Chapter IV

FUNCTIONS

In this chapter we will cover:

1. Inline functions.
2. Functions with examples on:
   - 2.1 Summing an infinite series.
   - 2.2 Solving linear algebraic simultaneous equations using Gauss-Jordan Method.
3. Recursive functions.
4. Function overloading.
5. Passing Arguments to a function.
6. Pointers to functions.

4.1 Inline Functions

In Chapter I we stated that a function is a block of code that can be called from any location in a program. This helps reduce the size of the compiled code. If a function contains just a few statements then you can declare the function as **inline**. In declaring a function as inline you are suggesting to the compiler to substitute the function call with the compiled code of the function body. The compiler can ignore the request for inline expansion if the function contains too many statements. The advantage of using inline functions is that it saves on the overhead required in a regular call statement and hence improves on the speed of execution. The following program demonstrates usage of inline functions:

```
#include <iostream.h>
#include <ctype.h>

inline double square(double x)
  {
  return x*x;
  }

//---------------------------------------------------------------------------
int main()
{
 char c;
 double a;

 while(1)
  {
 cout << "Enter a number --> ";
 cin >> a;
```

```
 cout << "Its square is " << square(a) << endl;
 cout << "enter a character from the alphabet --> ";
 cin >> c;
 cout << (char)toupper(c) << endl;
 cout << "Continue ? (y/n)";
 cin >> c;
 c=(char)toupper(c);
 if(c == 'N') break;   //break out of the while loop
  }

return 0;
}
```

The above program uses the built in function " int **toupper**(char a)" which is prototyped in ctype.h. It returns an int and therefore casting to character is required.. The function returns an upper case if the argument is lower or upper case. You can also, if you wish, write your own inline function for lower to upper case conversion as follows:

```
inline int ToUpperCase(char c)
   {
   return ( c >= 'a' && c <= 'z' )? c - 'a' + 'A': c;
     }
```

Note that the function ToUpperCase uses the ternary operator "**? :**" which follows the syntax:

```
var = (Test)? expression1 : expression2;
```

If Test is true then the result of expression1 is assigned to var otherwise var is assigned the result of expression2.

4.2 Functions

Functions are central to C++. The function main() is where execution of a program begins. Function main() can call other functions such as built in types or functions you supply. Function calling is not just restricted to main but can be used by any other function to call other functions needed for execution.

Example

Develop a function program for fsin(x) using the infinite series:

$$\text{f sin}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots.$$

x is in radians.

Solution

Before we can develop a program for fsin(x) we need to develop an equation for generating the various terms of the infinite expression. To do so we will assume that the first term is T_1 the second term is T_2 and the i^{th} term is T_i. The following steps illustrates the approach used in developing an expression for T_i.

From the series we can write:

$$T_1 = x$$

$$T_2 = -\frac{x^3}{3\times 2} = (\frac{-x^2}{3\times 2})x$$

$$= (\frac{-x^2}{3\times 2})T_1$$

$$T_3 = \frac{x^5}{5\times 4\times 3!} = (\frac{-x^2}{5\times 4})(-\frac{x^3}{3!})$$

$$= (\frac{-x^2}{5\times 4})T_2$$

Hence one can deduce the following equation:

$$T_{i+1} = \left(\frac{-x^2}{(2i+1)2i}\right)T_i \qquad i \geq 1 \qquad (i = 1,2,3,..)$$

$$and \quad T_1 = x$$

The following is a program in which fsin(x) function is developed and tested to calculate:

$$y = 1.0 + 3.0 * \sin(30^0) + \frac{\pi}{3.0}$$

```
#include <iostream.h>
#include <conio.h>

#define pi 3.1415927

// protyping
  double fsin(double);
```

```
int main()
{
cout << (1.0 + 3.0*fsin(pi/6.0) + pi/3.0);
getch();
return 0;
}

double fsin(double x)
 {
 double sum, T1, T2;
 int i;

 T1=x;
 sum=T1;
 i=1;

 while(1)
  {
  T2=(-x*x)*T1/double((2*i+1)*2*i);
  i++;
  sum+=T2;
  if((T1-T2) <= 1.e-7) break;
  T1=T2;
  }
return sum;
}
```

If you recall from Chapter I (see Table II in Chapter I) sin(x) is a built in library function. However, on occasions you will run into functions that are not part of the library functions that you need in your particular application. The development of fsin(x) would serve you as a guideline for developing program functions of similar nature.

