Brief Review of ‘C’:

Assignment Operator

```
a = 4;
b = a * a ;
c = b + a;
```

‘C’ compiler assigns a separate memory location to every new variable on the left side of the assignment operator.

if and if…else

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

Statement 2 will be executed anyway.

```
if (condition expression)
    statement1;
 else
    statement2;
statement 3;
```

Grades for 220 students of section 1 of COP3223 are available for an exam. Students are to be grouped as per following scheme:

- *group 1 : 85 and above*
- *group 2 : 70 to 84*
- *group 3 : 50 to 69*
- *group 4 : below 50*

It is desired to find out the number of students in each group.

```
#include<stdio.h>
int main (  ) {
    int grade, group1, group2, group3,
group4;
    group1= group2 = group3 = group4 = 0;

    for (jj = 1;          jj <=220;     jj++)
    {
        printf (“\n enter grade:”);
        scanf (“%d”, &grade);
        if (grade >= 85 )
                group1++ ;
        else  if ( grade >69  )
            group2++ ;
        else if ( grade >49 )
            group3++ ;
        else
            group4++;
    }

    printf (“\n group1= %d, group2= %d “,group1, group2);
    printf (“\n group3= %d, group4= %d “,group3, group4);
    return 0;
}
```

Operator precedence:

When an expression contains number of operators, they are processed in a particular order

```
!    –  (unary operators)  e.g. – 45

*       /   %

+        –

<       <=      >=  >

==  !=

&&

| |

= (assignment operator)
```

if(a < b+c   && c==d || a > e)

switch statement

Days in a month

```
int month, days;

switch (month){
 case 4 :
 case  6 :
 case 9 :
 case 11 :
        days = 30;
        break ;

 case 2 :
      days = 28;
      break ;

  default :
      days = 31;
}
```

while and do…while

Use a while loop to print out the square of the numbers entered by the user. It should stop when the user enters – 999.

```
aa = 1;
while (aa  !=  – 999)
{
    printf ("\n new value= ");
     scanf ("%d",&aa );
    printf ("sq of %d is  %d", aa , aa*aa);
}
```

```
do
{
    printf ("\nnew value= ");
     scanf ("%d",&aa);
    printf ("square of %d is  %d", aa,
    aa*aa);
}while (  aa  !=   – 999)
```

alternate version:

```
 while (1)
{
        printf ("\nnew value= ");
         scanf ("%d",&aa);

        if(  aa  ==   – 999)
                break;
        printf ("sq of %d is  %d", aa, aa*aa);
}
```

Function calls:

```
• # include  <stdio.h>
•  int max ( int,  int) ;
•  int main ( )  {
•         int  a, b,  m ;
•         scanf ( " %d %d ",  &a, &b );
•         m  =  max( a, b ) ;
•         printf ( "max of %d and %d is %d ",a, b, m );
• }
• int  max ( int n1, int n2  )
•  {
•          if  ( n1 > n2)         return  n1;
•          else                    return n2;
   }
```

- *Formal parameters*: Parameters listed in the function definition (e.g. n1, n2 in example)

- *Actual Parameters*: Values being passed on by the calling function (e.g. Main passing on values of a,b)

- Number of formal and actual parameters must match

- The order of parameters must match (e.g. a to n1 and b to n2)

- The indicated type for each parameter must match the type in the function definition

Do the actual parameter values get changed after the function is called?

```
#include <stdio.h>
int func1(int x, int y);
int main(void) {
  int mm, k = 10, j =6;
  mm = func1(k,     j );
  printf("\nIn the main program”);
  printf("mm = %d k=%d j=%d\n",mm, k, j);
  return 0;
}

int func1(int x, int y) {
     int ss;
     ss = 2 * x + y;
     x = x + 10;
     y = y * 2;

  printf("In func1: x=%d y=%d\n", x,y);
  return ss;
}
```

output:

```
In func1: x=20 y=12
In the main program
mm = 26 k=10 j=6
```

This is known as parameter *passing by value*. The parameters x and y in the called function, correspond to the parameters k and j of the main function.

They get their values from the main function,
Their values get changed in the called function,
BUT the change DOES NOT AFFECT the original values of k and j inside the main function.

