**RASHTREEYA SIKSHANA SAMAITHI TRUST**

# R.V. COLLEGE OF ENGINEERING

**Bengaluru - 560059**

(**Autonomous Institution Affiliated to VTU, BELAGAVI)**

## Department of Master of Computer Applications



**DATA STRUCTURE USING C 18MCA13**

### LITERATURE SERVEY

**Submitted To: - Submitted By: -**

**Dr. Andhe Dharani Mohammed Sharief.**

**Mr. Krishnan R. Manoj Khotakar M.**

*In partial fulfilment for the award of degree*

*Of*

*Post-Graduation*

*In*

*MASTER OF COMPUTER APPLICATION*

*2018-2019*

Header Files

Mathematical Problems functions in a Header File

Mohammed Sharief & Manoj Khotakar

### Analysis and Review of Mathematical Problems Algorithms

**Header files**

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

The application programming interface (API) of the C standard library is declared in a number of header files. Each header file contains one or more function declarations, data type definitions, and macros.

Header Files are included in the beginning of the program. It is a predefined code and we have to simply include the file. Thus, writing less is an advantage. It has .h extension. It contains C declarations and macro definition.

header.h new.c

main.c

#define

…

V

oid

…

…

I

nt

…..

#

include<>

#

include

header.h”

”

m

ain(

){

}

#

include<>

#

include

“header.h

”

m

ain(

){

}

**What should be in the header files for a complex project?**

In C, the contents of a module consist of structure type (struct) declarations, global variables, and functions. The functions themselves are normally defined in a source file (a “.c” file). Except for the main module, each source (.c) file has a header file (a “.h” file) associated with it that provides the declarations needed by other modules to make use of this module.

The idea is that other modules can access the functionality in module X simply by #include "**X.h**" for the header file, and the linker will do the rest. The code in **X.c** needs to be compiled only the first time or if it is changed; the rest of the time, the linker will link X’s code into the final executable without needing to recompile it, which enables the Unix make utility and IDEs to work very efficiently.

A well-organized C program has a good choice of modules, and properly constructed header files that make it easy to understand and access the functionality in a module. They also help ensure that the program is using the same declarations and definitions of all of the program components. This is important because compilers and linkers need help in enforcing the One Definition Rule.

Furthermore, well-designed header files reduce the need to recompile the source files for components whenever changes to other components are made. The trick is reduce the amount of “coupling” between components by minimizing the number of header files that a module’s header file itself #includes. On very large projects, minimizing coupling can make a huge difference in “build time” as well as simplifying the code organization and debugging.

**MATHEMATICAL PROBLEMS:**

There are many **Mathematical Problems** that can be solved using **C Programming Language.**

1. **Polynomial Equations**

What is Polynomial?

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent.  
**Example:** P(x) = 4x3+6x2+7x+9

A polynomial may be represented using array or structure. A structure may be defined such that it contains two parts – one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

Struct polynomial{

int coefficient;

int exponent;

};

The basic idea of polynomial addition is to add coefficient parts of the polynomials having same exponent.

**Algorithm:**

AddPoly(Struct Poly p1[10],Struct Poly p2[10],int t1,int t2,Struct Poly p3[10])

1.) [Initialize segment variables]

[Initialize Counter] Set i=0,j=0,k=0

2.) Repeat step 3 while i<t1 and j<t2

3.) If p1[i].expo=p2[j].expo, then

p3[i].coeff=p1[i].coeff+p2[i].coeff

p3[k].expo=p1[i].expo

[Increase counter] Set i=i+1,j=j+1,k=k+1

else if p1[i].expo > p2[j].expo, then

p3[k].coeff=p1[i].coeff

p3[k].expo=p1[i].expo

[Increase counter] Set i=i+1,k=k+1

else

p3[k].coeff=p2[j].coeff

p3[k].expo=p2[j].expo

Set j=j+1,k=k+1

[End of If]

[End of loop]

4.) Repeat while i<t1

p3[k].coeff=p1[i].coeff

p3[k].expo=p1[i].expo

Set i=i+1,k=k+1

[End of loop]

5.) Repeat while j<t2

p3[k].coeff=p2[j].coeff

p3[k].expo=p2[j].expo

Set j=j+1,k=k+1

[End of loop]

6.) Return k

7.) Exit

### C program for Polynomial Addition Using Array of Structure:

/\* program for addition of two polynomials

 \* polynomial are stored using structure

 \* and program uses array of structure

 \*/

 #include<stdio.h>

/\* declare structure for polynomial \*/

**struct** poly

{

**int** coeff;

**int** expo;

};

/\* declare three arrays p1, p2, p3 of type structure poly.