4.3 Solution of Linear Algebraic Equations using the Gauss-Jordan Method

The Gauss-Jordan method is a popular approach for solving linear equations. There are many other methods in the literature for solving linear algebraic equations, but in this book we will just develop the necessary algorithm and program code for the Gauss-Jordan method. This can serve you as guide for the development of your own library for solution of linear equations using other known methods. The Gauss Jordan method depends on two properties of linear equations:

1. Scaling one or more of any of the linear equations by a non-zero value has no effect on the solution of these equations.

2. Replacing any equation in the set by the addition or subtraction of that equation with any other equation in the set has no effect on the final solution.

To understand the Gauss-Jordan method and to develop an algorithm we will work with a numerical example. We will follow the steps of the Gauss-Jordan method to develop an algorithm for solving linear equations. Once an algorithm is developed the coding part of the development becomes straight forward as we will see.

Consider the following set of equations:

$$\begin{aligned} 3x_1 + 4x_2 + 6x_3 &= 9 \\ 7x_1 + 9x_2 + 10x_3 &= 2 \\ x_1 + 2x_2 + 5x_3 &= 7 \end{aligned}$$

Solution

The first step is to form the augmented matrix which contains just the constant coefficients plus the right hand side.

$$\begin{bmatrix} 3 & 4 & 6 & 9 \\ 7 & 9 & 10 & 2 \\ 1 & 2 & 5 & 7 \end{bmatrix}$$

The Gauss-Jordan method uses the two properties mentioned at the beginning of this section to reduce the augmented matrix to:

$$\begin{bmatrix} 1 & 0 & 0 & a_{14} \\ 0 & 1 & 0 & a_{24} \\ 0 & 0 & 1 & a_{34} \end{bmatrix}$$

The solution of the linear equations would then be:

$$\begin{aligned} x_1 &= a_{14} \\ x_2 &= a_{24} \\ x_3 &= a_{34} \end{aligned}$$

The Gauss Jordan method requires a number of general steps called passes. The number of passes is equal to the number of equations. Within each pass two steps are performed:

1. A normalization step to reduce a diagonal element to one.
2. An elimination step to reduce off diagonal elements in the same column as the normalized diagonal element to zeros.

As we work on the method we will assign variables and develop general equations to help us later in developing the algorithm. As we work on the development of the algorithm it is best to recall that the first index in any matrix coefficient represents the row number and the second index the column number, thus a_{23} represent the coefficient in the 2nd row and 3rd column.

Pass # 1 (j = pass number and j=1 for this pass)

(a) Normalization

In this step we normalize row # 1, that is:

$$\begin{bmatrix} 1 & \frac{4}{3} & 2 & 3 \\ 7 & 9 & 10 & 2 \\ 1 & 2 & 5 & 7 \end{bmatrix}$$

This step can be described with the assignment statement:

Row #1 = Row#1/ a_{11}

or in general: $\boldsymbol{R_j \Leftarrow R_j / a_{jj}}$

Where R_j is the j[th] row and a_{jj} is usually referred to as the pivot element.

(b) Elimination:

In order to eliminate a_{21} (=7) and a_{31} (=1) we subtract row # 1 after multiplying it with a_{21} from row # 2 and also subtract row # 3 after multiplying it by a_{31}. In notation form we can write:

$$R_2 \Leftarrow R_2 - R_1 \times a_{21}$$
$$R_3 \Leftarrow R_3 - R_1 \times a_{31}$$

Hence after elimination we get:

$$\begin{bmatrix} 1 & \frac{4}{3} & 2 & 3 \\ 0 & \frac{-1}{3} & -4 & -19 \\ 0 & \frac{2}{3} & 3 & 4 \end{bmatrix}$$

The elimination step can be summarized as follows:

$$\left[\begin{array}{l} \boldsymbol{i} = 1 \ \boldsymbol{to} \ \boldsymbol{N};\ (\boldsymbol{i} \neq \boldsymbol{j}) \\ \quad \boldsymbol{R}_i \Leftarrow \boldsymbol{R}_i - \boldsymbol{R}_j \times \boldsymbol{a}_{ij} \end{array}\right.$$

where $\boldsymbol{N}$ = number of equations.

