Introduction to C

- C is an More efficient Language to interact with hardware than other languages
- Languages before C were difficult to understand by humans and coding too
- So C invented to understand, ease of access easy to code by human these are mainly focused
- Due to this reason C still sustained these days too
- Developed by Dennis Ritchie

Application of C

- Works with Hardware easily
- Takes part on making Operating System(Command User Interface) Eg: like MS-DOS
- Drivers Software's and Other Simulation, System oriented software's done by C
- Compilers
- Filters
- Embedded System's

Features of C

- High level Language
- Highly intractable with hardware
- Easy to code
- Easy to understand
- Intermediate too
- Produce Efficient code
- Structure oriented
- C is an 2 parse compiling

Structure of C

- Documentation Section
- Header Section
- Main Section
- Other Function Declaration/ Definition
- Body of the main

Example C with Structure

```
// Simple calling
#include<stdio.h>
#include<conio.h>
void fun()
{
//
}
void main()
{clrscr();fun();getch();
}
```

Working process of C

- Skeleton (Actual Program)
- Preprocess(# and along this include library files should be defined first before compiling)
- Program (Except # headers)
- Compiling the program (Changes into high level to assembly and also compile whole program into stack at a time)
- Assembler(Change assembly into machine level)
- Linked---→link the all identifiers and functions
- Loader--→Load into CPU/Memory
- Output

Tokens of C

The formation words together called as language which is what we speak, Similarly formation tokens together as programming language. Followings are tokens

Comments →Description about program

Identifiers →Variables name, Function Name

Functions →A block of logic outside main function

Operators →To perform the various operation

Separators →To separate the fields

Keywords → Reserved keyword which not going to use as identifiers

Data Types → Various types data

Control/format string→ I/O access for data type variables

Escape sequences→ added attributes of cursor manipulations

Comments in C

- Single line comment
  - // (double slash)
  - Termination of comment is by pressing enter key
- Multi line comment

  /*….

  …….*/

  This can span over to multiple lines

Data types in C

- Primitive data types
  - int (2 byte), float(4 byte, it had 6 digits from the point I,e 6.567890), double(8 byte, it had 12 digits from point I,e 6.567890430000), char(1 byte)
- Aggregate(Derived) data types
  - Arrays, pointers come under this category
  - Arrays can contain collection of int or float or char or double data
- User defined data types
  - Structures, Union and enum fall under this category.

Variables

- Variables are data that will keep on changing
- Declaration

```
<<Data type>> <<variable name>>;
int a;
```

- Definition

```
<<varname>>=<<value>>;
a=10;
```

- Usage

```
<<varname>>
a=a+1;      //increments the value of a by 1
```

Variable names- Rules

- Should not be a reserved word like int etc..
- Should start with a letter or an underscore(_)
- Can contain letters, numbers or underscore.
- No other special characters are allowed including space
- Variable names are case sensitive
  - A and a are different.

Separators and Keywords

- Keywords those are all cannot use as identifiers
- Eg: keywords are…… if, else, main, switch, while, go, break etc………
- Separators are to divide the program or fields
- Eg: { } , ( ) ;

Input and Output

- Input
  - scanf(“%d”,&a);
  - Gets an integer value from the user and stores it under the name “a”
- Output
  - printf(“%d”,a)
  - Prints the value present in variable a on the screen

Control String and Escape Sequences

Control strings are:

- %d for int
- %f for float
- %c for char
- %s for string
- %lf for double

Escape sequences are:

- \n for new line
- \t for tab space
- \r for carrier return
- \b for backspace
- \a for alert

Operators

- Arithmetic
- Relational
- Logical
- Assignment/ Short hand
- Bit wise
- Increment/ Decrement

Arithmetic Operators

Operator	What It Is	Example	Result
=	assignment	int a = 8, b = 13, c = 3;	
+	addition	a = b + c;	a becomes 16
-	subtraction	a = b - c;	a becomes 10
*	multiplication	a = b * c;	a becomes 39
/	division	a = b / c;	a becomes 4

Increment or Decrement Operators

- The ++ operator is the equivalent of adding 1 to a variable
- There are two forms (assume the variable is an int variable called a with the value 8):
  - ++a (pre-increment)
  - a++ (post-increment)
- There's also a pre-decrement and post-decrement with the same basic behaviors

++ Operator

- The difference happens when you combine it with something else
  - ++a will do the increment first and then the other action
  - a++ will do the other action first and then the increment

