ID: 11757

NAME :SALMAN KHAN

SUBJECT: PROGRAMMING FUNDAMENTALS

TEACHER: DR FAZAL - E- MALIK

DATE: 29/09/2020

ANS:1 PART(A): IF STATEMENT:

When we need to execute a block of statements only when a given condition is true then we use if statement.

If statement consists a condition, followed by statement or a set of statements as shown below:

if(condition){ Statement(s); }

The statements gets executed only when the given condition is true. If the condition is false then the statements inside if statement body are completely ignored.

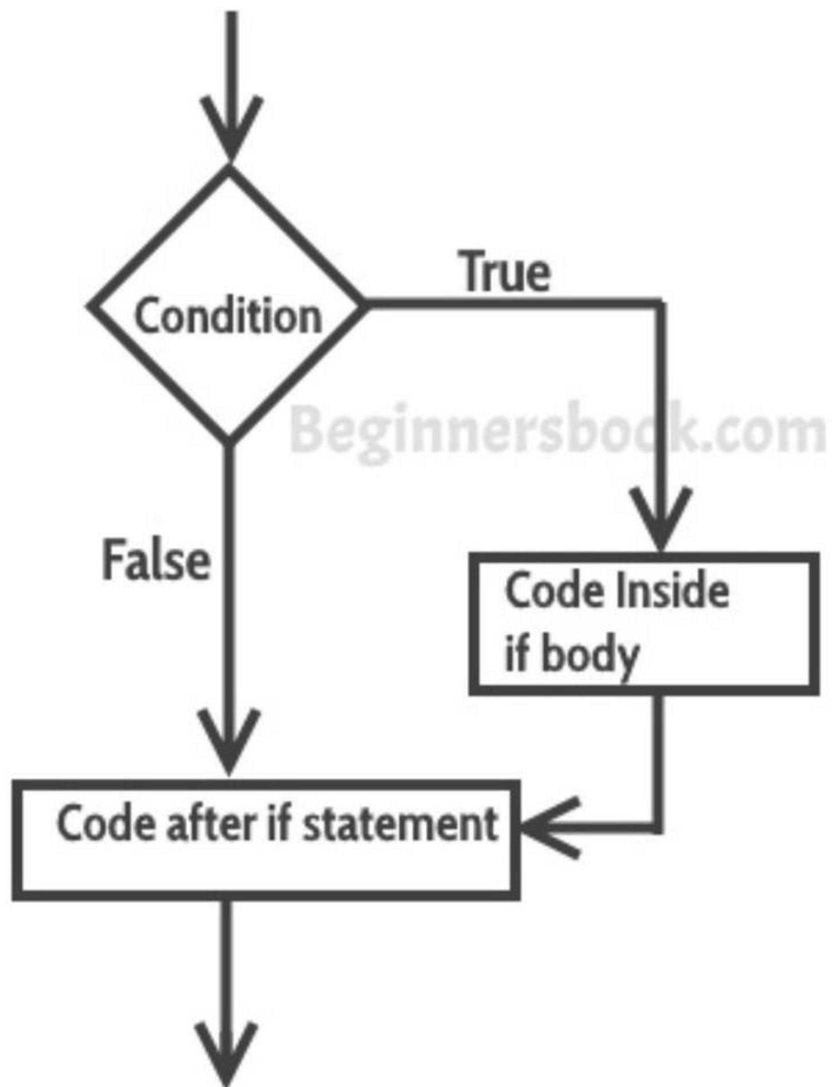
if (this condition is true)

execute this statement ;

Single-entry/single-exit

Nonzero is true, zero is false

FLOW CHART:



EXAMPLE : if (20 > 18) {

cout << "20 is greater than 18";

}

Run example »

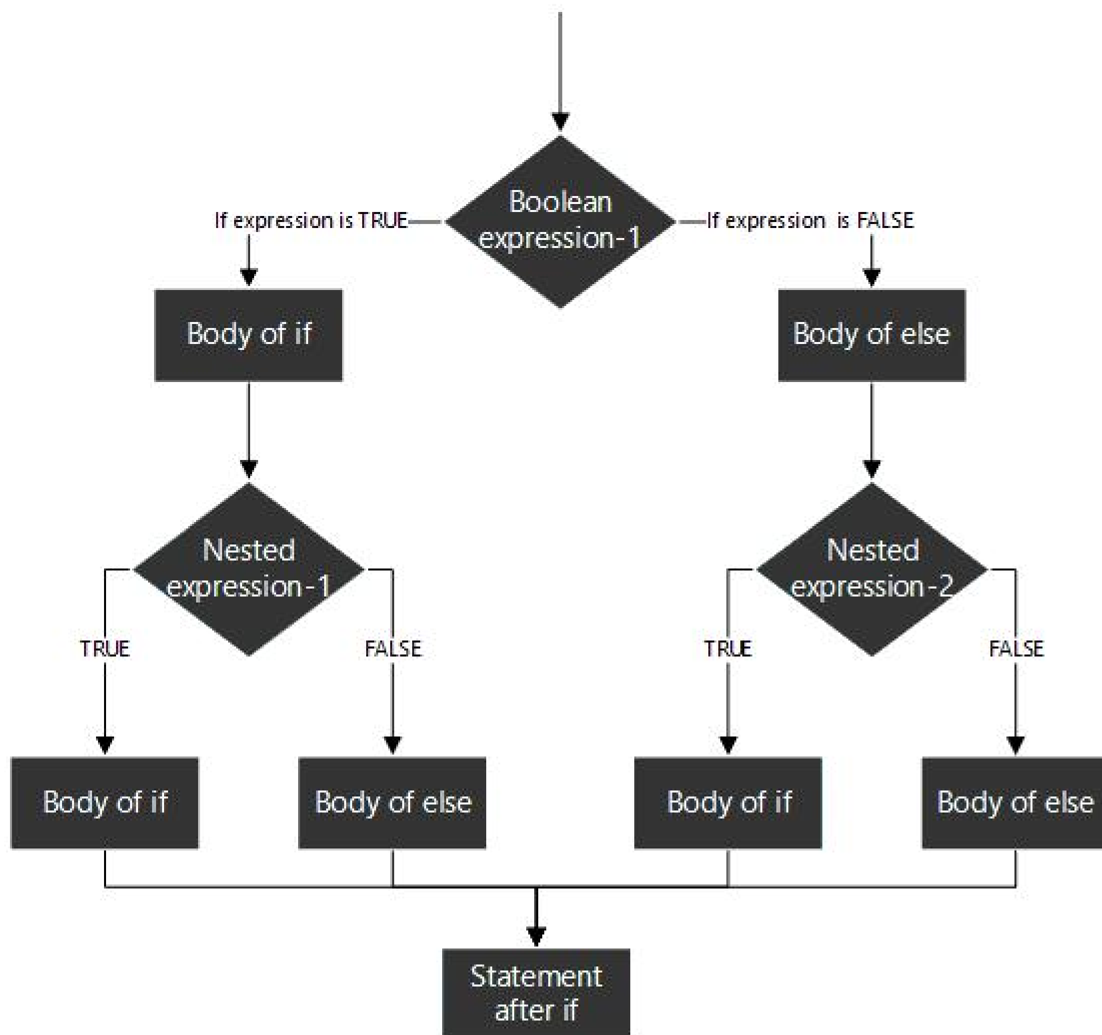
We can also test variables:

Example

```
int x = 20;

int y = 18;

if (x > y) {

  cout << "x is greater than y";

}
```

NESTED IF ELSE:

It is possible to write an entire if-else with either the if statement block or else statement block



IF ELSE: if-else

Different actions if conditions true or false

Example

if student’s grade is greater than or equal to 60

Display message “Passed”

else

Display message “Failed”

C++ Code

```
if ( grade >= 60 )
cout<<("Passed“);
else
cout<<("Failed“);
```

PART(B): #include<conio.h>

```
#include<iostream.h>

class largest

{

int d;

public :

void getdata(void);

void display_large(largest,largest);

};

void largest :: getdata(void)

{

cout<<\"\\n\\nEnter Value :-\";

cin>>d;
```

```
}
void largest :: display_large(largest o1,largest o2)
{
    if(o1.d > o2.d)
        cout<<\"\\nObject 1 contain Largest Value \"<<o1.d;
   else if(o2.d > o1.d)
        cout<<\"\\nObject 2 contain Largest Value \"<<o2.d;
   else
        cout<<\"\\nBOTH ARE EQUAL\";
}
void main()
{
largest o1,o2,o3;
clrscr();
o1.getdata();
o2.getdata();
o3.display_large(o1,o2);
getch();
}
```

ANS 2 PART(A): LOGICAL OPRATER: A logical operator is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator.[1] Common logical operators include AND, OR, and NOT.

Discussion

Within most languages, expressions that yield Boolean data type values are divided into two groups. One group uses the relational operators within their expressions and the other group uses logical operators within their expressions.

The logical operators are often used to help create a test expression that controls program flow. This type of expression is also known as a Boolean expression because they create a Boolean answer or value when evaluated. There are three common logical operators that give a Boolean value by manipulating other Boolean operand(s). Operator symbols and/or names vary with different programming languages:

Language	AND	OR	NOT
C++	&&	||	!
C#	&&	||	!
Java	&&	||	!
JavaScript	&&	||	!
Python	and	or	not
Swift	&&	||	!

The vertical dashes or piping symbol is found on the same key as the backslash \. You use the SHIFT key to get it. It is just above the Enter key on most keyboards. It may be a solid vertical line on some keyboards and show as a solid vertical line on some print fonts.

In most languages there are strict rules for forming proper logical expressions. An example is:

6 > 4 && 2 <= 14

6 > 4 and 2 <= 14

This expression has two relational operators and one logical operator. Using the precedence of operator rules the two “relational comparison” operators will be done before the “logical and” operator. Thus:

true && true

True and True

The final evaluation of the expression is: true.

We can say this in English as: It is true that six is greater than four and that two is less than or equal to fourteen.

When forming logical expressions programmers often use parentheses (even when not technically needed) to make the logic of the expression very clear. Consider the above complex Boolean expression rewritten:

(6 > 4) && (2 <= 14)

(6 > 4) and (2 <= 14)

Most programming languages recognize any non-zero value as true. This makes the following a valid expression:

6 > 4 && 8

6 > 4 and 8

But remember the order of operations. In English, this is six is greater than four and eight is not zero. Thus,

true && true

True and True

To compare 6 to both 4 and 8 would instead be written as:

6 > 4 && 6 > 8

6 > 4 and 6 > 8

This would evaluate to false as:

true && false

True and False

Truth Tables

A common way to show logical relationships is in truth tables.

Logical and (&&)

x	y	x and y

false	false	false
false	true	false
true	false	false
true	true	true

Logical or (||)

x	y	x or y
false	false	false
false	true	true
true	false	true
true	true	true

Logical not (!)

x	not x
false	true
true	false

Examples:

I call this example of why I hate “and” and love “or”.

Every day as I came home from school on Monday through Thursday; I would ask my mother, “May I go outside and play?” She would answer, “If your room is clean and your homework is done then you may go outside and play.” I learned to hate the word “and”. I could manage to get one of the tasks done and have some time to play before dinner, but both of them… well, I hated “and”.

On Friday my mother took a more relaxed viewpoint and when asked if I could go outside and play she responded, “If your room is clean or your homework is done then you may go outside and play.” I learned to clean my room quickly on Friday afternoon. Well, needless to say, I loved “or”.

For the next example, just imagine a teenager talking to their mother. During the conversation, mom says, “After all, your Dad is reasonable!” The teenager says, “Reasonable. (short pause) Not.”