 \* each polynomial can have maximum of ten terms

 \* addition result of p1 and p2 is stored in p3 \*/

**struct** poly p1[10],p2[10],p3[10];

/\* function prototypes \*/

**int** readPoly(**struct** poly []);

**int** addPoly(**struct** poly [],**struct** poly [],**int** ,**int** ,**struct** poly []);

**void** displayPoly( **struct** poly [],**int** terms);

**int** main()

{

**int** t1,t2,t3;

/\* read and display first polynomial \*/

t1=readPoly(p1);

printf(" \n First polynomial : ");

displayPoly(p1,t1);

/\* read and display second polynomial \*/

t2=readPoly(p2);

printf(" \n Second polynomial : ");

displayPoly(p2,t2);

/\* add two polynomials and display resultant polynomial \*/

t3=addPoly(p1,p2,t1,t2,p3);

printf(" \n\n Resultant polynomial after addition : ");

displayPoly(p3,t3);

printf("\n");

**return** 0;

}

**int** readPoly(**struct** poly p[10])

{

**int** t1,i;

printf("\n\n Enter the total number of terms in the polynomial:");

scanf("%d",&t1);

printf("\n Enter the COEFFICIENT and EXPONENT in DESCENDING ORDER\n");

**for**(i=0;i<t1;i++)

{

printf(" Enter the Coefficient(%d): ",i+1);

scanf("%d",&p[i].coeff);

printf(" Enter the exponent(%d): ",i+1);

scanf("%d",&p[i].expo); /\* only statement in loop \*/

}

**return**(t1);

}

**int** addPoly(**struct** poly p1[10],**struct** poly p2[10],**int** t1,**int** t2,**struct** poly p3[10])

{

**int** i,j,k;

i=0;

j=0;

k=0;

**while**(i<t1 && j<t2)

{

**if**(p1[i].expo==p2[j].expo)

{

p3[k].coeff=p1[i].coeff + p2[j].coeff;

p3[k].expo=p1[i].expo;

i++;

j++;

k++;

}

**else** **if**(p1[i].expo>p2[j].expo)

{

p3[k].coeff=p1[i].coeff;

p3[k].expo=p1[i].expo;

i++;

k++;

}

**else**

{

p3[k].coeff=p2[j].coeff;

p3[k].expo=p2[j].expo;

j++;

k++;

}

}

/\* for rest over terms of polynomial 1 \*/

**while**(i<t1)

{

p3[k].coeff=p1[i].coeff;

p3[k].expo=p1[i].expo;

i++;

k++;

}

/\* for rest over terms of polynomial 2 \*/

**while**(j<t2)

{

p3[k].coeff=p2[j].coeff;

p3[k].expo=p2[j].expo;

j++;

k++;

}

**return**(k); /\* k is number of terms in resultant polynomial\*/

}

**void** displayPoly(**struct** poly p[10],**int** term)

{

**int** k;

**for**(k=0;k<term-1;k++)

printf("%d(x^%d)+",p[k].coeff,p[k].expo);

printf("%d(x^%d)",p[term-1].coeff,p[term-1].expo);

}

**Output:**

Enter the total number of terms in the polynomial:4

Enter the COEFFICIENT and EXPONENT in DESCENDING ORDER

Enter the Coefficient(1): 3

Enter the exponent(1): 4

Enter the Coefficient(2): 7

Enter the exponent(2): 3

Enter the Coefficient(3): 5

Enter the exponent(3): 1

Enter the Coefficient(4): 8

Enter the exponent(4): 0

First polynomial : 3(x^4)+7(x^3)+5(x^1)+8(x^0)

Enter the total number of terms in the polynomial:5

Enter the COEFFICIENT and EXPONENT in DESCENDING ORDER

Enter the Coefficient(1): 7

Enter the exponent(1): 5

Enter the Coefficient(2): 6

Enter the exponent(2): 4

Enter the Coefficient(3): 8

Enter the exponent(3): 2

Enter the Coefficient(4): 9

Enter the exponent(4): 1

Enter the Coefficient(5): 2

Enter the exponent(5): 0

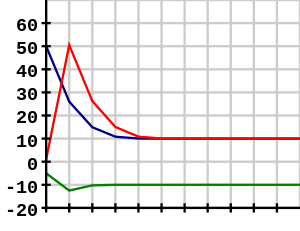
Second polynomial : 7(x^5)+6(x^4)+8(x^2)+9(x^1)+2(x^0)