Example of Pre-increment and Post-increment

```
int a = 8;
printf("%d", a);        8
printf("%d", ++a);      9
printf("%d", a);        9
printf("%d", a++);      9
printf("%d", a);        10
```

```
int a = 8;
printf("%d", a);        8
printf("%d", a++);      8
printf("%d", a);        9
printf("%d", ++a);      10
printf("%d", a);        10
```

Also note that we've put an arithmetic expression in a printf() argument

-- Operator

```
int a = 8;
printf("%d", a);        8
printf("%d", --a);      7
printf("%d", a);        7
printf("%d", a--);      7
printf("%d", a);        6
```

```
int a = 8;
printf("%d", a);        8
printf("%d", a--);      8
printf("%d", a);        7
printf("%d", --a);      6
printf("%d", a);        6
```

Assignment Operators

Operator	**Example**	**Result**
+=	int a = 8, b=8, b += a; (b=b+a)	b becomes 16
-=	b -= a;(b=b-a)	b becomes 8
*=	b *= a;(b=b*a)	b becomes 64
/=	b /= a;(b=b/a)	b becomes 8

Bit Wise Operation

Operations done on bit representations that is every operands change into bits and done bit wise operations

- &
- |

- Eg:- int a=7, b=4, c;

a →0111

b →0100   (&)

----

c →0100   c=4

a→0111

b→0100   (|)

----

c→0111   c=7

Relational Operators

Operator	***Meaning***	***Example***
==	Equal to	**count == 10**
!=	Not equal to	**flag != DONE**
<	Less than	**a < b**
<=	Less than or equal to	**<= LIMIT**
>	Greater than	**pointer > end_of_list**
>=	Greater than or equal to	**lap >= start**

Logical Operators

- Combination of relational operators called logical
- &&
- ||
- !

- Eg:-
- (a>b)&&(a>c)
- (a>=c)||(b<=a)
- !(a>b)

Control Statements

- Statement which controls the flow of the program
- Its all about conditions what we going to be write
- Regarding that condition instruct flow of the program whether its true / false
- Whether its true or false the corresponding block works only one time
- Conditions are written by using logical and relational operators
- Eg:- if(a>b){}

Control Statement keywords

- If
- Nested if
- If else
- If else if………… else
- switch

if-else

- **The** if-else **statement can exist in two forms: with or without the** else**. The two forms are:**

```
if(expression)
   statement
```

- **or**

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

if-else

- If

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

int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Nested if

- *nested* **if** adalah statement if yang targetnya adalahjuga if atau else.

```
if (i == 10) {
   if (j < 20) a = b;
   if (k > 100) c = d; // this if is
            else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

if-else-if Ladder

- Bentuknya:

```
if(condition)
   statement;
else if(condition)
            statement;
else if(condition)
statement;
.
.
.
else
statement;
```

if-else-if Ladder

```
// Demonstrate if-else-if statements (IfElse.c).
#include <stdio.h>
main () {
    int bulan = 4; // April
    char season[10];
    if(bulan == 12 || bulan == 1 || bulan == 2)
            strcpy(season,"Salak");
    else if(bulan == 3 || bulan == 4 || bulan == 5)
              strcpy(season,"Durian");
        else if(bulan == 6 || bulan == 7 || bulan == 8)
                 strcpy(season,"Mangga");
               else if(bulan == 9 || bulan == 10 || bulan == 11)
                       strcpy(season,"Jeruk");
                     else
                       strcpy(season,"Mbuh");
    printf("April adalah musim %s\n ",season);
}
```

switch

- switch merupakan statement percabangan dengan banyak cabang. Bentuknya seperti berikut:

```
switch (expression) {
    case value1:
            // statement sequence
            break;
    case value2:
            // statement sequence
            break;
    .
    .
    .
    case valueN:
            // statement sequence
            break;
    default:
            // default statement sequence
}
```

switch

- *expression* harus bertype byte, short, int, or char;

```
// A simple example of the switch(switch.c)
#include <stdio.h>
main() {
   int i;
   for(i=0; i<6; i++)
    switch(i) {
          case 0:
                       printf("i is zero.\n");
                                  break;
                 case 1:
                                  printf("i is one.\n");
                                  break;
                 case 2:
                                  printf("i is two.\n");
                                  break;
                 case 3:
                                  printf("i is three.\n");
                                  break;
                 default:
                                  printf("i is greater than 3.\n");
       } // switch
} // main
```

Looping---Iteration

The block of codes executes till the condition true or false, every loop block contains,

1.Initialization

2.Condition

3.Increment/ Decrement

Those are categorized as follows

- Entry check loop
  - While
  - for
- Exit check loop
  - Do while

While

- **while**
- **while** loop Condition checks first then body of the loop executes regarding with condition

