Chapter Three
Program Control Constructs.

- A running program spends all of its time executing statements. The order in which statements are executed is called *flow control* (or *control flow*).
- This term reflects the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement.
- Flow control in a program is typically *sequential*, from one statement to the next, but may be divided to other paths by *branching statements*.
- Flow control is an important consideration because it determines what is executed during a run and what is not, therefore, affecting the overall outcome of the program.

Selection Statements.

The if Statement.

- It is sometimes desirable to make the execution of a statement dependent upon a *condition being satisfied*. The if statement provides a way of expressing this, the general form of which is:

```
if (expression)
        statement;
```

- First expression is evaluated. If the outcome is non zero (true), then the statement is executed. Otherwise, nothing happens (the statement will not be executed).
- To make multiple statements dependent on the same condition we can use a compound statement.
- E.g.

```
if(balance > 0)
{
        interest = balance * creditRate;
        balance += interest;
}
```

- A variant form of the if statement allows us to specify two alternative statements: one which is executed if a condition is *satisfied* and one which is executed if the condition is *not satisfied*. This is called the *if-else* statement and has the general form:

```
if(expression)
        statement1;
else
        statement2;
```

- First expression is evaluated. If the outcome is non zero(true), then statement1 will be executed. Otherwise, statement2 will be executed.
- For e.g.

```
if(balance > 0)
        {
```

```
                interest = balance * creditRate;
                balance += interest;
        }
        else
        {
                interest = balance * debitRate;
                balance += interest;
        }
```

- 'if' statements may be nested by having an if statement appear inside another if statement. For instance:

```
if(callHour > 3)
{
        if(callDuration <= 5)
                charge = callDuration * tarif1;
        else
                charge = 5 * tarif1 + (callDuration - 5) * tarif2;
}
else
        charge = flatFee;
```

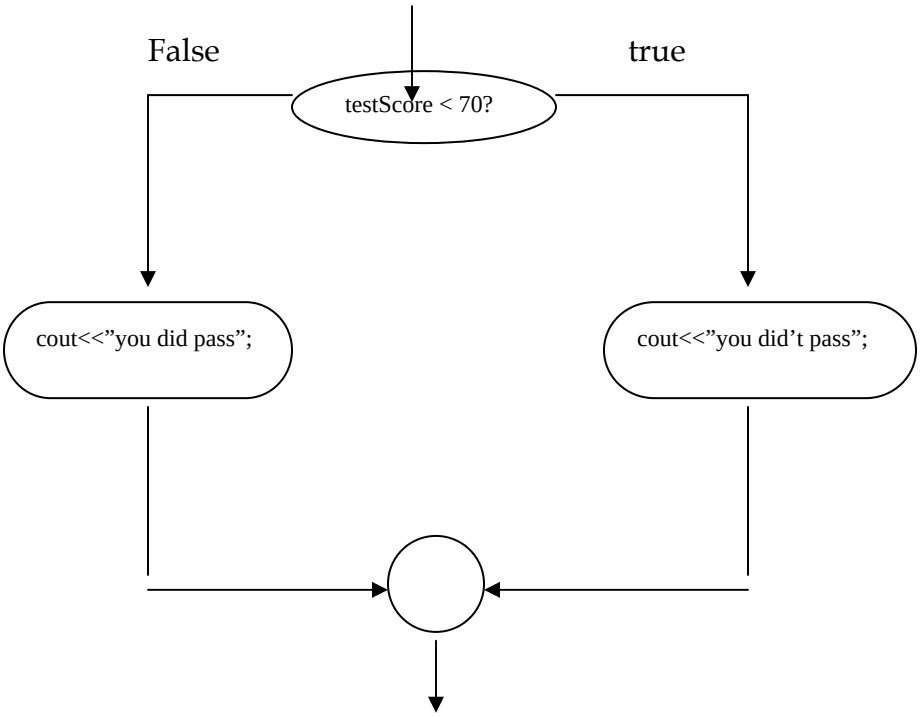
e.g2.

```
#include<iostream>

int main()
{
        int testScore;
        cout<<"\nEnter your test score:";
        cin>>testScore;

        if(testScore < 70)
                cout<<"\n You did not pass:"; → then block
        else
                cout<<"\n You did pass"; →  else block
        getch();
        return 0;
}
```