Maybe college professors will think that all their students studied for the exam. Ha ha! Not. Well, I hope you get the point.

Examples:

25 < 7 || 15 > 36

15 > 36 || 3 < 7

14 > 7 && 5 <= 5

4 > 3 && 17 <= 7

! false

! (13 != 7)

9 != 7 && ! 0

5 > 1 && 7

More examples:

25 < 7 or 15 > 36

15 > 36 or 3 < 7

14 > 7 and 5 <= 5

4 > 3 and 17 <= 7

not False

not (13 != 7)

9 != 7 and not 0

5 > 1 and 7

Key Terms

logical operator

An operator used to create complex Boolean expressions.

truth tables

A common way to show logical relationships.

PART (B):

```
#include<iostream>
using namespace std;
int main()

{
float fahrenheit, celsius;

    cout << "Enter the temperature in Celsius : ";

    cin >> celsius;

    fahrenheit = (celsius * 9.0) / 5.0 + 32;
    If(fahrenheit < 30) {
    Cout<<" cool" ;
}else if (fahrenheit >=30 &&fahrenheit<35) {
Cout <<" warm" ;
}else if (fahrenheit>=35 && fahrenheit <40) {
Cout<<" tolerable" ;
}else {cout<<" very hot" }
```

ANS 3 PART(A): LOOP:

A loop is used for executing a block of statements repeatedly until a particular condition is satisfied. ... In C++ A loop is used for executing a block of statements repeatedly until a particular condition is satisfied. For example, when you are displaying number from 1 to 100 you may want set the value of a variable to 1 and display it 100 times, increasing its value by 1 on each loop iteration.

while Loops (Condition-Controlled Loops):

Both while loops and do-while loops (see below) are condition-controlled, meaning that they continue to loop until some condition is met.

Both while and do-while loops alternate between performing actions and testing for the stopping condition.

While loops check for the stopping condition first, and may not execute the body of the loop at all if the condition is initially false.

Syntax:

```
while( condition )
    body;
```

where the body can be either a single statement or a block of statements within { curly braces }.

Example:

```
int i = 0;
while( i < 5 )
    printf( "i = %d\n", i++ );
printf( "After loop, i = %d\n", i );
```

do-while Loops:

do-while loops are exactly like while loops, except that the test is performed at the end of the loop rather than the beginning.

This guarantees that the loop will be performed at least once, which is useful for checking user input among other things (see example below.)

Syntax:

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

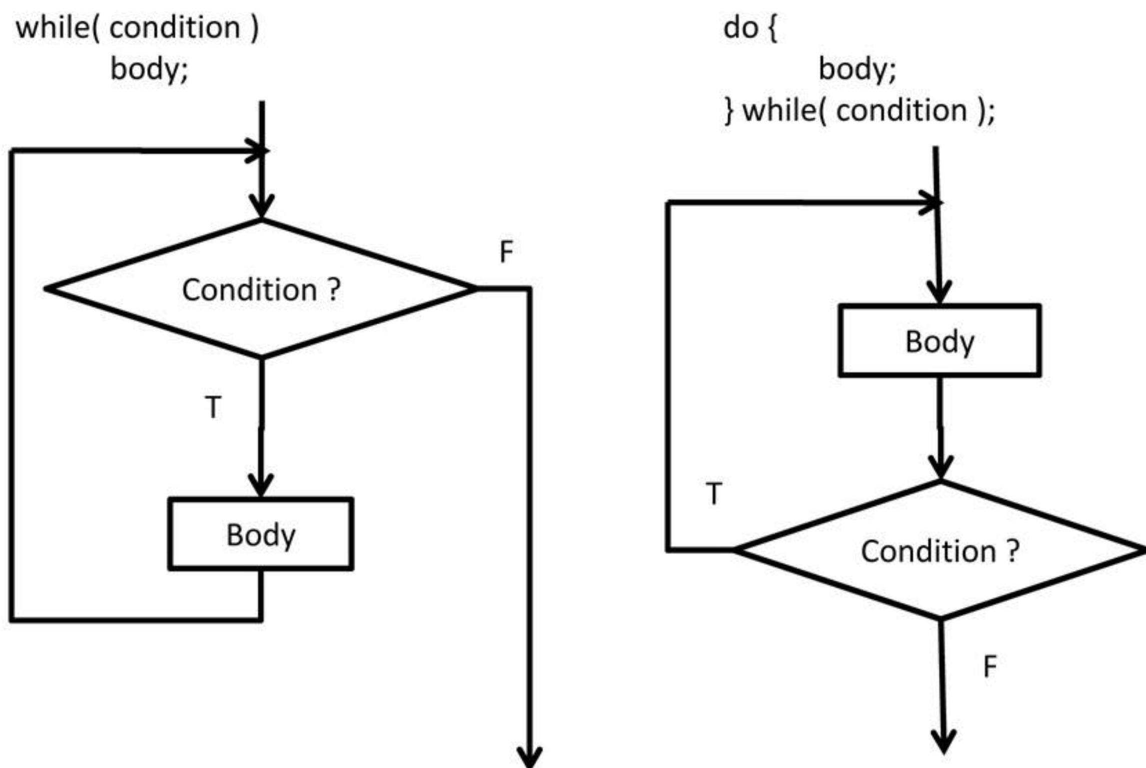
In theory the body can be either a single statement or a block of statements within { curly braces }, but in practice the curly braces are almost always used with do-whiles.

Example:

```
int month;

do {

    printf( "Please enter the month of your birth > " );

    scanf( "%d", &month );

} while ( month < 1 || month > 12 );
```

Note that the above example could be improved by reporting to the user what the problem is if month is not in the range 1 to 12, and that it could also be done using a while loop if month were initialized to a value that ensures entering the loop.