```
while(condition) {
   // body of loop
}
```

while

```
// Demonstrate the while loop (while.c). Here even numbers
    between 10 to 100 should be print
#include <stdio.h>
    main() {
            int n = 10;
            while(n <=100) {
                    if(n%2==0){
                    printf("%d  \n",n);
                    }
                    n++;
            } // while
    } // main
```

do-while

- Body of the loop executes first then condition checked

```
do {
   // body of loop
} while (condition);
```

do-while

```
// Demonstrate the do-while loop (dowhile.c). Here even
    numbers between 10 to 100 should be print
#include <stdio.h>
    main() {
            int n = 102;
            do {
                    if(n%2==0){
                printf("%d  \n",n);
                    }
                n++;
            } while(n <= 100);
    } // main
```

// Here my range initiated against my objective that 10 to 100 but abpove initialted 102 though my body of the loop executes

for

- Initialization, condition, increment/ decrement done at same line

```
for(initialization; condition; iteration) {
   // body
}
```

for

```
// Demonstrate the for loop (loop.c). Here even numbers
    between 10 to 100 should be print
#include <stdio.h>
    main() {
            int n;
            for(n=10; n<=100; n++){
            if(n%2==0){
               printf("%d   \n",n);
            }
    }
    }
```

for

```
// Using the comma (comma.c)
#include <stdio.h>
    main() {
           int a, b;
           for(a=1, b=4; a<b; a++, b--) {
              printf("a = %d  \n", a);
              printf("b = %d  \n", b);
           }
    }
```

Nested Loops

- Like all other programming languages, C allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
// Loops may be nested (nestedfor.c).
   #include <stdio.h>
   main() {
            int i, j;
            for(i=0; i<10; i++) {
               for(j=i; j<10; j++)
                         printf(".");
               printf("\n");
            }
   }
```

Jump

- C supports four jump statements:
  - **break**,
  - **continue**,
  - **return**
  - **goto**.
- These statements transfer control to another part of your program.

break

- In C, the **break** statement has two uses.
  - First, as you have seen, it terminates a statement sequence in a **switch** statement.
  - Second, it can be used to exit a loop.

break

```
// Using break to exit a loop (break.c).
#include <stdio.h>
    main() {
         int i;
          for(i=0; i<100; i++) {
                    if(i == 10) break; // terminate loop if i is 10
                    printf("i: %d \n", i);
          }
          printf("Loop complete.");
    }
```

break

```
// Using break to exit a while loop (break2.c).
#include <stdio.h>
    main() {
          int i = 0;
          while(i < 100) {
             if(i == 10) break; // terminate loop if i is 10
             printf("i: %d  \n", i);
             i++;
          }
          printf("Loop complete.");
    }
```

continue

- continue go immediately to next iteration of loop
- In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression.

continue

```
// Demonstrate continue (continue.c).
#include <stdio.h>
    main() {
            int i;
            for(i=0; i<10; i++) {
                        if (i%2 == 0)
                                    continue;
                        printf("\n");
                        printf("%d ", i);
            }
    }
```

return

- The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- The following example illustrates this point. Here, **return** causes execution to return to the C, since it is the run-time system that calls **main( )**.

return

```
// Demonstrate return (return.c).
#include <stdio.h>
    main() {
        int t = 1;
        printf("Before the return.");
        if(t==1) return; // return to caller
        printf("This won't execute.");
    }
```

goto

- **It is possible to jump to any statement within the *same* function using goto.**
- **A label is used to mark the destination of the jump.**

```
goto label1;
:
:
label1:
```

goto

```
// Using continue with a label (goto.c).
#include <stdio.h>
    main() {
        int i,j;
          for (i=0; i<10; i++) {
                    for(j=0; j<10; j++) {
                      if(j > i) {
                              printf("\n");
                              goto outer;
                      }
                      printf(" %d", (i * j));
                    }
            outer: printf("..  outer ..\n");
          }

    }
```