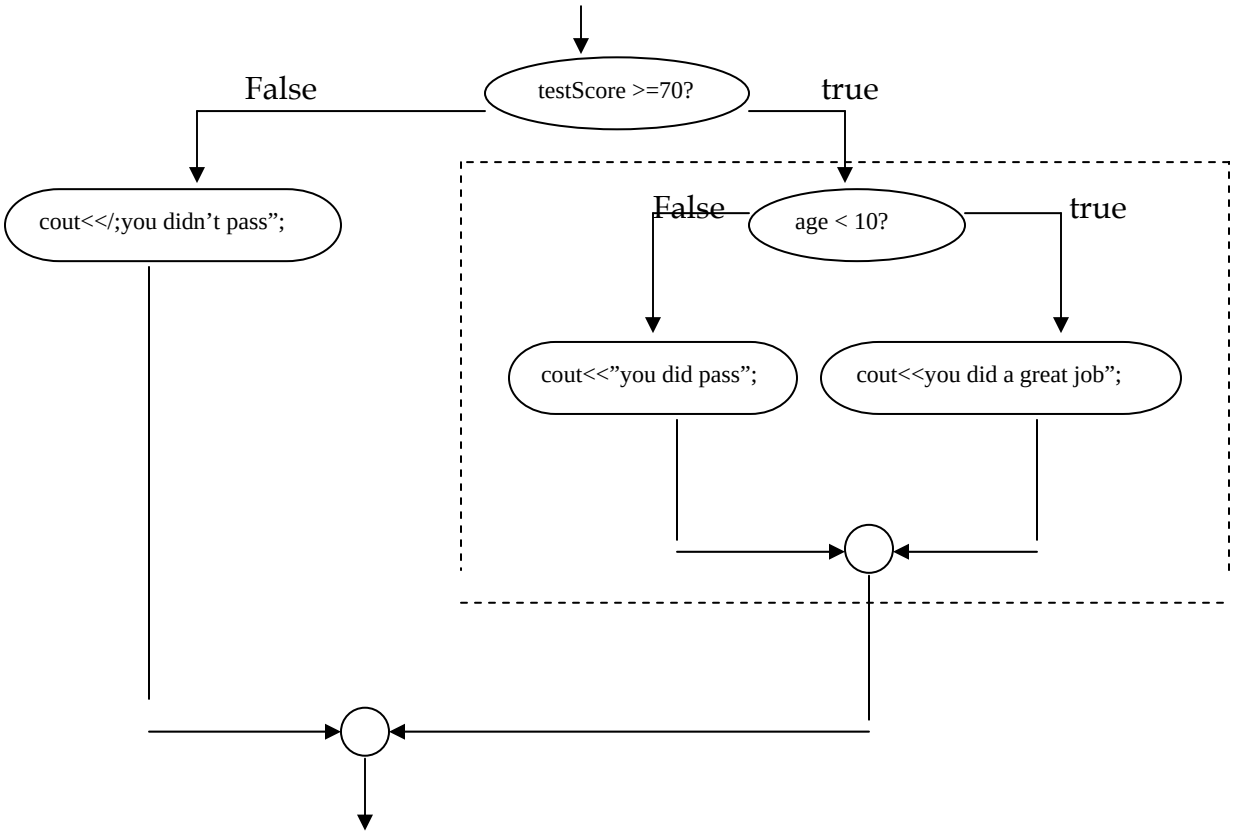
- the above sample program can be diagrammatically expressed as follows:



- Lets have an example on nested if statements

```
#...
#...
int main()
{
        int testScore, age;
        cout<<"\n Enter your test score :";
        cin>>testScore;
        cout<<"\n enter your age:";
        cin>>age;
        if(testScore >= 70)
        {
                if(age < 10)
                        cout<<"\n You did a great job";
                else
                        cout<<"\n You did pass";
        }
        else
                cout<<"\n You did not pass";
        getch();
        return 0;
}
```

- The above program can be expressed diagrammatically as follows.



- The if-else statement is the standard way to indent (to improve readability) of the nested if else statement.