The following diagram shows the difference between while and do-while loops. Note that once you enter the loop, the operation is identical from that point forward:

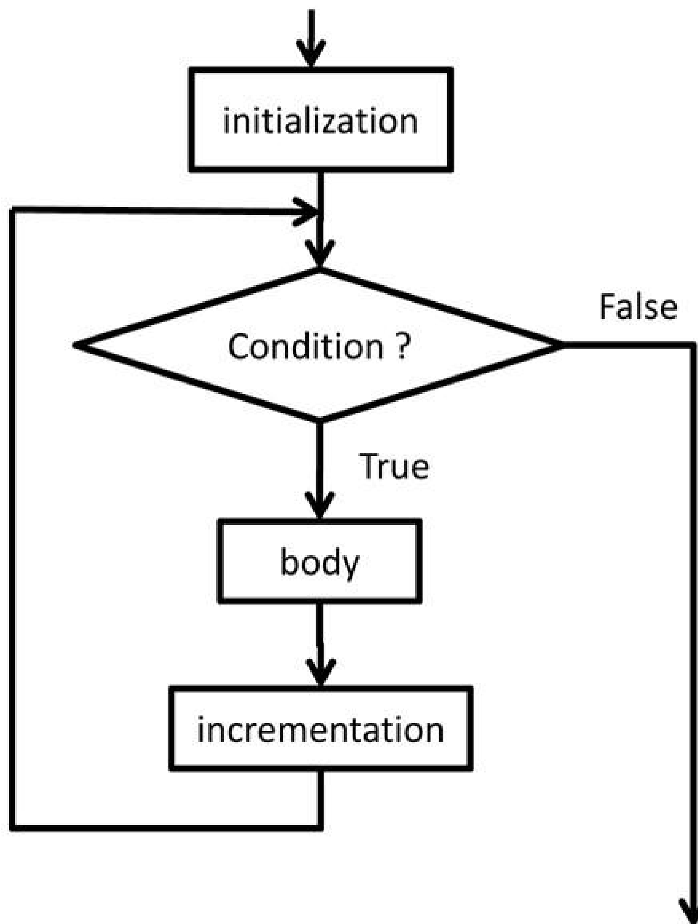


for Loops:

for-loops are counter-controlled, meaning that they are normally used whenever the number of iterations is known in advance.

Syntax:

```
for( initialization; condition; incrementation )
        body;
```



where again the body can be either a single statement or a block of statements within { curly braces }.

Details:

The initialization step occurs one time only, before the loop begins.

The condition is tested at the beginning of each iteration of the loop.

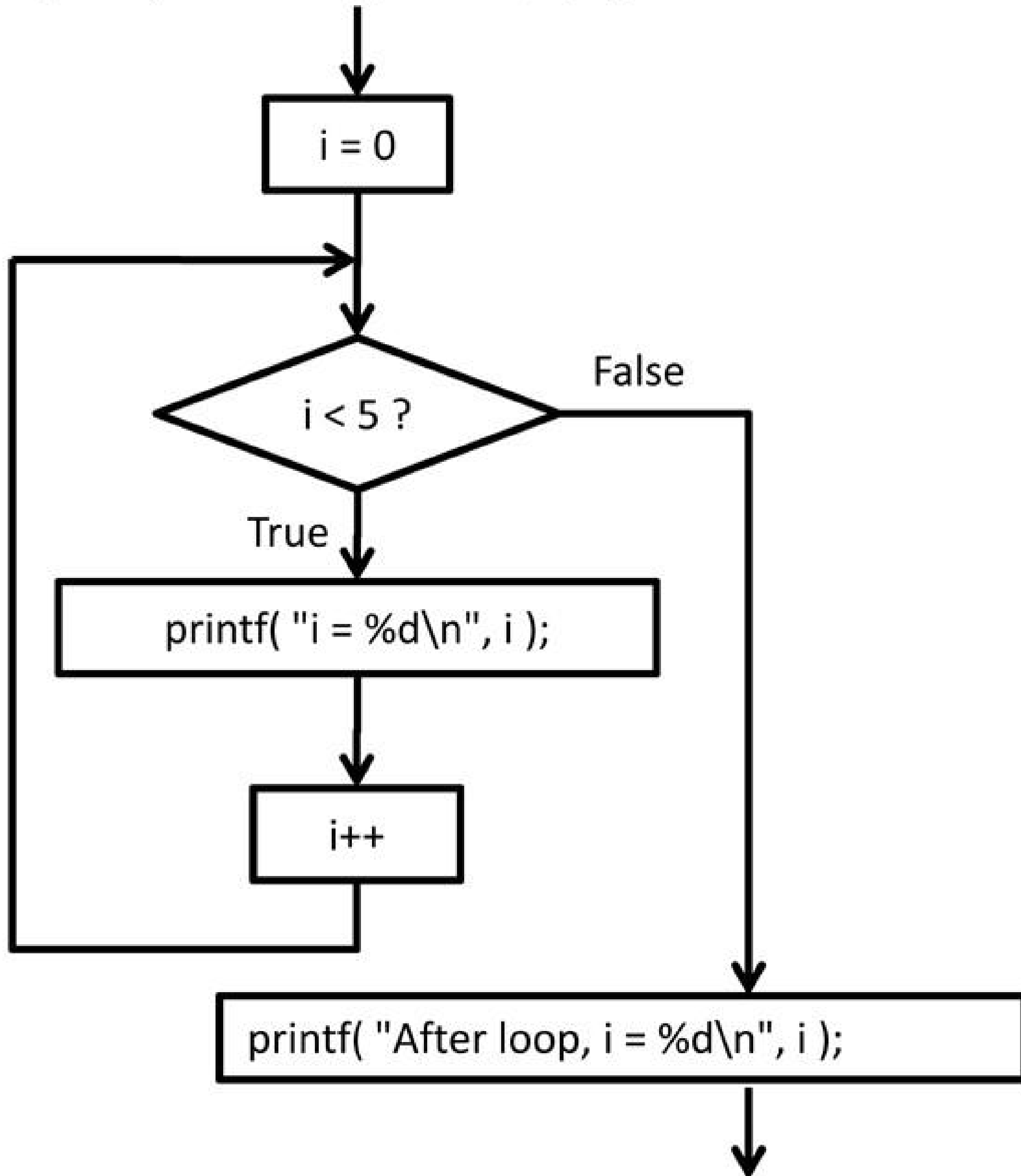
If the condition is true (non-zero), then the body of the loop is executed next.