Functions

- Function are block of logic which is used to avoid the confusion and conflict with bulk of coding
- It let us to make coding which understandable one
- “Divide and Conquer”
- Instead of writing all logic in main we could use function for better enhancement of coding

Functions

- Function declaration:
  - returntype functionname(argument);
- Function definition:
  - returntype functionname(argument)

```
{
    //body of the function
}
```

- Function calling: functionname(argument);

Functions types

Any function would be following these types of formats,

- Argument with return type
- Argument without return type
- No argument with return type
- No argument without return type

Argument with Return type

It contains the argument or parameter and return type syntax as,

```
returntype
fun.name(parameter)
{
//////
}
```

Eg:

```
int fun1(int a)
{
a*=a;
return a;
}
void main()
{
int a=7,r;
clrscr();
r=fun1(a);
getch();
}
```

Argument without Return type

It contains the argument but it doesn’t return anything from current function, Syntax as,

```
void
functionname(parameter)
{
//////
}
```

Eg:-

```
void fun1(int a)
{
printf(“%d”,a);a/=a;
}
void main()
{
fun1(8);
}
```

No Argument with Return type

It contains the return type but it doesn’t have the arguments, syntax as

```
returntype fun.name()
{
////
}
```

Eg:-

```
float fun1()
{
float c=3.14;
return c;
}
void main()
{
float r;
r=fun1();
}
```

No Argument without Return type

It doesn’t contain return type and doesn’t contain argument too, Syntax as

```
void fun.name()
{
////
}
```

Eg:-

```
void fun1()
{
char c;
c=‘R’;
}
void main()
{
fun1();
}
```

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - **`#include <math.h>`**

- Format for calling functions

  `FunctionName (argument);`
    - If multiple arguments, use comma-separated list
  - **`printf( "%.2f", sqrt( 900.0 ) );`**
    - Calls function **`sqrt`**, which returns the square root of its argument
    - All math functions return data type **`double`**
  - Arguments may be constants, variables, or expressions

Math Library Functions

- double acos(double x) -- Compute arc cosine of x.
  double asin(double x) -- Compute arc sine of x.
  double atan(double x) -- Compute arc tangent of x.
  double atan2(double y, double x) -- Compute arc tangent of y/x.
  double ceil(double x) -- Get smallest integral value that exceeds x.
  double cos(double x) -- Compute cosine of angle in radians.
  double cosh(double x) -- Compute the hyperbolic cosine of x.
  div_t div(int number, int denom) -- Divide one integer by another.
  double exp(double x -- Compute exponential of x
  double fabs (double x) -- Compute absolute value of x.
  double floor(double x) -- Get largest integral value less than x.
  double fmod(double x, double y) -- Divide x by y with integral quotient and return remainder.
  double frexp(double x, int *expptr) -- Breaks down x into mantissa and exponent of no.
  labs(long n) -- Find absolute value of long integer n.
  double ldexp(double x, int exp) -- Reconstructs x out of mantissa and exponent of two.
  ldiv_t ldiv(long number, long denom) -- Divide one long integer by another.
  double log(double x) -- Compute log(x).
  double log10 (double x) -- Compute log to the base 10 of x.
  double modf(double x, double *intptr) -- Breaks x into fractional and integer parts.
  double pow (double x, double y) -- Compute x raised to the power y.
  double sin(double x) -- Compute sine of angle in radians.
  double sinh(double x) - Compute the hyperbolic sine of x.
  double sqrt(double x) -- Compute the square root of x.
  void srand(unsigned seed) -- Set a new seed for the random number generator (rand).
  double tan(double x) -- Compute tangent of angle in radians.
  double tanh(double x) -- Compute the hyperbolic tangent of x.

Header Files

- Header files
  - contain function prototypes for library functions
  - **`<stdlib.h>`** , **`<math.h>`** , etc
  - Load with **`#include <filename>`**

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

- Custom header files
  - Create file with functions
  - Save as **`filename.h`**
  - Load in other files with **`#include "filename.h"`**
  - Reuse functions

Calling Functions: Call by Value and Call by Reference

- Used when invoking functions
- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions
- For now, we focus on call by value