- E.g.
```
if(score >= 90)
        cout<< "\n your grade is A";
else if(score >= 80)
        cout<< "\n your grade is B";
else if(score >= 70)
        cout<< "\n your grade is C";
else if(score >= 60)
        cout<< "\n your grade is D";
else
        cout<< "\n your grade is F";
```

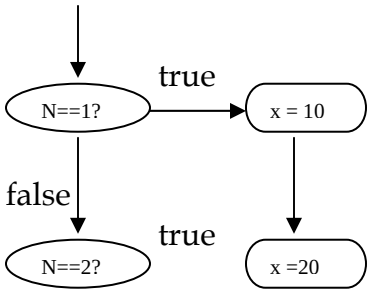
The Switch Statement

- Another C++ statement that implements a selection control flow is the switch statement (*multiple-choice statement*). The switch statement provides a way of choosing between a set of alternatives based on the value of an expression. The general form of the switch statement is:

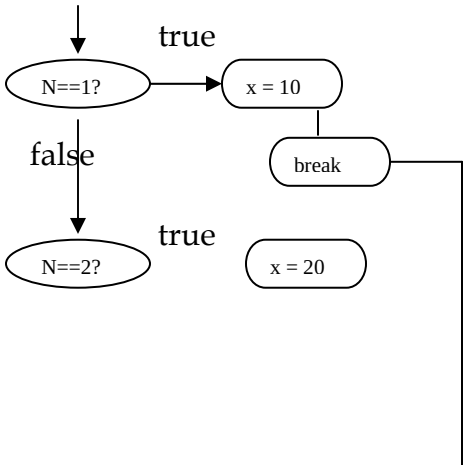
```
switch(expression)
{
        case constant1:
                statements;
        …
        case constant n:
                statements;
        default:
                statements;
}
```

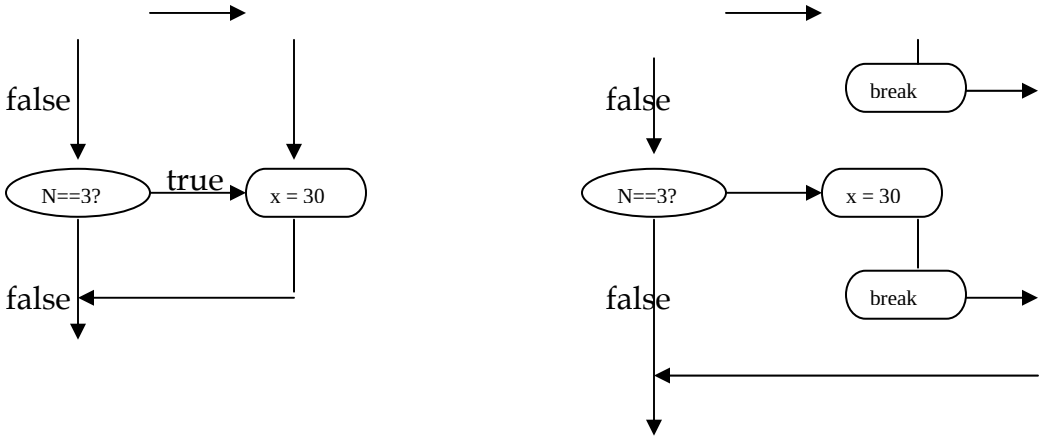
- First expression (called the *switch tag*) is evaluated, and the outcome is compared to each of the numeric constants (called *case labels*), in the order they appear, until a match is found.
- The statements following the matching case are then executed. Note the plural: each case may be followed by zero or more statements (not just one statement).
- Execution continues until either a *break* statement is encountered or all intervening statements are executed.
- The final default case is optional and is exercised if none of the earlier cases provide a match.
- Now let us see the effect of including a break statement in the switch statement.

```
switch (N){
        case 1: x=10;
        case 2: x=20;
        case 3: x=30;
}
```




```
switch(N){
    case 1: x=10; break;
    case 2: x=20; break;
    case 3: x=30; break;
}
```





- The break terminates the switch statement by jumping to the very end of it.
- There are, however, situations in which it makes sense to have a case without a break. For instance:

```
switch(operator)
{
        case '+': result = operand1 + operand2;
                  break;
        case '-' : result = operand1 – operand2;
                  break;
        case 'x':
        case '*':  result = operand1 * operand2;
                  break;
        case '/':   result = operand1 / operand2;
                  break;
        default: cout<<" unknown operator:"<<operator<<"\n";
}
```

- Because case 'x' has no break statement (in fact no statement at all!), when this case satisfied, execution proceeds to the statements of the next case and the multiplication is performed.
- Switch evelautes expression and compares the result to each of the case values. Note, however, that the evaluation is only for *equality*; relational operators may not be used in switch statements, nor can boolean operators.