The remaining passes will be used to verify the general equations derived for both the elimination and normalization steps.

Pass # 2 (j = 2)

(a) Normalization

$$\boldsymbol{R}_2 \Leftarrow \boldsymbol{R}_2 / \boldsymbol{a}_{22}$$

which results in

$$\begin{bmatrix} 1 & \frac{4}{3} & 2 & 3 \\ 0 & 1 & 12 & 57 \\ 0 & \frac{2}{3} & 3 & 4 \end{bmatrix}$$

Note that division by a_{jj} starts afer the (j+1) column (3rd column in this case). The value at (j,j) is then set to 1. Note also that elements in row # j prior to column j are zero and hence are not included in the division by the pivot element a_{jj}.

(b) Elimination

$$\boldsymbol{R}_1 \Leftarrow \boldsymbol{R}_1 - \boldsymbol{R}_2 \times \boldsymbol{a}_{12}$$
$$\boldsymbol{R}_3 \Leftarrow \boldsymbol{R}_3 - \boldsymbol{R}_2 \times \boldsymbol{a}_{32}$$

which results in:

$$\begin{bmatrix} 1 & 0 & -14 & -73 \\ 0 & 1 & 12 & 57 \\ 0 & 0 & -5 & -34 \end{bmatrix}$$

Note for the two rows in which elimination was carried out the process of elimination could start from the (j+1) column (3rd column in this case). After completion of this step we can set

$$a_{12} = 0$$
$$a_{32} = 0$$

The values prior to the j[th] column for each of these rows will not be effected since we are basically subtracting zeros.

Pass # 3 (j = 3)

(a) Normalization

$$\boldsymbol{R}_3 \Leftarrow \boldsymbol{R}_3 / \boldsymbol{a}_{33}$$

This results in:

$$\begin{bmatrix} 1 & 0 & -14 & -73 \\ 0 & 1 & 12 & 57 \\ 0 & 0 & 1 & 6.8 \end{bmatrix}$$

(b) Elimination

$$\boldsymbol{R}_1 \Leftarrow \boldsymbol{R}_1 - \boldsymbol{R}_3 \times \boldsymbol{a}_{13}$$
$$\boldsymbol{R}_2 \Leftarrow \boldsymbol{R}_2 - \boldsymbol{R}_3 \times \boldsymbol{a}_{23}$$

which leads to:

$$\begin{bmatrix} 1 & 0 & 0 & -73-(-14)\times 6.8 \\ 0 & 1 & 0 & 57-12\times 6.8 \\ 0 & 0 & 1 & 6.8 \end{bmatrix}$$

Therefore:

$$x_1 = -73+14\times 6.8$$
$$x_2 = 57-12\times 6.8$$
$$x_3 = 6.8$$

or $x_1 = 22.2, \quad x_2 = -24.6, \quad x_3 = 6.8$

If you are to examine the general equations for normalization and elimination derived in pass # 1 against the equations used in pass # 2 and # 3 you will be able verify their validity. Hence the Gauss-Jordan algorithm can be summarized in the following steps:

$$
\begin{bmatrix}
\boldsymbol{j} = 1,2,\ldots,\boldsymbol{N} \\
\text{//}\ \textbf{\textit{Normalization}} \\
\boldsymbol{R_j} \Leftarrow \boldsymbol{R_j} / \boldsymbol{a_{jj}} \\
\text{//}\ \textbf{\textit{Elimination}} \\
\begin{bmatrix}
\boldsymbol{k} = 1,2,3,\ldots,\boldsymbol{N};\ \boldsymbol{k} \neq \boldsymbol{j} \\
\boldsymbol{R_k} = \boldsymbol{R_k} - \boldsymbol{R_j} \times \boldsymbol{a_{kj}}
\end{bmatrix}
\end{bmatrix}
$$