If the condition is false (zero), then the body is not executed, and execution continues with the code following the loop.

The incrementation happens AFTER the execution of the body, and only when the body is executed.

Example:

```
int i;

for( i = 0; i < 5; i++ )
    printf( "i = %d\n", i );
printf( "After loop, i = %d\n", i );
```



Special Notes:

The third part of the loop is labeled "incrementation", because it usually takes the form of "i++" or something similar. However it can be any legal C/C++ statement, such as "N += 3" or "counter = base + delta".

In the example above, the variable i is declared before the loop, and continues to exist after the loop has completed. You will also see occasions where the loop variable is declared as part of the for loop, (in C99), in which case the variable exists only within the body of the loop, and is no longer valid when the loop completes:

```
for( int i = 0; i < 5; i++ )
    printf( "i = %d\n", i );
printf( "After loop, i is no longer valid\n" );
```

(In Dev C++ you can specify support for C99 by selecting Project->Project Options from the menu, and then selecting the Parameters tab. Under "C compiler", add the line: -std=c99)

The comma operator:

C has a comma operator, that basically combines two statements so that they can be considered as a single statement.

About the only place this is ever used is in for loops, to either provide multiple initializations or to allow for multiple incrementations.

For example:

```
int i, j = 10, sum;
for( i = 0, sum = 0; i < 5; i++, j-- )
    sum += i * j;
```

break and continue

break and continue are two C/C++ statements that allow us to further control flow within and out of loops.

break causes execution to immediately jump out of the current loop, and proceed with the code following the loop.

continue causes the remainder of the current iteration of the loop to be skipped, and for execution to recommence with the next iteration.

In the case of for loops, the incrementation step will be executed next, followed by the condition test to start the next loop iteration.

In the case of while and do-while loops, execution jumps to the next loop condition test.

(We have also seen break used to jump out of switch statements. Continue has no meaning in switch statements.)

Infinite Loops:

Infinite loops are loops that repeat forever without stopping.

Usually they are caused by some sort of error, such as the following example in which the wrong variable is incremented:

```
int i, j;

for( i = 0; i < 5; j++ )

    printf( "i = %d\n", i );

printf( "This line will never execute\n" );
```

Other times infinite loops serve a useful purpose, such as this alternate means of checking user input:

```
while( true ) {

    printf( "Please enter a month from 1 to 12 > " );

    scanf( "%d", &month );

    if( month > 0 && month < 13 )

        break;

    printf( "I'm sorry, but %d is not a valid month.\nPlease try again.\n",
month );

}
```

Empty Loops:

A common error is to place an extra semi-colon at the end of the while or for statement, producing an empty loop body between the closing parenthesis and the semi-colon, such as:

```
int i;

for( i = 0; i < 5; i++ ); // Error on this line causes empty loop

    printf( "i = %d\n", i );    // This line is AFTER the loop, not inside it.
```

or:

```
int i = 0;
while( i < 5 );    // Error - empty loop on this line
        printf( "i = %d\n", i++ ); // Again, this line is AFTER the loop.
```

In the case of the while loop shown above, it is not only empty but also infinite.

There are some very rare circumstances in which a programmer will deliberately write an empty loop, most of which are beyond the scope of this course. (This is know as a busy loop.)

In this case, the semicolon should be placed on a line by itself, and clearly commented to indicate that the empty loop is deliberate and not an oversight:

```
while( ( error = someFunction( ) ) != 0 )
        ;    // Empty loop - Does nothing forever, until someFunction returns a
zero
printf( "error = %d\n", error ); // After the loop.    error must be zero to get
here.
```

Nested Loops:

The code inside a loop can be any valid C code, including other loops.

Any kind of loop can be nested inside of any other kind of loop.

for loops are frequently nested inside of other for loops, for example to produce a simple multiplication table:

```
const int NROWS = 10;
const int NCOLS = 10;
for( int r = 0; r < NROWS; r++ ) { // Loop through rows
        for( int c = 0; c < NCOLS; c++ ) { // Loop through columns
                printf( "%5d",    r * c    ); // No newline in the middle of a row
        } // End of loop through columns
        printf( "\n" );    // One newline at the end of each row
} // End of loop through rows
```

Some programmers like to use successive integers i, j, k, l, etc. for use in nested for loops. This can be appropriate if the mathematics being implemented uses multiple ijk subscripts.

Other times it can be less confusing to use alternative variables that are more meaningful to the problem at hand, such as the r and c variables used above to keep track of rows and columns.

The limits as to how deeply loops may be nested is implementation dependent, but is usually too high to be of any concern, except in cases of extremely complex or extremely poorly written programs.

Floating Point Increments Within Loops:

Engineers and scientists frequently write iterative programs in which a floating point value steps through a range of values in small increments.

For example, suppose the "time" variable needs to change from a low of tMin to a high of tMax in steps of deltaT, where all these variables are doubles.

The obvious BUT INCORRECT approach is as follows:

```
for( time = tMin; time <= tMax; time += deltaT ) {

    // Use the time variable in the loop
```

So why is this so wrong?

If deltaT is small and/or the range is large (or both), the loop may execute for thousands of iterations.

That means that by the end of the loop, time has been calculated by the summation of thousands of addition operations.