Repetition Statements.

- Repetition statements control a block of code to be executed for a fixed number of times or until a certain condition is met.

- We will see C++'s three repetition statements: *while, do-while* and *for loop*

The for statement / loop

- The "for" statement (also called loop) has two additional components: an expression which is evaluated only once before every thing and an expression which is evaluated once at the end of each iteration. The general form is

```
for(expression1 ; expression2 ; expression3)
        statement;
```

- First expression1 is evaluated. Each time round the loop, expression2 is evaluated. If the outcome is non zero then statement is executed and expression3 is evaluated. Otherwise, the loop is terminated.
- The general format can be expressed as follows for the sake of clarity:

```
for(initialization ; condition ; increase/decrease)
        statement;
```

Steps of execution of the for loop:

1. Initialization is executed. (will be executed only once)
2. Condition is checked, if it is true the loop continues, otherwise the loop finishes and statement is skipped.
3. Statement is executed.
4. Finally, whatever is specified in the increase or decrease field is executed and the loop gets back to step 2.

E.g. guess the output of the following code:

```
int main()
{
        for(int i=10;i>0;i--)
        {
                cout<<n<<",";
        }
        cout<< "FIRE!";
        getch();
        return 0;
}
```

- The initialization and increase fields are optional. They can be avoided but not the semi colon signs among them.

e.g. for(;n<10;) if we want no initialization neither increase
for(;n<10;n++) if no initialization is needed.
For(; ;) is an infinite loop.

- Optionally, using the comma operator (,) we can specify more than one instruction in any of the two fields included in a for loop.

- E.g. for(n=0,i=100;n!=I;n++,i--)
  {
  //what ever here
  }

The while statement.

- The *while* statement (also called *while loop*) provides a way of repeating a statement or a block while a condition *holds / is true.*
- The general form of the while loop is:
  while(expression)
  statement;
- First expression (called the loop condition) is evaluated. If the outcome is non zero then statement (called the loop body) is executed and the whole process is repeated. Otherwise, the loop is terminated.
- Suppose that we wish to calculate the sum of all numbers from 1 to some integer denoted by n. this can be expressed as :
- E.g i=1;
  sum = 0;
  while(i <= n)
  sum += i++;

  e.g2. while(number <= 100)
  {
  sum += number;
  number++;
  }
- The second example of the while statement is called count-controlled loops because the loop body is executed for a fixed number of times. Now let us see another example.
- E.g3. cout<<"\n enter your age [between 0 and 130]:";
  cin>>age;
  while(age < 0 || age > 130)
  {
  cout<<"\n invalid age. Plz try again!";
  cin>>age;
  }
- This example of while loop is called *sentinel-controlled loops*. With a sentinel-controlled loop, the loop body is executed repeatedly until any one of the designated values called a sentinel is encountered. Here the sentinel is any valid age between 0 and 130 (breaking point).

Do…while loop.

- The do statement (also called the do loop) is similar to the while statement, except that its body is executed first and then the loop condition is examined. The general form is:

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

- First statement is executed and then expression is evaluated. If the outcome of the expression is nonzero, then the whole process is repeated. Otherwise the loop is terminated.
- E.g. what do you think is the outcome of the following code:

```
int main()
{
        unsigned long n;
        do{
                cout<<"\n enter number (0 to end):";
                cin>>n;
                cout<<"\n you entered:"<<n;
        }while(n != 0);
        getch();
        return 0;
}
```

Pitfalls in witting repetition statements.