The j[th] row in the normalization step can be written as follows:

$$
\underbrace{a_{j1} \quad a_{j2} \quad \ldots \quad a_{j,j-1}}_{\text{Note that these values have been reduced to zeros}} \; a_{jj} \quad a_{j,j+1} \quad \ldots \quad a_{j,n+1}
$$

Hence the normalization step can be expanded into:

$$
\begin{bmatrix}
\boldsymbol{i} = \boldsymbol{j} + 1,\ldots,\boldsymbol{N} + 1 \\
\boldsymbol{a_{ji}} = \boldsymbol{a_{ji}} / \boldsymbol{a_{jj}}
\end{bmatrix}
$$

$$
\boldsymbol{a_{jj}} = 1.0
$$

For the elimination step consider the k[th] and j[th] rows:

$$
\begin{array}{lccccccc}
R_k \rightarrow & a_{k1} & a_{k2} & \ldots & a_{kj} & a_{k,j+1} & \ldots & a_{k,N+1} \\
\vdots & & & & & & & \\
R_j \rightarrow & 0 & 0 & \ldots & 1 & a_{j,j+1} & \ldots & a_{j,N+1}
\end{array}
$$

Since the elements in the jth row prior to the j[th] column are all zeros we can expand the elimination step as follows:

$$\begin{cases} k = 1,2,3,\ldots,N; \quad k \neq j \\ \begin{cases} i = j+1, j+2,\ldots,N+1 \\ \quad a_{ki} = a_{ki} - a_{ji} \times a_{kj} \end{cases} \\ \qquad a_{kj} = 0 \end{cases}$$

The following program translates the above algorithm into code.

```
#include <iostream.h>
#include <conio.h>

// protyping
   void simq(double a[][4], double x[], int N);   //Gsuss-Jordan

int main()
{
double a[][4]= { { 3, 4, 6, 9 },
                 { 7, 9, 10, 2 },
                 { 1, 2, 5, 7 } };
double x[3];
int i, N=3;
simq(a,x,N);
for(i=0; i < N ; i++)
  cout << "x[ " << i << " ] = " << x[i] << endl;

getch();
return 0;
}

void simq(double a[][4], double x[], int N)
  {
int i,j,k;

// passes
for(j=0; j < N; j++)
  {
  //Normalization
  for ( i=j+1; i < (N+1) ; i++)
       a[j][i]/=a[j][j];
```

```
 a[j][j]=1.0;
 //Elimination
 for(k=0; k < N; k++)
   {
   if(k==j) continue; //go to end of loop
   for(i=j+1; i < (N+1); i++)
      a[k][i]-=a[j][i]*a[k][j];
   a[k][j]=0.0;
   }
  }
for(i=0; i < N; i++)
  x[i]=a[i][N];
}
```

The above program can obviously be made more general if we used pointers and dynamic memory allocation. We have left these out in our initial development stage of the program in order not to clutter the code.

Two questions that may come to mind regarding the previously described Gauss-Jordan method are:

1. What happens if a diagonal element we are using as pivot is zero?
2. What happens if there is no unique solution to the algebraic set?

To accommodate these possibilities the method will have to be modified as follows:

For j=1,2,3, ...,N ; N= number of equations

1. In the j^{th} column of the augmented matrix starting from the j^{th} row determine the largest absolute value and its row location. Denote the largest absolute value by BIG and its location by L.
2. Check if BIG < 10^{-7}. If true then no unique solution exists and exit the program, otherwise continue. (10^{-7} was chosen on the basis of single precision. We could not have chosen zero instead because the effect of truncation and round-off errors may result in zero being approximated to a very small number.)
3. Switch the L and j rows.
4. Normalization step carried out as before.
5. Elimination step carried out as before.

The above method is usually referred to as the Gauss-Jordan method with maximum pivoting.