Numbers that seem "exact" to us in decimal form, such as 0.01 are not exact when the computer stores them in binary, which means that the value used for deltaT is really an approximation to the exact value.

Therefore each addition step introduces a very small amount of roundoff error, and by the time you add up thousands of these errors, the total error can be significant.

The correct approach is as follows, if you know the minimum and maximum values and the desired change on each iteration:

```
int nTimes = ( tMax - tMin ) / deltaT + 1;

for( int i = 0; i < nTimes; i++ ) {

    time = tMin + i * deltaT;
```

```
// NOW use a more accurate time variable
```

Or alternatively if you know the minimum, maximum, and number of desired iterations:

```
double deltaT = ( tMax - tMin ) / ( nTimes - 1 );
for( int i = 0; i < nTimes; i++ ) {
    time = tMin + i * deltaT;
    // NOW use a more accurate time variable
```

In general there are four values that can be used to specify stepping through a range - the low end of the range, the high end of the range, the number of step to take, and the increment to take on each step - and if you know any three of them, then you can calculate the fourth one.

The correct loop should use an integer counter to complete the loop a given number of times, and use the low end of the range and the increment as shown to calculate the floating point loop variable at the beginning of each iteration of the loop.

So why is that better?

The number of times that the loop executes is now controlled by an integer, which does not have any roundoff error upon incrementation, so there is no chance of performing one too many or one too few iterations due to accumulated roundoff.

The time variable is now calculated from a single multiplication and a single addition, which can still introduce some roundoff error, but far less than thousands of additions.

Where does that +1 come from?

The +1 is needed in order to include both endpoints of the range.

Suppose tMax were 20 and tMin were 10, and deltaT were 2.

The desired times would be 10, 12, 14, 16, 18, 20, which is a total of 6 time values, not 5. (Five intervals if you want to look at it that way.)

(20 - 10) / 2 yields 5, so you have to add the extra 1 to get the correct number of times of 6.

Another way of looking at this is that if nTimes is the number of data points in the range, then nTimes - 1 is the number of gaps between the data points.

Example:
interpolate.c is a quick-and-dirty example of interpolating floating point numbers in a loop, that was whipped up in 10 minutes in class. It is NOT an example of good code, but it is an example of how a quick little program can be used to test out, play with, or in this case demonstrate a new or unfamiliar concept. This example

interpolates the function f(x) = x^3 over the range from -1.0 to 4.0 in steps of 0.5, using three approachs:

Constant - Take the average of the inputs at the end points, evaluate f(average input), and assume the function is constant over the range.

Linear - Evaluate the function at the endpoints, and then use a linear interpolation of the endpoint function values in between.

Non-Linear - Linearly interpolate the function inputs over the range, and at each evaulation point, evaluate the function of the interpolated inputs.

When to Use Which Loop ?

If you know (or can calculate) how many iterations you need, then use a counter-controlled (for) loop.

Otherwise, if it is important that the loop complete at least once before checking for the stopping condition,

or if it is not possible or meaningful to check the stopping condition before the loop has executed at least once,

then use a do-while loop.

Otherwise use a while loop.

PART(B):

ANS 4 PART(A): Break and Continue Statements:

The one-token statements continue and break may be used within loops to alter control flow; continue causes the next iteration of the loop to run immediately, whereas break terminates the loop and causes execution to resume after the loop. Both control structures must appear in loops. Both break and continue scope to the most deeply nested loop, but pass through non-loop statements.

Although these control statements may seem undesirable because of their goto-like behavior, their judicious use can greatly improve readability by reducing the level of nesting or eliminating bookkeeping inside loops.

Break Statements:

When a break statement is executed, the most deeply nested loop currently being executed is ended and execution picks up with the next statement after the loop. For example, consider the following program:

```
while (1) {

   if (n < 0) break;

   foo(n);

   n = n - 1;

}
```

The while~(1) loop is a “forever” loop, because 1 is the true value, so the test always succeeds. Within the loop, if the value of n is less than 0, the loop terminates, otherwise it executes foo(n) and then decrements n. The statement above does exactly the same thing as

```
while (n >= 0) {

   foo(n);

   n = n - 1;

}
```

This case is simply illustrative of the behavior; it is not a case where a break simplifies the loop.

Continue Statements:

The continue statement ends the current operation of the loop and returns to the condition at the top of the loop. Such loops are typically used to exclude some values from calculations. For example, we could use the following loop to sum the positive values in the array x,

```
real sum;

sum = 0;

for (n in 1:size(x)) {

   if (x[n] <= 0) continue;

   sum += x[n];

}
```

When the continue statement is executed, control jumps back to the conditional part of the loop. With while and for loops, this causes control to return to the conditional of the loop. With for loops, this advances the loop variable, so the the above program will not

go into an infinite loop when faced with an x[n] less than zero. Thus the above program could be rewritten with deeper nesting by reversing the conditional,

```
real sum;

sum = 0;

for (n in 1:size(x)) {

    if (x[n] > 0)

        sum += x[n];

}
```

While the latter form may seem more readable in this simple case, the former has the main line of execution nested one level less deep. Instead, the conditional at the top finds cases to exclude and doesn't require the same level of nesting for code that's not excluded. When there are several such exclusion conditions, the break or continue versions tend to be much easier to read.

PART(B):

```
#include<iostream.h>

#include<conio.h>

void main()

{

clrscr();

int i,n,sum=0;

cout<<"1+2+3+……+n";

cout<<"nEnter the value of n:";

cin>>n;

for(i=1;i<=n;++i)

sum+=i;

cout<<"nSum="<<sum;

getch();

}
```

ANS 5 PART(A): C++ CHARECTER SET: Character set is the combination of English language (Alphabets and White spaces) and math's symbols (Digits and Special symbols). Character Set means that the characters and symbols that a C++ Program can understand and accept. These are grouped to form the commands, expressions, words, c-statements and other tokens for C++ Language.