- *Infinite loop*: no matter what you do with the while loop (and other repetition statements), make sure that the loop will *eventually terminates*.
- E.g.

```
int product = 0;
while(product < 50)
        Product *= 5;
```

Do you know why this is an infinite loop?

e.g2.

```
int counter = 1;
while(counter != 10)
        counter += 2;
```

- In the second example, the variable counter is initialized to 1 and increment is 2, counter will never be equal to 10. In theory, this while loop is an infinite loop, but in practice, this loop eventually terminate because of an *overflow error*.
- *Off-by-one error*: another thing for which you have to watch out in writing a loop is the so called *off-by-one* error. Suppose we want to execute the loop body 10 times. Does the following code work?

```
count = 1;
while(count < 10)
{
        …
        count++;
}
```

- No, the loop body is executed nine times. How about the following?

```
count = 0;
while(count <= 10)
{
        …
        count++;
}
```

- No this time the loop body is executed eleven times. The correct is

```
count = 0;
while(count < 10)
{
        …
        count++;
}
```

OR

```
count = 1;
while(count <= 10)
{
        …
        count++;
}
```

The continue and break statements

The continue statement

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It is an error to use the continue statement outside a loop.
- In while and do while loops, the next iteration commences from the loop condition.
- In a for loop, the next iteration commences from the loop's third expression.

e.g.

```
int main()
{
        for(int n=10;n>0;n--)
```

```
        {
            if(n==5)
                continue;       //causes a jump to n—
            cout<<n<< “,”;
        }
        getch();
        return 0;
    }
```

- When the continue statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops. For e.g., in the following set of nested loops, the continue statement applies to the for loop, and not to the while loop.

```
    while(more)
    {
        for(i=0;i<n;i++)
        {
            cin>>num;
            if(num<0)
                continue; //causes a jump to : i++
                …
        }
        //etc
    }
```

The break statement.

- A break statement may appear insode a loop (while,do, or for) or a switch statement. It causes a jump out of these constructs, and hence teriminates them.
- Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it. It is an error to use the break statement outside a loop or a switch statement.
- E.g.

```
    int main()
    {
        int n;
        for(n=10;n>0;n--)
        {
            cout<<n<< “,”;
            if(n = = 3)
            {
                cout<< “count down aborted!!”;
                break;
            }
        }
```

```
        getch();
        return 0;
}
```

Chapter Four.
Functions.

4.1 What is a function?

- A function provides a convenient way of packaging a computational recipe, so that it can be used as often as required.
- Therefore, a function is a block of code designed to tackle a specific problem.

4.1.1 Function Basics

- One of the best ways to attack a problem is to start with the overall goal, then divide this goal into several smaller tasks. You should never lose sight of the overall goal, but think also of how individual pieces can fit together to accomplish such a goal.
- If your program does a lot, break it into several functions. Each function should do only one primary task.

4.1.2. Summarized function basics.

- C++ functions generally adhere to the following rules.

1. Every function must have a name.
2. Function names are made up and assigned by the programmer following the same rules that apply to naming variables. They can contain up to 32 characters, they must begin with a letter, and they can consist of letters, numbers, and the underscore (_) character.
3. All function names have one set of parenthesis immediately following them. This helps you (and C++ compiler) differentiate them from variables.
4. The body of each function, starting immediately after parenthesis of the function name, must be enclosed by braces.

Declaring, defining and calling functions

Declaring function: the interface of a function (also called its prototype) specifies how it may be used. It consists of three entities:

- The function name. this is simply a unique identifier
- The function parameters (also called its signature). This is a set of zero or more typed identifiers used for passing values to and from the function.
- The function return type. This specifies the type of value the function returns. A function which returns nothing should have a return type void.

Defining a function: a function definition consists of two parts: interface (prototype) & body. The brace of a function contains the computational steps (statements) that computerize the function. The definition consists of a line called the decelerator.

Calling the function: using a function involves 'calling' it. A function call consists of the function name followed by the call operator brackets '()', inside which zero or more comma-separated arguments appear. The number of arguments should match the number of function parameters. Each argument is an expression whose type should match the type of the corresponding parameter in the function interface.

- When a function call is executed, the arguments are first evaluated and their resulting values are assigned to the corresponding parameters. The function body is then executed. Finally the return value (if any) is passed to the caller.
- Lets have a simple example:

```
void func1();

void main()
{
        ---------------
        ---------------
        ---------------
        func1();
        ---------------
        ---------------
        ---------------
}
```

no return type.

function call (notice the semicolon)

```
void func1()
{
        ---------------
        ---------------
        ---------------
```

Return type

o semicolon. Decelerator

function body

function definition

no semicolon.