If we wanted to change the values of k and j in the main program, then we need to pass the *parameters by reference*, that is we have to send the addresses of the variables, instead of the variables themselves. Look at the following program.

```
#include <stdio.h>
int func1(int x, int y);
int main(void) {
  int mm, k = 10, j =6;
  mm = func1(  &k,     &j );
  printf("\n In the main program”);
  printf("\n mm = %d k=%d j=%d\n",mm, k, j);
  return 0;
}

int func1(int *x, int *y) {
     int ss;
     ss =  *x  +   *y;
     *x =  *x + 10;
     *y =  *y + 2;

  printf("In func1: x=%d y=%d\n", *x,*y);
  return ss;
}
```

output:

```
In func1: x=20 y=8
In the main program
mm = 16 k=20 j=8
```

<u>**Library Functions:**</u>

<u>**Library Functions:**</u>

C compiler provides a number of library functions. A nice collection of most commonly used functions can be found in http://www.cs.ucf.edu/courses/cop3502/spr04/recitation/clibs.doc

You can also create your own libraries and use them in various client programs. The access is provided through interfaces.

An **interface** is the boundary between the implementation of the library and the client programs. The purpose of the interface is to provide each client the relevant information needed to use the library, without showing the detailed implementation.

The interface is provided by a header file of the same name that implements the library functions with extension **.h**. Suppose all the library functions are included in a file **libraries.c**. Then create another file called **libraries.h**. which just contain the corresponding functions prototypes.

To enable a client program to use the libraries, you have to just include **libraries.h** in the client program. Also make sure that you have included it in **libraries.c**.

ARRAYS:

Array size must be declared in the beginning of the program

```
char names[200];
 int grades[200];
```

Consider grades obtained by 80 students in 3 tests.

Test 1	Test 2	Test3
84	76	70
67	60	82
78	56	75
91	85	90
62	55	50
…	….	….
…	….	….

Let us say we want to find
i) average performance of each student in the 3 tests
ii) mean grade for each test

Let us first store the grades in a two dimensional array

```
int grades[80][3];
double perform[80],
double mean[3];
```

The grades can be entered from the keyboard row-wise using

```
for ( j = 0;   j  < 80 ;  j++ )
     scanf ( “ %d %d %d “,&grades [j][1],
       &grades [j][2],    &grades[j][3] );
```

To find the performance of each student, the outer loop variable refers to each student

```
• for ( j = 0 ; j < 80 ; j++ )
• {
•     sum= 0;
•     for ( k = 0 ; k < 3 ; k ++ )
•          sum = sum + grades [ j ] [ k ] ;
•     perform[j] = sum / 3;

•  printf (“student [%d] = %f \n”,  j+1,
  perform[j] )
•  }
```

To find the mean grade for each test, the outer variable refers to the individual tests

```
• for ( k = 0 ; k < 3 ; k++ )
• {
•     sum = 0;
•     for ( j = 0 ; j < 80 ;    j ++ )
             sum =sum + grades [j][k] ;
•     mean[k] = sum / 80 ;
•     printf (“test[%d] = %f \n”,  k+1,
            mean[k] ) ;
•  }
```

Passing arrays to functions:

Consider an event , where a number of judges award scores and it is desired to find the mean score. In the main program we define the array of scores as

```
#define nJudges 15
……………
…………..

double scores[nJudges];
double average;
```

Assume a function reads the scores given by various judges and stores in the array *scores[nJudges]*. Then the following call can be made from the main function

```
average = mean(scores, nJudges);
```

The function to compute the mean can take the following form:

```
double mean ( double arrayd[], int n)
{
    int j;
    double total;
    total  = 0;
    for (j =0; j<n; j++) {
        total + = arrayd[j];
    }
    return ( total/ n);
}
```

Pointers

Let us consider the declarations:

```
int x, y;
int *p1 ,*p2;
```

This will cause 4 memory locations to be allocated. Let us also assign some values to x and y:

x = - 42;
y = 163;

The memory map may look like this.

1000	-42	x
1004	163	y
1008		p1
1012		p2

Let us assign p1 and p2 to point to addresses of x and y:

```
p1  = &x;
p2= &y;
```

1000	-42	x
1004	163	y
1008	1000	p1
1012	1004	p2

*p1 is just another name for x after the above declaration. It can be assigned a different value if desired. Let us assign

```
*p1 = 56;
```

This will change whatever is stored in location p1.

1000	56	x
1004	163	y
1008	1000	p1
1012	1004	p2

p1 still remains 1000, as only contents have been changed. What is the value of x now?

It is possible to assign new values to the pointer variables. Let

```
p1 = p2;
```

What is *p1 now? It is 163.

1000	56	x
1004	163	y
1008	1004	p1
1012	1004	p2

Consider again the memory map

1000	56	x
1004	163	y
1008	1000	p1
1012	1004	p2

But this time consider the assignment

```
 *p1 = *p2;
```

Now p1 will point to what was being pointed to by p2.