Character Set is the combination of alphabets or characters, digits, special symbols and white spaces same as learning English is to first learns the alphabets, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. More about a C++ program we can say that it is a sequence of characters. These character from the character set plays the different role in different way in the C++ compiler. The special characters are listed in Table

<table>
<tr><th colspan="5">Special Characters</th></tr>
<tr><td>+</td><td>></td><td>/</td><td>[</td><td>\</td></tr>
<tr><td>!</td><td>;</td><td>"</td><td>]</td><td>{</td></tr>
<tr><td><</td><td>*</td><td>.</td><td>%</td><td>}</td></tr>
<tr><td>:</td><td>^</td><td>,</td><td>~</td><td>#</td></tr>
<tr><td>-</td><td>(</td><td>=</td><td>-</td><td>|</td></tr>
<tr><td>?</td><td>)</td><td>'</td><td>&</td><td></td></tr>
</table>

n addition to these characters, C++ also uses a combination of characters to represent special conditions. For example. character combinations such as '\nt, '\b' and '\t' are used to represent newline, backspace and horizontal tab respectively.

When we have to learn English language we start from beginning as given below

When we have to learn C language we start from beginning as given below:

There are mainly four categories of the character set as shown in the Figure

1. Alphabets:

Alphabets are represented by A-Z or a-z. C- Language is case sensitive so it takes different meaning for small and upper case letters. By using this character set C statements and character constants can be written very easily. There are total 26 letters used in C-programming.

2. Digits:

Digits are represented by 0-9 or by combination of these digits. By using the digits numeric constant can be written easily. Also numeric data can be assigned to the C-tokens. There are total 10 digits used in the C-programming.

3. Special Symbols:

All the keyboard keys except alphabet, digits and white spaces are the special symbols. These are some punctuation marks and some special symbols used for special purpose.

There are total 30 special symbols used in the C-programming. Special symbols are used for C-statements like to create an arithmetic statement +, -, * etc. , to create relational statement <, >, <=, >=, == etc. , to create assignment statement =, to create logical statement &&, II etc. are required.

4. White Spaces:

White spaces has blank space, new line return, Horizontal tab space, carriage ctrl, Form feed etc. are all used for special purpose. Also note that Turbo-C Compiler always ignore these white space characters in both high level and low level programming

PART(B): Constants in C/C++

As the name suggests the name constants is given to such variables or values in C/C++ programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any types of constants like integer, float, octal, hexadecimal, character constants etc. Every constant has some range. The integers that are too big to fit into an int will be taken as long. Now there are various ranges that differ from unsigned to signed bits. Under the signed bit, the range of an int varies from -128 to +127 and under the unsigned bit, int varies from 0 to 255.

In C/C++ program we can define constants in two ways as shown below:

Using #define preprocessor directive

Using a const keyword

Literals: The values assigned to each constant variables are referred to as the literals. Generally, both terms, constants and literals are used interchangeably. For eg, “const int = 5;“, is a constant expression and the value 5 is refered to as constant integer literal.

Refer here for various Types of Literals in C++.

Using #define preprocessor directive: This directive is used to declare an alias name for existing variable or any value. We can use this to declare a constant as shown below:

```
#define identifierName value
```

identifierName: It is the name given to constant.

value: This refers to any value assigned to identifierName.

EXAMPLE:

```
#include<stdio.h>

#define val 10

#define floatVal 4.5

#define charVal 'G'

int main()

{

        printf("Integer Constant: %d\n",val);

        printf("Floating point Constant: %.1f\n",floatVal);

        printf("Character Constant: %c\n",charVal);

        return 0;

}
```

Output:

Integer Constant: 10

Floating point Constant: 4.5

Character Constant: G

using a const keyword: Using const keyword to define constants is as simple as defining variables, the difference is you will have to precede the definition with a const keyword.

PART(C): VARIABLE: A variable is a name given to a memory location. It is the basic unit of storage in a program.

The value stored in a variable can be changed during program execution.

A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.

In C++, all the variables must be declared before use.

How to declare variables?

A typical variable declaration is of the form:

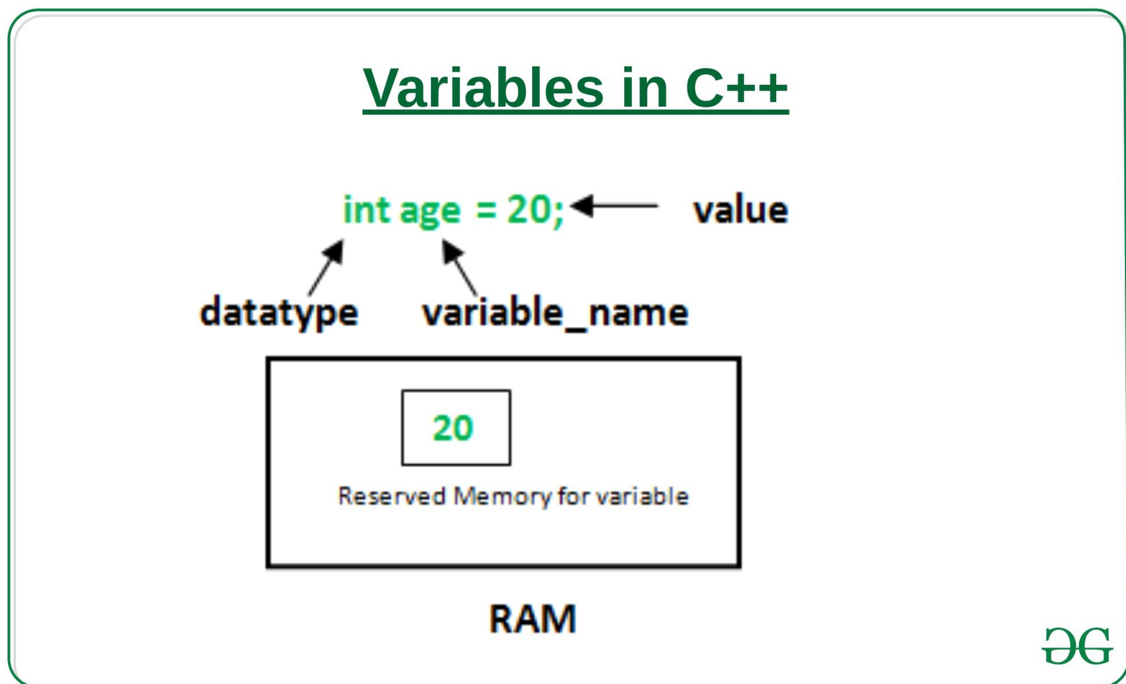
// Declaring a single variable

type variable_name;