Example

Solve the following set of linear equations using the Gauss-Jordan method with maximum pivoting:

$$\begin{aligned} 2x_1 + x_2 - 3x_3 &= -1 \\ -x_1 + 3x_2 + 2x_3 &= 12 \\ 3x_1 + x_2 - 3x_3 &= 0 \end{aligned}$$

Solution

Form the augmented matrix.

$$\begin{bmatrix} 2 & 1 & -3 & -1 \\ -1 & 3 & 2 & 12 \\ 3 & 1 & -3 & 0 \end{bmatrix}$$

Pass # 1 (j=1)

(a) BIG=3, L=3.

(b) Check BIG<10^{-7}? no. Therefore, continue.

(c) Switch 3rd and 1st rows

$$\begin{bmatrix} 3 & 1 & -3 & 0 \\ -1 & 3 & 2 & 12 \\ 2 & 1 & -3 & -1 \end{bmatrix}$$

(d) Normalization

$$\begin{bmatrix} 1 & 0.333 & -1 & 0 \\ -1 & 3 & 2 & 12 \\ 2 & 1 & -3 & -1 \end{bmatrix}$$

(e) Elimination

$$\begin{bmatrix} 1 & 0.333 & -1 & -0 \\ 0 & 3.333 & 1 & 12 \\ 0 & 0.333 & -1 & -1 \end{bmatrix}$$

Pass # 2 (j=2)

(a) BIG=3.333, L=2 (remember search starts from the j^{th} row in the j^{th} column)

(b) Check BIG<10^{-7}? no. Therefore, continue.

(c) L=j $\therefore$ no switching of rows.

(d) Normalization

$$\begin{bmatrix} 1 & 0.333 & -1 & 0 \\ 0 & 1 & 0.3 & 3.6 \\ 0 & 0.333 & -1 & -1 \end{bmatrix}$$

(e)Elimination

$$\begin{bmatrix} 1 & 0 & -1.1 & -1.2 \\ 0 & 1 & 0.3 & 3.6 \\ 0 & 0 & -1.1 & -2.2 \end{bmatrix}$$

Pass # 3 (j=3)

(a) BIG=1.1, L=3.

(b) Check BIG<10^{-7}? no. Therefore, continue.

(c) Normalization

$$\begin{bmatrix} 1 & 0 & -1.1 & -1.2 \\ 0 & 1 & 0.3 & 3.6 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

(d) Elimination

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

$\therefore \quad x_1 = 1$

$x_2 = 3$

$x_3 = 2$

Determining BIG and L

$$\text{jth column} \rightarrow \begin{pmatrix} a_{0j} & a_{1j} & \dots & a_{jj} & a_{j+1,j} & \dots & a_{N-1,j} \end{pmatrix}^T$$

$\uparrow$

Search for BIG starts from here onwards.

The following algorithm can be used in determining BIG and L

$$BIG = |a_{jj}| \quad // \text{ j = pass \#}$$
$$L = j$$
$$\begin{bmatrix} i = j+1, j+2, \ldots, N-1 \\ \quad if\ (BIG < |a_{ij}|) \\ \quad\quad \{ \\ \quad\quad BIG = |a_{ij}| \\ \quad\quad L = i \\ \quad\quad \} \end{bmatrix}$$

Switching the L and j rows

A temporary variable is required for the switching of the two rows as shown in the following algorithm.

$$\begin{bmatrix} \boldsymbol{k} = 1,2,\ldots,\boldsymbol{N} \\ \boldsymbol{Temp} = \boldsymbol{a_{LK}} \\ \boldsymbol{a_{LK}} = \boldsymbol{a_{jK}} \\ \boldsymbol{a_{jK}} = \boldsymbol{Temp} \end{bmatrix}$$

Exercise 4.1

Develop a function for solving a set of simultaneous equations using Gauss-Jordan method with maximum pivoting. Use the following program segments as guideline:

```
void main()
  {
double a[][4]={{2,1,-3,-1},
               {-1,3,2,12},
               {3,1,-3,0}};
double **p;
int i,j, N=3;
//Allocate dynamically storage for p
// to store a 3x4 matrix of type double
.
.
```

```
.
//Assign the values of a to p
.
.
.
//Call simq - a routine you will develop based on the
// Gauss-Jordan Method with maximum pivoting
        Simq(p,n)
// Print result
.
.
.
// Deallocate memory
.
.
.
}

int simq(p,n)     // program should return some indication if a solution exists.
  {
.
.
.
   }
```

4.4 Recursive Functions

Functions that call themselves are called recursive functions. In a recursive function a condition to stop the recursion should be included to prevent infinite recursion. Consider the following example for calculating the factorial of a number.