Array

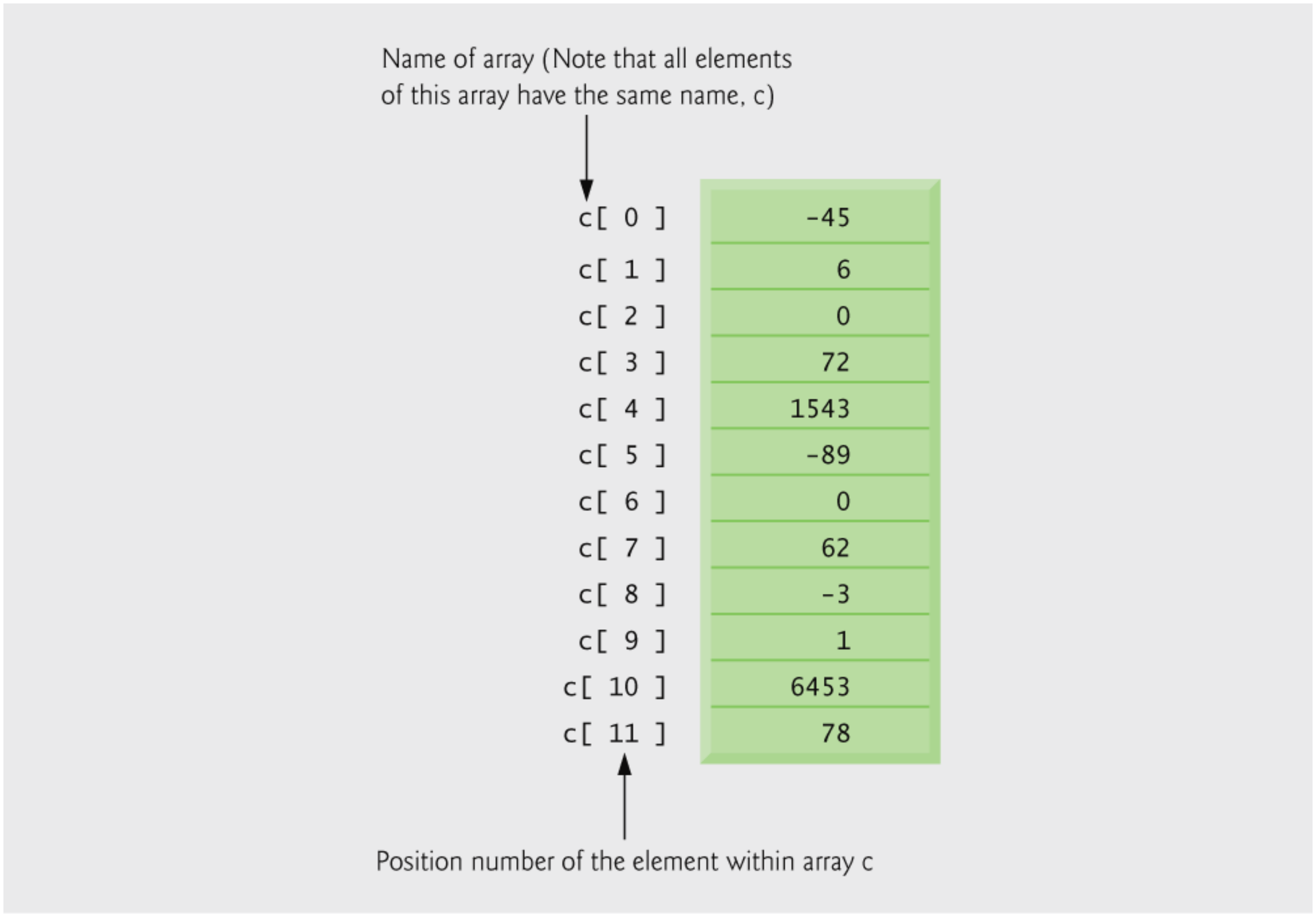
- A group of similar data types elements called an array
- A single variable of any type have single value at a time
- Array variable contains number of values according to what we gave before with reference of same variable name
- Eg: int a=7;→normal variable a have only one value 7, int a[4]={12,14,77,99};→array variable ‘a’ contains 4 int values with same variable name
- The distinguished values accessed regarding with their index number i,e a[0] refers 12,a[1] refers 14,a[2] refers 77, a[3] refers 99
- In memory it could be like this,

12	**14**	**77**	**99**

Array

- Array is an static memory allocation
- Which means if you allocate memory some amount during the initialization means you cannot increase at run time of that memory
- An array is an one of the derived data type, which means its depends with primitive data types such as, int float char double
- It let us we can create array with all the primitive data types
- Eg: int a[n],float f[n],char c[n],double d[n] like this
- Eg: if int a[4] in the sense you can store 4 elements but their index value I,e accessing of particulars starts with 0, a[0],a[1],a[2],a[3]