Function declaring, calling and defining syntax.

- Lets have a very simple function example:

```
#include<iostream>

void starLine();
int main()
{
        starLine();
        cout<< "Data type Range" << endl;
```

```
        starLine();
        cout<< "char        -128 to 127" <<endl
             << "short     -32,768  to 32,767" <<endl
            << " int    system dependent" <<endl
            << "long    -2,147,483,648  to 2,147,483,647" << endl;
            starLine();
            return 0;
}

void starLine()
{
        for(int j=0;j<45;j++)
                cout<< "*";
        cout<<endl;
}
```

- A function call in C++ is like a detour on a highway. Imagine that you are traveling along the "road" of the primary function called main(). When you run into a function-calling statement, you must temporarily leave the main() function and execute the function that was called. After that function finishes (its return statement is reached), program control reverts to main(). In other words, when you finish a detour, you return to the "main" route and continue the trip. Control continues as main() calls other functions.
- Given the next program, which function is the calling function and which is the called function?

```
#include<iostream>

void nextMsg()
int main()
{
        cout<< "Hello!\n";
        nextMsg();
        return 0;
}

void nextMsg()
{
        cout<< "GoodBye!\n";
        return;
}
```

Function Parameters and arguments.

- C++ supports two styles of parameters: value and reference.

Passing by value:

- A value parameter receives a copy of the value of the argument passed to it. As a result, if the function makes any changes to the parameters, this will not affect the argument. For instance:

```
#.....
#.....
Void Foo(int num)
{
      Num = 0;
      cout<< "num = " << num << " \n";
}
int main(void)
{
        int x = 10;
        Foo(x);
        cout<< "x = "<<x<< "\n";
        getch();
        return 0;
}
```

- The single parameter of Foo is a value parameter. As far as this function is concerned, num behaves just like a local variable inside the function. When the function is called and x passed to it, num receives a copy of the value of x. As a result, although num is set to 0 by the function, this does not affect x. the program produces the following output:

```
Num = 0
x = 10
```

- Passing arguments in this way, where the function creates copies of the arguments passed to it is called *passing by value*.

Passing by Reference:

- A reference parameter, on the other hand, receives the argument passed to it and works on it directly. Any changes made by the function to a reference parameter is in effect directly applied to the argument.
- Passing parameters in this way is called *pass-by-reference*.
- Suppose you have pairs of numbers in your program and you want to be sure that the smaller one always precedes the larger one. To do this, you call a function, order(), which checks two numbers passed to it by reference and swaps the originals if the first is larger than the second.

```
#.......
#.......
void order(int &, int &);
int main()
{
        int n1 = 99, n2=11;
        int n3 = 22, n4=88;
        order(n1,n2);
        order(n3,n4);
        cout<< "n1="<<n1<<endl;
        cout<< "n2="<<n2<<endl;
        cout<< "n3="<<n3<<endl;
        cout<< "n4="<<n4<<endl;
        return 0;
}

void order(int & num1,int & num2)
{
        if(num1 > num2)
        {
                int temp=num1;
                num1 = num2;
                num2 = temp;
        }
}
```

- In main() there are two pairs of numbers-the first pair is not ordered and the second pair is ordered. The order() function is called once for each pair, and then all the numbers are printed out. The output revels that the first pair has been swapped while the second pair has not. Here it is:

```
N1 = 11
N2 = 99
N3 = 22
N4 = 88
```

- In the order() function the first variable is called num1 and the second is num2. If num1 is greater than num2, the function stores num1 in temp, puts num2 in num1, and finally puts temp back in num2.
- Using reference arguments in this way is a sort of remote control operation. The calling program tells the function what variables in the calling program to operate on, and the function modifies these variables without ever knowing their real names.

Global versus local variables

- Everything defined at the program scope level (outside functions) is said to have a *global scope*, meaning that the entire program knows each variable and has the capability to change any of them.