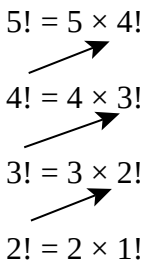
```
#include <iostream.h>
#include <conio.h>

unsigned long factorial(unsigned x);

//------------------------------------------------------------------------
int main()
{
cout << factorial(5);
getch();
return 0;
}
```

```
unsigned long factorial(unsigned x)
   {
 if(x>1)
  return (unsigned long)x*factorial(x-1);
 else
 return 1;
   }
```

In the above example when x=1 then the function factorial(1) returns a one to the calling program which is factorial(2). Three is then multiplied by 2! And the result is returned to the calling program which is factorial(3) and so on until the calling program is main() which then receives the final answer of 5 multiplied by factorial(4). This process is illustrated in the following diagram:



Factorial is used to illustrate recursive functions. Factorial can, however, be achieved through iteration without the use of recursion as shown.

```
unsigned long factorial(unsigned x)
 {
 unsigned long fact=x;
 while(x>1)
   {
   fact*=(x-1);
   x--;
    }
 return fact;
  }
```

Any problem that can broken down into lesser versions of the same structure as the original solution, i.e. factorial(n-1) has the same solution version as factorial(n), is a candidate for recursion.

4.5 Default Arguments

Arguments in functions can be given default values. Omitting an argument that has a default value results in the automatic replacement of that argument by its default value. There are, however,

a set of rules to follow:

- ! If you assign a default value to an argument, you must do the same for all subsequent arguments in the function.
- ! In the calling routine an argument that has no default value must be assigned a value.
- ! You can when calling the function omit arguments with default values. If you omit one argument you must do so for all subsequent arguments.

The following example illustrates the usage of default arguments in functions.

Example. Develop a program function for e^x where

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Following the same approach used in the example of sec.4.2 we obtain the iterative function:

$$T_{i+1} = T_i \times \frac{x}{i} \quad \text{where } T_1 = 1$$

Using the iterative function we can develop the following program.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

double _fexp(double x, double eps=1.e-15); //prototype with
                // one default argument

//------------------------------------------------------------------------
int main()
{
cout << _fexp(2) << endl
    << _fexp(2, 1.e-3) << endl
    << exp(2) << endl; // exp() is a library function

getch();
return 1;
   }

//Function _fexp(x) = 1 + x + (x^2)/2! + (x^3)/3!+...

double _fexp(double x, double eps)
 {
 double t, sum;
```

```
  int i;

t=double(1.0);
sum=double(1.0);
i=1;

while(1)
  {
  t=t*x/double(i);
  sum+=t;
  i++;
  if(fabs(t) < eps) break;
   }
 return sum;
 }
```

You can also redevelop the above program by comparing terms instead of the absolute of a term as follows:

```
//Function _fexp(x) = 1 + x + (x^2)/2! + (x^3)/3!+...

double _fexp(double x, double eps)
  {
  double t1,t2,sum;
  int i;

t1=double(1);
sum=t1;
i=1;

while(1)
  {
  t2=(t1*x)/double(i);
  sum+=t2;
  i++;
  if((t1>t2) && ((t1-t2) <= eps)) break;
  t1=t2;
   }
 return sum;
 }
```

Note that in this series the terms can increase sequentially at the beginning before they begin to decay and hence the statement `if((t1>t2) && ((t1-t2) <= eps)) break;` was used as a stopping criteria.

4.6 Function Overloading

C++ allows you to declare multiple functions with the same name but different parameter list. The following example illustrates this feature of the language.

Example. Develop a function _fabs(...) that accepts either a complex, double or integer value.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

inline double sqr(double x)
  {
  return x*x;
  }

struct _Complex{
    double Real;
    double Imag;
    };

double _fabs(_Complex a)
 {
 return sqrt(sqr(a.Real)+sqr(a.Imag));
 }

double _fabs(double a)
 {
 return (a>double(0))? a:-a;
 }

 int _fabs(int a)
 {
 return (a>int(0))? a: -a;
 }

//---------------------------------------------------------------------------
int main()
{
_Complex a= {3.0, 4.0};
double b=-8.95;
int c=3;
```

```
cout << _fabs(a) << endl
    << _fabs(b) << endl
    << _fabs(c) << endl;

getch();
return 1;
   }
```