Array



Name of array (Note that all elements of this array have the same name, c)
c[0]
-45
c[1]
6
c[2]
0
c[3]
72
c[4]
1543
c[5]
-89
c[6]
0
c[7]
62
c[8]
-3
c[9]
1
c[10]
6453
c[11]
78
Position number of the element within array c

Array

- Declaration:
  - » *arrayname* [*size of array elements*]
  - » Eg: int a[4]={33,44,55,77}; or int a[4]={};a[0]=33;a[1]=44;a[2]=55;a[3]=77;
- Definition:
  - » Eg: a[0]+=a[3];
- Types:
  - One dimensional
  - Multi dimensional

Array example

```
void main()
{
float f[4]={4.3,6.7,8.9,9.07};
clrscr();
f[2]=7.1F;
getch();
}
```

Iteration in Array

- The index value is change till the array ending so we can use looping to get and print the array’s Eg:
- If array a[4]={4,5,6,7}; means, we can print the values one by one by using any loop as,
- int h=0;
- while(h<4)
- {
- printf(“%d”,a[h]);///if you want get values from user, use scanf
- h++;
- }

Operations in Array

- Swapping two Elements in array:

```
void main()
{
int a[2]={4,7};
a[0]=a[0]+a[1];//11
a[1]=a[0]-a[1];//4
a[0]=a[0]-a[1];//7 so
values are swapped
}
```

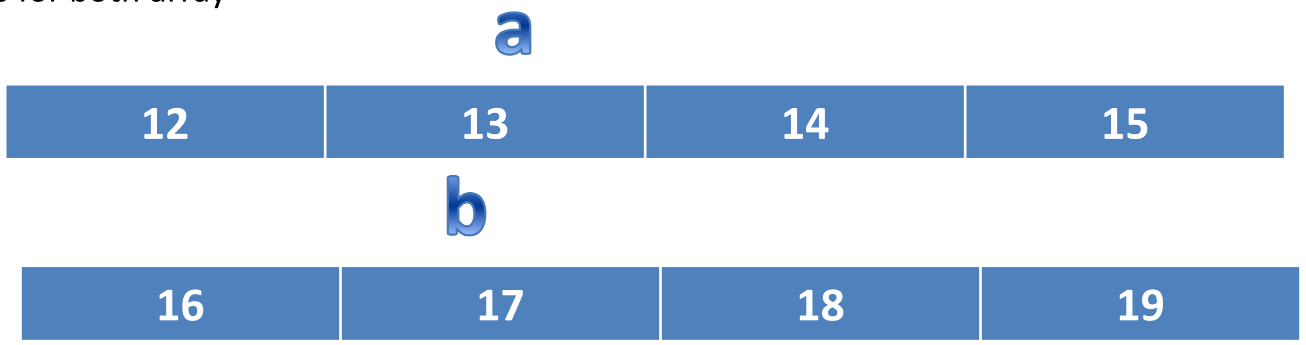
- If its more than two elements or swapping between two array means use temporary variable