Eg.
```
int year = 1994;//global variable
int max(int,int);//gloabal funcrion
int main(void)
{
        //…
}
```

- Global variables are visible ("known") from their point of definition down to the end of the program.
- Each block in a program defines a local scope. Thus the body of a function represents a local scope. The parameters of a function have the same scope as the function body.
- Variables defined within a local scope are visible to that scope only. Hence, a variable need only be unique within its own scope. Local scopes may be nested, in which case the inner scope overrides the outer scopes. Eg:

```
int xyz;//xyz is global
void Foo(int xyz)//xyz is local to the body of Foo
{
        if(xyz > 0)
        {
                Double xyz;//xyz is local to this block
                …
        }
}
```

Scope Operator

- Because a local scope overrides the global scope, having a local variable with the same name as a global variable makes the latter inaccessible to the local scope. Eg

```
int num1;
Void fun1(int num1)
{
    //…
}
```

- The global num1 is inaccessible inside fun1(), because it is overridden by the local num1 parameter.

- This problem is overcome using the scope operator '::' which takes a global entity as argument.

```
int num1 = 2;
void fun1(int num1)
{
   //…
   if(::num1 != 0)//refers to global num1
        //…
}
```

Automatic versus static variables

- The terms automatic and static describe what happens to local variables when a function returns to the calling procedure. By default, all local variables are automatic, meaning that they are erased when their function ends. You can designate a variable as automatic by prefixing its definition with the term auto.
- Eg. The two statements after main()'s opening brace declared automatic local variables:

```
main()
{
        int i;
        auto float x;
        …
}
```

- The opposite of an automatic is a static variable. All global variables are static and, as mentioned, all static variables retain their values. Therefore, if a local variable is static, it too retains its value when its function ends-in case this function is called a second time.

- To declare a variable as static, place the static keyword in front of the variable when you define it. The following code section defines three variables i, j, k. the variable i is automatic, but j and k are static.
- Eg.

```
void my_fun()
{
        int i;
        static j = 25;
        static k = 30;
}
```

- **N.B**. if local variables are static, their values remain in case the function is called again.

Inline functions

- Suppose that a program frequently requires to find the absolute value of an integer quantity. For a value denoted by n, this may be expressed as:

```
(n > 0 ? n : -n)
```

- However, instead of replicating this expression in many places in the program, it is better to define it as a function:

```
int Abs(int n)
{
    return n > 0 ? n : -n;
}
```

- The function version has a number of disadvantages. First, it leads to a more readable program. Second, it is reusable.
- The disadvantage of the function version, however is that its frequent use can lead to considerable performance penalty due to overheads associated with calling a function.
- The overhead can be avoided by defining Abs as an inline function.

```
inline int Abs(int n)
{
   Return n > 0 ? n : -n;
}
```

- The effect of this is that when Abs is called, the compiler, instead of generating code to call Abs, expands and substitute the body of Abs in place of the call. While essentially the same computation is performed, no function call is involved and hence no stack frame is allocated.

Default arguments and function overloading

- C++ has two capabilities that regular C doesn't have. Default arguments and function overloading.
- Default argument is a programming convenience which removes the burden of having to specify argument values for all of a function parameters.
- Eg. You might pass a function an error message that is stored in a character array, and the function displays the error for a certain period of time. The prototype for such a function can be this:

```
Void pr_msg(char note[]);
```

- Therefore, to request that pr_msg() display the line 'Turn printer on', you call it this way:

```
Pr_msg("Turn printer on");
```

- As you write more of the program, you begin to realize that you are displaying one message-for instance, the 'Turn printer on' msg-more often than any other message.
- Instead of calling the function over and over, typing the same message each time, you can set up the prototype for pr_msg() so that it defaults to the 'turn printer on' message in this way:

```
Void pr_msg(char note[] = "Turn printr on");
```

- This makes your programming job easier. Because you would usually want pr_msg() to display 'turn printer on', the default argument list takes care of the message and you don't have to pass the message when you call the function.

Overloaded Functions

- Unlike C, C++ lets you have more than one function with the same name. In other words, you can have three functions called abs() in the same program.
- Functions with the same name are called overloaded functions. C++ requires that each overloaded functions differ in its argument list. Overloaded functions enable you to have similar functions that work on different types of data.
- Suppose that you wrote a function that returned the absolute value of what ever number you passed to it:

```
int iabs(int i)
{
        if(i<0)
            Return (i*-1);
        else
            Return (i);
}

float fabs(float x)
{
         if(x<0.0)
            Return (x * -1.0);
         else
             Return (x);
}
```