4.7 Passing Arguments to a Function

4.7.1 Passing Arrays as Arguments

The following program demonstrates the passing of arrays as arguments to a function.

```
#include <iostream.h>
#include <conio.h>

double *add_vect(double a[], double b[], int);
            // prototype
double *c;  //Global variable

//---------------------------------------------------------------------------

int main()
{
double a[]={3,4,5};
double b[]={6,7,8};

c=new double[3];
c=add_vect(a,b,3);

for(int i=0; i<3; i++)
   cout << c[i] << "  ";
cout << endl;
getch();
delete []c;
return 1;
   }


double *add_vect(double a[], double b[], int n)
 {
 for(int i=0; i<n; i++)
   c[i]=a[i]+b[i];
```

```
return c;
  }
```

4.7.2 Passing Simple Variables

Arguments can be passed to a function:

- by value
- by reference using pointers
- by reference using aliases.

The following illustrates these three possibilities.

```
#include <iostream.h>
#include <conio.h>

void f1(int a, int b); //by value
void f2(int *a, int *b); //by reference using pointers
void f3(int &a, int &b); //by reference using aliases

//---------------------------------------------------------------------------

int main()
{
int i=1, j=2;

f1(i,j);
cout << i << "  " << j
    << " <-result of passing by value"
    << endl;

i=1; j=2;
f2(&i,&j);
cout << i << "  " << j
    << " <-result of passing by reference using pointers"
    << endl;

i=1; j=2;
f3(i,j);
cout << i << "  " << j
    << " <-result of passing by reference using aliases"
    << endl;
```

```
getch();
return 1;
   }

void f1(int a, int b)
 {
 a++;
 b++;
 }

void f2(int *a, int *b)
  {
 (*a)++;
 (*b)++;
  }

void f3(int &a, int &b)
 {
 a++;
 b++;
 }
```

The printout from the above program is as follows:

```
1  2 <-result of passing by value
2  3 <-result of passing by reference using pointers
2  3 <-result of passing by reference using aliases
```

The following can be deduced:

- Variables passed by value can be used by the function but any changes to these variables are not reflected in the calling function. The function receives its own private copy of a variable when it is passed by value.
- Any change to a variable passed by reference using either pointer or alias will be reflected in the calling function. The function has direct access to the variable ans does not receive a private copy.

In passing arrays the function does not receive a private copy of the array and any changes to the array are reflected in the calling program. As a matter fact the address of the array is the name of the array. In other words if an array is declared as float vector[3] in a program then vector = &vector[0]. Same thing applies to higher order arrays such as matrices.

4.7.3 Passing Structures

In the following example two functions are developed:

- A function that accepts as arguments pointers to a structure and returns a pointer to a structure.
- A function that accepts by reference structures in its argument list and returns a structure.

```
 #include <iostream.h>
#include <conio.h>

struct TPoint {
   float x;
   float y;
   };

TPoint *getMidPoint(TPoint *, TPoint *);
TPoint getCentrePoint(TPoint &, TPoint &);

//---------------------------------------------------------------------------

int main()
{
TPoint p1={1,1};
TPoint p2={2,2};

TPoint *p3,p4;

p3 = new TPoint;

p3 = getMidPoint(&p1, &p2);
p4 = getCentrePoint(p1,p2);

cout << p3->x << "  " << p3->y << endl;
cout << p4.x  << "  " << p4.y  << endl;

getch();
 return 1;
  }

TPoint *getMidPoint(TPoint *p1, TPoint *p2)
 {
 TPoint *p3;
 p3 = new TPoint;

 p3->x = (p1->x + p2->x)/2.0;
```

```
 p3->y = (p1->y + p2->y)/2.0;

 return p3;
 }

TPoint getCentrePoint(TPoint &p1, TPoint &p2)
 {
 TPoint p3;
 p3.x = (p1.x + p2.x)/2.0;
 p3.y = (p1.y + p2.y)/2.0;

 return p3;
 }
```