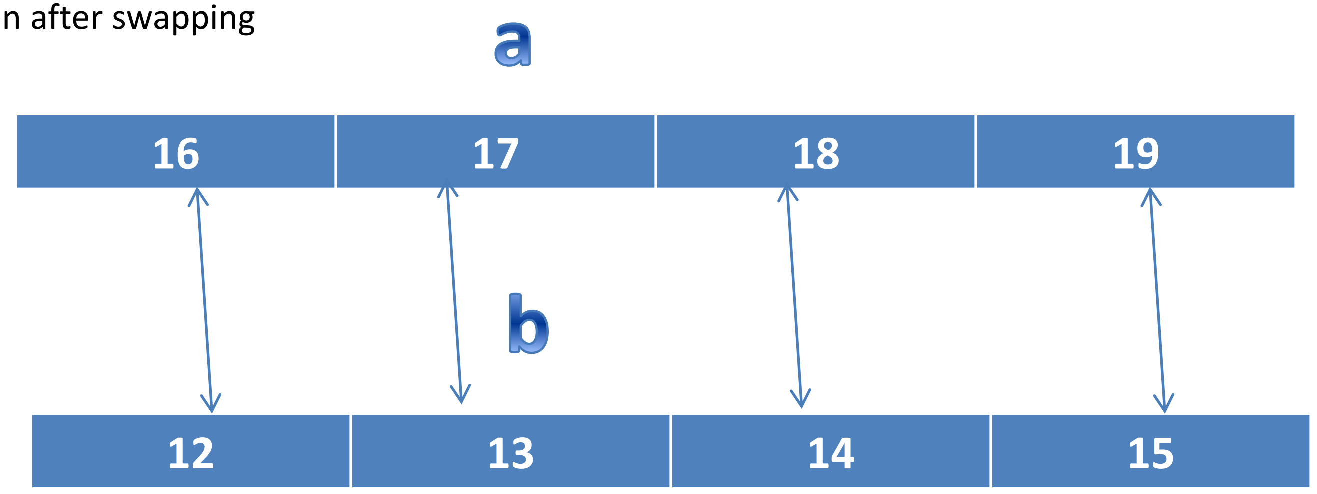
Swapping between two array’s

```
void main()
{
int a[4],b[4],t,y;
clrscr();
//get the values from the user for both  array at same time
for(y=0;y<4;y++)
{
scanf(“%d”,a[y]);
scanf(“%d”,b[y]);
}
//swapping process, that is each index of both values are swapped one by one eg, index 0 of a & b swapped
for(y=0;y<4;y++)
{
t=a[y];
a[y]=b[y];
b[y]=t;
}
}
```

Sample Output for above program

```
// inserting elements for both array
12//a[0]
16//b[0]
13//a[1]
17//b[1]
14//a[2]
18//b[2]
15//a[3]
19//b[3]
//swapping done then after swapping
```



