ++counter;
    }
    cout << endl;
    return 0;
}
```

The for repetition Structure:

The for structure handles all the details of the counter-controlled repetition automatically. The previous program is re-written as:

```
/* counter-controlled loop with for structure */
#include <iostream>
using namespace std;
int main(void)
{
    int counter;
    for (counter = 1; counter <= 10; counter++)
        cout << counter;

    cout << "\n";
    return 0;

}/*end main */
```

Sample Program Output

1 2 3 4 5 6 7 8 9 10

An integer variable *counter* is declared.

The first part of the *for* statement

```
for( counter = 1;
```

initializes the value of *counter* to 1.

The *for* loop continues for as long as the condition

```
counter <= 10;
```

evaluates as TRUE (1).

As the variable *counter* has just been initialized to 1, this condition is TRUE and so the program statement

```
cout<< counter ;
```

is executed, which prints the value of *counter* to the screen, followed by a space character.

Next, the remaining statement of the *for* is executed

```
++counter ) ;
```

which adds one to the current value of *counter*.

Control now passes back to the conditional test,

```
counter <= 10;
```

which evaluates as true (counter =2), so the program statement :

```
cout<< counter ; is executed.
```

counter is incremented again, the condition re-evaluated etc, until *counter* reaches a value of 11. When this occurs, the conditional test

counter* <= 10;** evaluates as FALSE (0), and the *for* loop terminates, and program control passes to the statement ***cout<<endl; which prints a newline,

The program then terminates, as there are no more statements left to execute.

General form of for loop structure is:

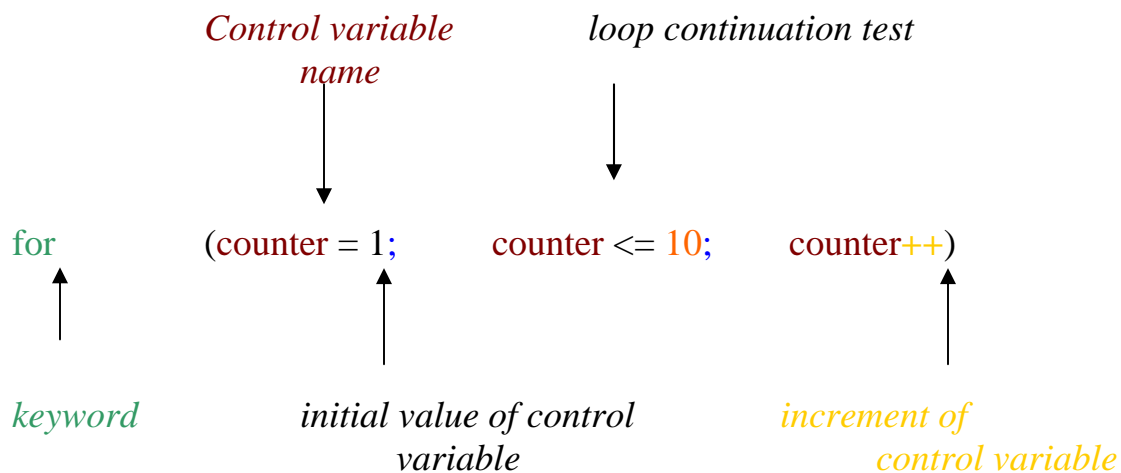
for (*expression1*; *expression2*; *expression3*)

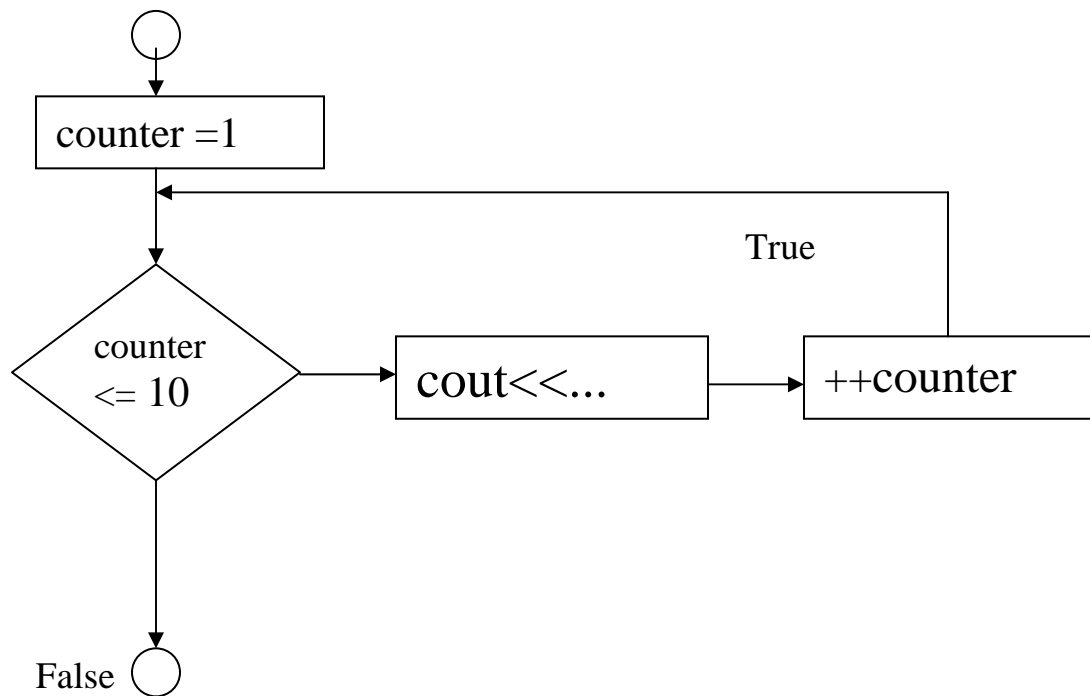
where:

1. *expression1* initializes the loop's control variable,
2. *expression2* is the loop continuation condition,
3. and *expression3* increments the control variable.

This is equivalent to:

```
expression1;  
while(expression2) {  
    statement  
    expression3;  
}
```





Examples Using for Structure:

1. Vary the control variable from 1 to 100 in increments of 1:
for (i=1; i<= 100; i++)
2. Vary the control variable from 100 to 1 in increments of -1:
for (i=100; i>=1; i--)
3. Vary the control variable from 20 to 2 in increments of -2:
for (i=20; i >= 2; i -=2)
4. Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55,0

Write a program that will sum all the even numbers from 2 to 100 using a **for loop**

Input: Numbers from 2 to 100.

Output: Sum of even numbers

```
#include <iostream>
using namespace std;

int main(void) {
    int sum = 0, number;
    for (number = 2; number <= 100; number += 2)
        sum += number;
    cout << "The sum of even numbers from 2 to 100 is " << sum << endl;
    return 0;
}
```

Do-While loop structure:

This structure is similar to the **while** loop, except for one thing:

- In the **while** loop, the continuation condition is tested at *the beginning* of the loop.
- In the **do-while**, the continuation condition is tested after the loop body is performed.
- When a **do-while** terminates, execution continues with the statement after the while clause.

General Structure of do-while:

```
do
    statement;

while (condition);
```

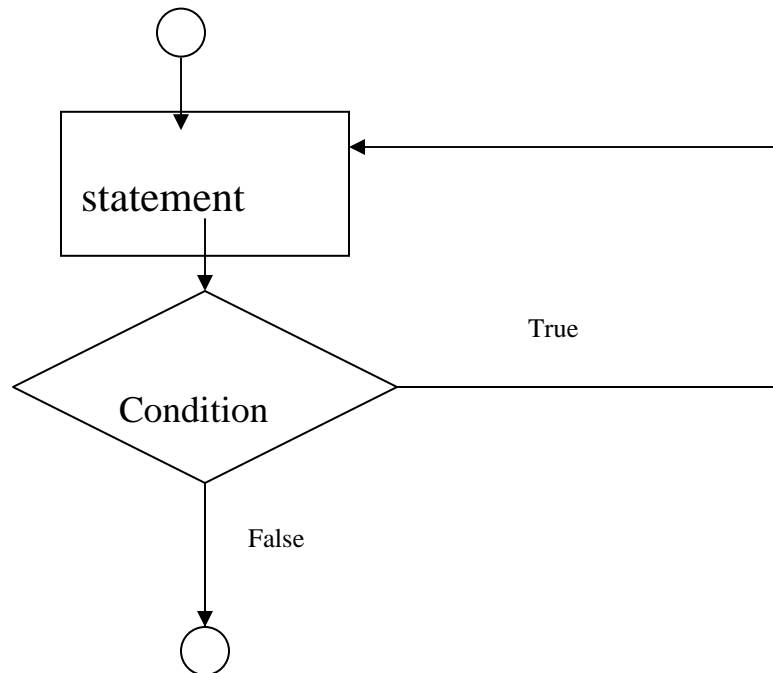
If there is only one statement, braces are not required.

However, for clarity, it is better to put them.

So, it is usually written as:

```
do {
    statement;
} while (condition);
```

do-while loop flowchart:



The break and continue Statements:

break and **continue** are used to alter the flow of control.

The break statement:

When used in a **for**, **while**, **do-while** or **switch** structures

- causes immediate exit from the structure,
- program execution continues with the first statement after the structure.
- commonly used to escape early from a loop, or skip the remainder of a **switch** structure.

```
#include <iostream>
using namespace std;
int main(void) {
    int x;
    for (x=1; x<=10; x++) {
        if (x==5)
            break;
        cout<< x;
    }//end for
```



```
cout << "\nBroke out of loop at " << x << endl;
return 0;
} //end main
```

Program Output:

```
1 2 3 4
Broke out of loop at x == 5
```

The continue statement:

- The **continue** statement when executed in a **while**, **for** or **do-while** structure, skips the remaining statements in the body of the loop and performs an iteration of the loop.
- In **while** and **do-while** loop, the loop continuation test is evaluated *after* the **continue** is executed.
- In the **for** structure, the increment expression is executed, then the continuation test is evaluated.

```
#include<stdio.h>
int main(void) {
int x;
for (x=1; x<=10; x++) {
    if (x==5)
        continue;
    cout<< x;
}
cout<< "\n Used continue to skip printing the value 5" << endl;
return 0;
} //end main
```

Program Output:

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Relational and Logical Operators:

The term **relational**, in *relational operator* refers to the relationships that values can have with one another for example ==, <=, !=, etc..

These operators are used to test a *single* condition.

In the term *logical operator*, **logical** refers to the ways these relationships can be connected.

We can use these operators to form more complex conditions.

Logical operators statements return 0 for false and 1 for true.

The logical operators are:

- && (logical AND)
- || (logical OR)
- ! (logical not).

<i>Operator</i>	<i>Action</i>
&&	AND
	OR
!	NOT

p	q	p&&q	p q	!p
0	0	0	0	1
0	nonzero	0	1	1
nonzero	nonzero	1	1	0
nonzero	0	0	1	0

Precedence:

Both the relational and logical operators are lower in precedence than the arithmetic operators.

The table below shows the relative precedence of the relational and logical operators:

Highest :

>	>=	<	<=
==	!=		
!			
&&			

Lowest :

||

An expression like: `20 < 1+ 12 && 20 < 34-20`
is evaluated as :

- 1- `20 < (1 + 12)` and returns 0 (false).
- 2- `20 < (44-20)` and returns 0 (false)
- 3- `0 && 0` returns 0 (false).

Types of programming errors:

- Compiler errors or syntax error

These are caught by the compiler and happen when the programmer does not follow the structure of a given construct. For example:

- forgetting parentheses after the if for the expression.
- Forgetting the semi-colon ; after a statement...

Logical Errors or runtime errors:

These errors are only detected at execution:

- The program goes off and you lose the control: Infinite loops
- The program performs an illegal operation: like Dividing by zero
- The program gives unexpected results.
- Putting a ; immediately after the **if** statement or the **while**, or **for** loops

Most common programming errors:

1. Forgetting the break in a switch structure:
C++ does not exit a switch statement if a case matches, rather it executes the statements of the other cases sequentially.
2. Confusing = and ==

The C++ `=` operator is used exclusively for assignment and returns the value assigned.

The `==` operator is used exclusively for comparison and returns an integer value (0 for *false*, not 0 for *true*). Because of these return values, the compiler often does not flag an error when `=` is used when one really wanted an `==` or vice-versa

3. Integer division

C++ uses the `/` operator for both real and integer division. If both operands are of an integral type, integer division is used, else real division is used. For example:

```
double half = 1/2;
```

This code sets `half` to 0 not 0.5! Why? Because 1 and 2 are integer constants.

To fix this, change at least one of them to a real constant.

```
double half = 1.0/2;
```

If both operands are integer variables and real division is desired, cast one of the variables to `double` (or `float`).