1000	163	x
1004	163	y
1008	1000	p1
1012	1004	p2

Passing arrays to functions using pointers

Consider an array variable list containing 3 items.

```
double list [3];
double *p;
```

1000		list[0]
1008		list[1]
1016		list[2]
1024		p

Now assign the following values to the elements of the array

```
list[0] = 1.0;
list[1] = 1.1;
list[2] = 1.2;

p = &list[0];
```

Once p has been assigned the address of the first element of the array list, other elements of the array can be accessed using p.

Thus p+1 will correspond to 1008, the address of the element with index 1, i.e. list[1].

p+2 will correspond to 1016, the address of the element with index 2, i.e. list[2].

Suppose, instead we assign

```
p = &list[1];
```

then p+1 will correspond to address of list[2],
and p – 1 will correspond to address of element list [0].

p+k will correspond to list[k+1], if it exists.

If you are writing a for loop to go through the list, then you can do automatic incrementing using

```
*p++
```

This will first dereference the pointer p, and return the object in the list it is pointing to.
Then, p will get incremented, so that on next “for-loop” call, it will be pointing to the next element in the list.

Arrays can be passed to functions using pointers.
Let us first look at an example of passing an “array” parameter.

Let us say the function sum_array is being called from the main function through

```
sum = sum_array(list, 5) ;
```

and the function is defined as follows:

```
int sum_array( int arrayd[],  int n){
    int j, sum;
    sum = 0;
    for (j=0; j<n ; j++) {
        sum += arrayd[j];
    }
    return(sum);
}
```

The alternative scheme involving pointers may have a function call

```
sum = sum_array_P( list, n);
```

with the function defined as

```
int sum_array_P(int *ip, int n)
{
    int j,  sum;
    sum = 0;
    for (j=0; j<n ; j++)  {
        sum += *ip++;
    }
    return (sum);
}
```

A pointer can be initialized by setting it to name of the array. Thus after the assignment

```
pp = arrayd;
```

the pointer pp would point to the first location of the array arrayd.

Structures

Items of different data types can be clubbed together inside a structure.

```
int main ( ) {
  int j;
  struct person {
    int g1;
    int g2;
    char name[30];
    double average;
    };
```

Once the structure is defined, new variables of this data type can be declared. Let us declare a structure variable to hold information for 100 persons.

```
struct person   ss[100];
```

To access the components of a structure, the dot operator can be used.

```
   for(j = 0;  j<100;   j++)
       scanf( “%s %d %d”, ss[j].name,
        &ss[j].g1, &ss[j].g2);

   find_av( ss, 100);

   for(j = 0;    j<100;    j++)
       printf( “%s %6.2f”, ss[j].name,
        ss[j].average);
   return 0;
}



void find_av(struct person ss[], int n)
{
    int j;
    for(j = 0; j < n;  j++)
        ss[j].average = (ss[j].g1 +
        ss[j].g2)/2;
}
```

Typedef

Meaningful identifiers can be created using typedef

```
typedef struct {
    int g1;
    int g2;
    char name[30];
    double average;
    }student;


student cop3502[100], cop3223[200];


void find_av(student course[], int n)
{
    int j;
    for(j = 0; j < n;  j++)
        course[j].average = (course[j].g1 +
                course[j].g2)/2;
}
```

File Handling

Let us consider a situation, where we have to read names and grades from a file, process the data, and store the result in a different file.
First of all you need to declare two file pointer variables:

```
FILE * infile;
FILE * outfile;
```

Then make an appropriate call to the *fopen* function.

```
if(( infile = fopen(“students.txt”,“r”))==
NULL)
    printf(“\n input file could not be
opened”);

if((outfile = fopen ( “new.txt” ,“w”) ) ==
NULL)
    printf(“\n output file could not be
opened”);
```

now you can start reading the data from the file into your temporary variables and arrays. Let us say that the first entry on the file indicates the total number of students.

```
fscanf( infile, “%d “, &num);
```

note the use of file scanf function   “fscanf”. Every fscanf reads one line from the file.
Now process the data

```
for (j = 0;  j <num;  j++)
```

```
    fscanf( infile, “%s %d “, names[ j ],
&grades[ j ]);
for (j = 0;  j <num;  j++)
     grades[ j ]   =  grades[ j ] + 10;
for (j = 0;  j <num;  j++)
    printf( “%s %d “, names[j], grades[ j ]);
```

Next to store the grades into the output file then you can use the following statement.
(*outfile* here refers to the *new.txt file* which we opened in the beginning).

```
fprintf( outfile, “%d “, num);
for (j = 0;  j <num;  j++)
    fprintf( outfile, “%s %d “, names[ j ],
grades[ j ]);
```

As there are only 3 columns, we can type from keyboard 3 values in each line.

```
fclose(infile);
fclose(outfile);
```