Resultant polynomial after addition : 7(x^5)+9(x^4)+7(x^3)+8(x^2)+14(x^1)+10(x^0)

### Babylonian method for square root

## Babylonian method:

*"Heron's method" redirects here. For the formula used to find the area of a triangle, see* [*Heron's formula*](https://en.wikipedia.org/wiki/Heron's_formula)*.*



Graph charting the use of the Babylonian method for approximating the square root of 100 (10) using starting values *x*0 = 50, *x*0 = 1, and *x*0 = −5. Note that using a negative starting value yields the negative root.

Perhaps the first [algorithm](https://en.wikipedia.org/wiki/Algorithm) used for approximating is known as the **Babylonian method**, despite there being no direct evidence, beyond informed conjecture, that the eponymous [Babylonian mathematicians](https://en.wikipedia.org/wiki/Babylonian_mathematics) employed exactly this method.

The method is also known as **Heron's method**, after the first-century Greek mathematician [Hero of Alexandria](https://en.wikipedia.org/wiki/Hero_of_Alexandria) who gave the first explicit description of the method in his [AD 60](https://en.wikipedia.org/wiki/AD_60) work *[Metrica](https://en.wikipedia.org/wiki/Hero_of_Alexandria" \l "Bibliography)*.

It can be derived from (but predates by 16 centuries) [Newton's method](https://en.wikipedia.org/wiki/Newton's_method). The basic idea is that if *x* is an overestimate to the square root of a non-negative real number *S* then *Sx* will be an underestimate, or vice versa, and so the average of these two numbers may reasonably be expected to provide a better approximation (though the formal proof of that assertion depends on the [inequality of arithmetic and geometric means](https://en.wikipedia.org/wiki/Inequality_of_arithmetic_and_geometric_means) that shows this average is always an overestimate of the square root, as noted in the article on [square roots](https://en.wikipedia.org/wiki/Square_root" \l "Geometric_construction_of_the_square_root), thus assuring convergence).

/

More precisely, if *x i*s our initial guess of and *e* is the error in our estimate such that*S*= (*x*+*e*)2, then we can expand the binomial and solve for

since.

Therefore, we can compensate for the error and update our old estimate as

Since the computed error was not exact, this becomes our next best guess. The process of updating is iterated until desired accuracy is obtained. This is a [quadratically convergent](https://en.wikipedia.org/wiki/Quadratic_convergence) algorithm, which means that the number of correct digits of the approximation roughly doubles with each iteration. It proceeds as follows:

1. Begin with an arbitrary positive starting value *x*0 (the closer to the actual square root of *S*, the better).
2. Let *xn* + 1 be the average of *xn* and *Sxn* (using the [arithmetic mean](https://en.wikipedia.org/wiki/Arithmetic_mean) to approximate the [geometric mean](https://en.wikipedia.org/wiki/Geometric_mean)).

/

1. Repeat step 2 until the desired accuracy is achieved.

It can also be represented as:

This algorithm works equally well in the [*p*-adic numbers](https://en.wikipedia.org/wiki/P-adic_number), but cannot be used to identify real square roots with *p*-adic square roots; one can, for example, construct a sequence of rational numbers by this method that converges to +3 in the reals, but to −3 in the 2-adics.

### Example

To calculate √*S*, where *S* = 125348, to six significant figures, use the rough estimation method above to get

Therefore, √125348 ≈ 354.045.

Algorithm:

This method can be derived from (but predates) Newton–Raphson method.

1 Start with an arbitrary positive start value x (the closer to the

root, the better).

2 Initialize y = 1.

3. Do following until desired approximation is achieved.

a) Get the next approximation for root using average of x and y

b) Set y = n/x

Example:

n = 4 /\*n itself is used for initial approximation\*/

Initialize x = 4, y = 1

Next Approximation x = (x + y)/2 (= 2.500000),

y = n/x (=1.600000)

Next Approximation x = 2.050000,

y = 1.951220

Next Approximation x = 2.000610,

y = 1.999390

Next Approximation x = 2.000000,

y = 2.000000

Terminate as (x - y) > e now.

If we are sure that n is a perfect square, then we can use following method. The method can go in infinite loop for non-perfect-square numbers. For example, for 3 the below while loop will never terminate.