// Declaring multiple variables:

type variable1_name, variable2_name, variable3_name;

A variable name can consist of alphabets (both upper and lower case), numbers and the underscore ‘_’ character. However, the name must not start with a number.



In the above diagram,

datatype: Type of data that can be stored in this variable.

variable_name: Name given to the variable.

value: It is the initial value stored in the variable.

Examples:

```
// Declaring float variable

float simpleInterest;

// Declaring integer variable

int time, speed;

// Declaring character variable

char var;
```

Difference between variable declaration and definition

The variable declaration refers to the part where a variable is first declared or introduced before its first use. A variable definition is a part where the variable is assigned a memory location and a value. Most of the times, variable declaration and definition are done together.

See the following C++ program for better clarification:

```
#include <iostream>

using namespace std;

int main()

{

    // declaration and definition

    // of variable 'a123'

    char a123 = 'a';

    // This is also both declaration and definition

    // as 'b' is allocated memory and

    // assigned some garbage value.

    float b;

    // multiple declarations and definitions

    int _c, _d45, e;
```

```
    // Let us print a variable
    cout << a123 << endl;
    return 0;
}
```

Output:

a

Types of variables in C++

```
class GFG {
public:
    static int a;   — Static Variable
  int b;            — Instance Variable
public:
    func()
    {
        int c;      — Local Variable
    };
};
```

Let us now learn about each one of these variables in detail.

Local Variables: A variable defined within a block or method or constructor is called local variable.

These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.

The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.

Initilisation of Local Variable is Mandatory.

Instance Variables: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.

Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

Initilisation of Instance Variable is not Mandatory.

Instance Variable can be accessed only by creating objects.

Static Variables: Static variables are also known as Class variables.

These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.

Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.

Static variables are created at the start of program execution and destroyed automatically when execution ends.

Initialization of Static Variable is not Mandatory. Its default value is 0

If we access the static variable like Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name to class name automatically.

If we access the static variable without the class name, Compiler will automatically append the class name.

To access static variables, we need not create an object of that class, we can simply access the variable as

class_name::variable_name;

Instance variable Vs Static variable

Each object will have its own copy of instance variable whereas We can only have one copy of a static variable per class irrespective of how many objects we create.

Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of instance variable. In case of static, changes will be reflected in other objects as static variables are common to all object of a class.

We can access instance variables through object references and Static Variables can be accessed directly using class name.

Syntax for static and instance variables:

```
class Example
{
    static int a; // static variable
    int b;        // instance variable
}
```

PART(D): KEYWORDS:

is a predefined or reserved word in C++ library with a fixed meaning and used to perform an internal operation. C++ Language supports more than 64 keywords.

Keywords are those words whose meaning is already defined by Compiler. These keywords cannot be used as an identifier. Note that keywords are the collection of reserved words and predefined identifiers. Predefined identifiers are identifiers that are defined by the compiler but can be changed in meaning by the user.

Every Keyword exists in lower case latter like auto, break, case, const, continue, int etc.

Keywords in C++

Keywords in C++ A keyword is a reserved word. You cannot use it as a variable name, constant name etc. A list of 32 Keywords in C++ Language which are also available in C language

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Sitesbay

Keywords in C++

Keywords in C++ A keyword is a reserved word. You cannot use it as a variable name, constant name etc. A list of 30 Keywords in C++ Language which are not available in C

asm dynamic_cast namespace reinterpret_cast bool explicit
new static_cast false catch operator template friend
private class this inline public throw
const_cast delete mutable protected
true try typeid typename using virtual wchar_t

32 Keywords in C++ Language which is also available in C language.

auto double int struct

break else long switch

case enum register typedef

char extern return union

const float short unsigned

continue for signed void

default goto sizeof volatile

do if static while

Another 30 reserved words that were not in C, these are new to C++

asm dynamic_cast namespace reinterpret_cast

bool explicit new static_cast

catch false operator template

class friend private this

const_cast inline public throw

delete mutable protected true

try typeid typename using

using virtual wchar_t

PART(E): RELATIONAL OPRATERS:

In C++ Programming, the values stored in two variables can be compared using following operators and relation between them can be determined.

Various C++ relational operators available are-

Operator, Meaning

> ,Greater than

>= , Greater than or equal to

== , Is equal to

!= , Is not equal to

< , Less than or equal to <= , Less than [/table]

As we discussed earlier, C++ Relational Operators are used to compare values of two variables. Here in example we used the operators in if statement.

Now if the result after comparison of two variables is True, then if statement returns value 1.

And if the result after comparison of two variables is False, then if statement returns value 0.

EXAMPLE:

```
#include<iostream>

using namespace std;

int main()

{

    int a=10,b=20,c=10;

    if(a>b)

        cout<<"a is greater"<<endl;

    if(a<b)

        cout<<"a is smaller"<<endl;
```

```
    if(a<=c)
        cout<<"a is less than/equal to c"<<endl;
    if(a>=c)
        cout<<"a is less than/equal to c"<<endl;
    return 0;
}
```

Output:

a is smaller

a is less than/equal to c

a is greater than/equal to c