It is always best to pass structures by reference rather than value. In doing so the function does not receive a private copy of the structures and can work directly on the original data. The reason for passing structures by reference is to save memory during execution of the program. If a large structure was used and passed to a function by value, the structure will have to be copied to another segment in memory to provide a private copy to the function.

4.7.2 Pointers to functions

Since a function is a set of code located in memory its address can be utilized to access that code. One possible use of defining a pointer to a function is shown in the following program. Other uses will be demonstrated when we get into classes.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

double sqr(double x)
 {
 return x*x;
 }

double cube(double x)
 {
 return x*sqr(x);
 }

double quad(double x)
 {
 return sqr(x)*sqr(x);
 }
```

```
//---------------------------------------------------------------------------

int main()
{
double x=2;
double (*f[4])(double); //Pointer to a function

f[0]=sqr;
f[1]=cube;
f[2]=quad;

for(int i=0; i<3; i++)
  cout << (*f[i])(x) << endl;

getch();

 return 1;
   }
```

In the above program an array of pointers to a function is used for pointing at three different functions. This allowed the use of a loop to access each function in turn. A function can also be defined at the outset as a pointer to a function thus allowing its name to be changed by the calling program. We will make use of that in classes.

Problems

6. Develop **inline** functions for
   a. Converting an upper case character to an lower case character.
   b. Obtaining the absolute value of an int, float, double or Complex (use function overloading).
7. Develop a function to calculate π using the infinite series:

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots=\frac{\pi}{4}$$

   allow the user to decide on the number of significant digits up to the maximum allowed by double.
8. Develop a functions for calculating:

   a. $$a^x = e^{x\ln a} = 1 + x\ln a + \frac{(x\ln a)^2}{2!} + \frac{(x\ln a)^3}{3!} + \ldots.$$

$$-\infty < x < \infty$$

b. $$\ln x = 2\left\{\left(\frac{x-1}{x+1}\right) + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \cdots\right\} \qquad x > 0$$

c. $$\tan^{-1} x = \begin{cases} x - \dfrac{x^3}{3} + \dfrac{x^5}{5} - \dfrac{x^7}{7} + \cdots & |x| < 1 \\ \pm\dfrac{\pi}{2} - \dfrac{1}{x} + \dfrac{1}{3x^3} - \dfrac{1}{5x^5} + \cdots & [+ \text{ if } x \geq 1, \ - \text{ if } x \leq -1] \end{cases}$$

9. Develop a C++ function for solving a set of linear algebraic equations using the Gauss-Jordan algorithm with maximum pivoting. Test your function on the following equations:

a. $$\begin{bmatrix} 1 & 2 & 0 & 1 \\ 3 & 4 & -1 & 1 \\ 0 & 1 & 5 & 2 \end{bmatrix}$$

b. $$\begin{bmatrix} 4 & 1 & 5 & 5 \\ 1 & -4 & 7 & 2 \\ 5 & 7 & -1 & 6 \end{bmatrix}$$

10. Calculate solutions to the two set of linear algebraic equations given in problem 4 by hand calculations using the Gauss-Jordan method with maximum pivoting.

11. Develop a C++ function for solving a set of linear algebraic equations with complex numbers. Test your function on the following equations:

$$\begin{bmatrix} 3+i & 1-2i & 2+2i & 4+6i \\ -1+2i & 1+i & 5-i & 2+i \\ 2+3i & -2+7i & 1+3i & 3 \end{bmatrix}$$

12. The exponent of a matrix is given by the infinite series:

$$e^A = I + A + \frac{1}{2!}A^2 + \frac{1}{3!}A^3 + \frac{1}{4!}A^4 + \cdots$$

Develop a C++ function to calculate the above series.