- With out using overloading, you have to call the function as:

```
int ans = iabs(weight);//for int arguments
float ans = fabs(weight);//for float arguments
```

- But with overloading, the above code can be used as:

```
int abs(int i);
float abs(float x);
int main()
{
        …
        ians = abs(i);//calling abs with int arguments
        fans = abs(p);//calling abs with float arguments
        …
}
```

```
int abs(int i)
{
        if(i<0)
                Return i*-1;
        else
                Return I;
}

float abs(flaot x)
{
        if(x<0.0)
                Return x*-1.0;
        else
                Return x;
}
```

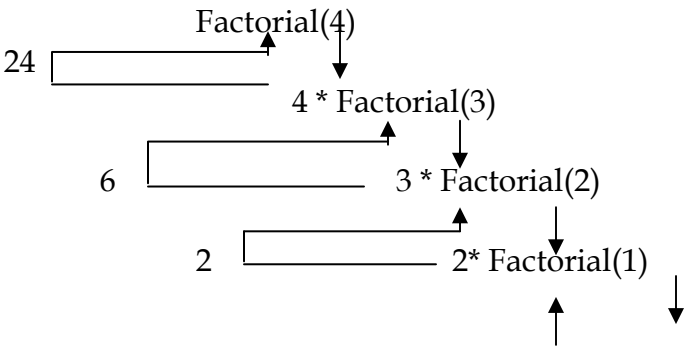
- **N.B**: if two or more functions differ only in their return types, C++ can't overload them. Two or more functions that differ only in their return types must have different names and can't be overloaded

Recursion.

- A function which calls itself is said to be recursive. Recursion is a general programming technique applicable to problems which can be defined interms of themselves. Take the factorial problem, for instance which is defined as:
    - factorial of 0 is 1
    - factorial of a positive number n is n time the factorial of n-1.
- The second line clearly indicates that factorial is defined in terms of itself and hence can be expressed as a recursive function.

```
int Factorial(unsigned int n )
{
        return n = = 0 ? 1 : n * factrial(n-1);
}
```

- For n set to 4, the following figure shows the recursive call:



1 ————1

- The stack frames for these calls appear sequentially on the runtime stack, one after the other.
- A recursive function must have at least one termination condition which can be satisfied. Otherwise, the function will call itself indefinitely until the runtime stack overflows.
- The three necessary components in a recursive method are:
    1. A test to stop or continue the recursion
    2. An end case that terminates the recursion
    3. A recursive call(s) that continues the recursion
- let us implement two more mathematical functions using recursion
- e.g the following function computes the sum of the first N positive integers 1,2,…,N. Notice how the function includes the three necessary components of a recursive method.

```
int sum(int N)
{
        if(N==1)
                return 1;
        else
                return N+sum(N);
}
```

- The last method computes the exponentiation A^n where A is a real number and N is a positive integer. This time, we have to pass two arguments. A and N. the value of A will not change in the calls, but the value of N is decremented after each recursive call.

```
float expo(float A, int N)
{
        if(N==1)
                return A;
        else
                return A * expo(A,N-1);
}
```

- Try to use a recursive function call to solve the Fibonacci series. The Fibonacci series is :

    0,1,1,2,3,5,8,13,21,…

- the recursive definition of the Fibonacci series is as follows:

    Fibonacci (0) =0
    Fibonacci (1) =1
    Fibonacci (n) =Fibonacci (n-1) +Fibonacci (n-2);

Recursion versus iteration

- Both iteration and recursion are based on control structure. Iteration uses a repetition structure (such as for, while, do…while) and recursive uses a selection structure (if, if else or switch).
- Both iteration and recursive can execute infinitely-an infinite loop occurs with iteration if the loop continuation test become false and infinite recursion occurs id the recursion step doesn't reduce the problem in a manner that coverage on a base case.
- Recursion has disadvantage as well. It repeatedly invokes the mechanism, and consequently the overhead of method calls. This can be costly in both processor time and memory space. Each recursive call creates another copy of the method (actually, only the function's variables); this consumes considerable memory.
- **N.B**: Use recursion if:
    1. A recursive solution is natural and easy to understand
    2. A recursive solution doesn't result in excessive duplicate computation.
    3. the equivalent iterative solution is too complex and
    4. of course, when you are asked to use one in the exam!!