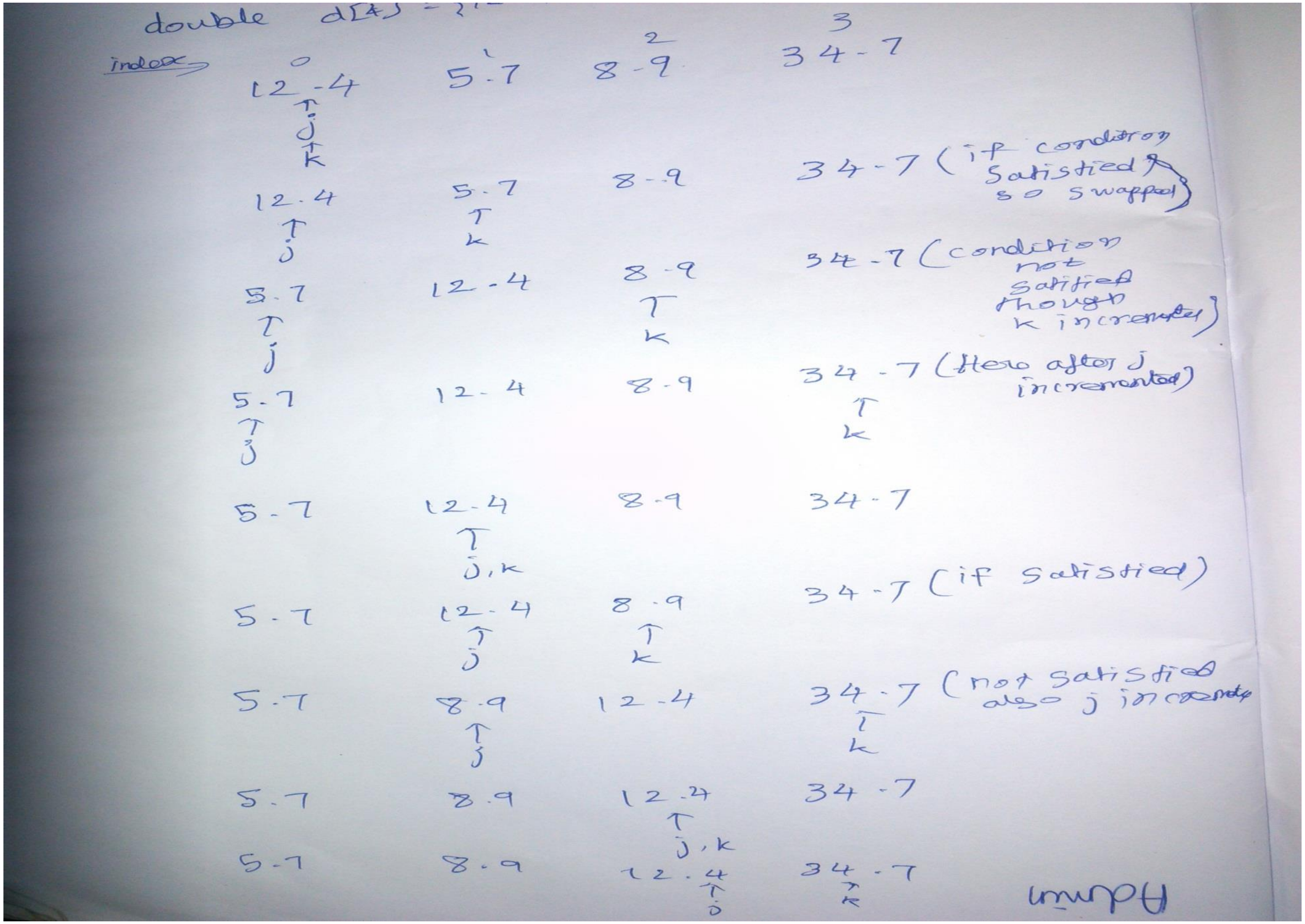


Changing array elements in Ascending/Descending order

```
void main()
{
double d[4]={12.4,5.7,8.9,34.7},t;
int j,k;
clrscr();
for(j=0;j<4;j++)
{
for(k=j;k<4;k++)
{
if(d[j]>d[k])//you want descending change '<'
{
t=d[j];
d[j]=d[k];
d[k]=t;
}
}
}
}
```

Logic of this program is, we have two loops one for holding one index which going to compare with other all index using another loop

Sample Output



```
double   d[4] = {1

                                    2            3
index→      0          1                         34.7
          12.4        5.7          8.9
           ↑
           j
           ↑
           k
                                               34.7 (if condition
                      5.7          8.9               satisfied
          12.4         ↑                              so swapped)
           ↑           k
           j
                                   8.9        34.7 (condition
         5.7         12.4           ↑                 not
          ↑                         k                 satified
          j                                           though
                                                      k incremented)
                                              34.7 (Here after j
         5.7         12.4          8.9                 incremented)
          ↑                                     ↑
          j                                     k

         5.7         12.4          8.9        34.7
                      ↑
                     j,k
                                              34.7 (if satisfied)
         5.7         12.4          8.9
                      ↑             ↑
                      j             k
                                              34.7 (not satisfied
         5.7          8.9         12.4          ↑    also j incremented
                      ↑                         k
                      j
         5.7          8.9         12.4        34.7
                                   ↑
                                  j,k
         5.7          8.9         12.4        34.7
                                   ↑            ↑
                                   j            k       [illegible]
```

Multi Dimensional Array(2 D)

- Here we goanna see two dimensional array
- A 2D declaration like this, int a[3][3]
- The first size one refers the rows and second one refers the column
- The size my vary according the need
- Here matrix addition and multiplication example will illustrated you to understand the multidimensional array concept

Matrix Addition

```
void main()
{
int a[3][3],b[3][3],c[3][3],i,j;
clrscr();
//get the values from user for both a and b array
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%d",&a[i][j]);
scanf("%d",&b[i][j]);
}
}
//print the values of a and b
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("\n%d",a[i][j]);
printf("\t%d",b[i][j]);
}
}
```

```
//adding a and b array elements into c array
For(i=0;i<3;i++)
{
For(j=0;j<3;j++)
{
C[i][j]=a[i][j]+b[i][j];
}
}
//printing the values of c array
For(i=0;i<3;i++)
{
For(j=0;j<3;j++)
{
Printf("\n%d",c[i][j]);
}
}
Getch();
}
```

Sample output:

```
2--→a[0][0]      8-→a[1][0]    4-→a[2][0]    c[0][0]→5
3--→b[0][0]      9-→b[1][0]    6-→b[2][0]    c[0][1]→9
4--→a[0][1]      0-→a[1][1]    3-→a[2][1]    c[0][2]→13
5--→b[0][1]      2-→b[1][1]    2-→b[2][1]    c[1][0]→17
6--→a[0][2]      2-→a[1][2]    1-→a[2][2]    c[1][1]→2
7--→b[0][2]      3-→b[1][2]    3-→b[2][2]    c[1][2]→5
                                              c[2][0]→10
                                              c[2][1]→5
                                              c[2][2]→4
```

Matrix Multiplication

- Try the same operation what we done before program, but instead of adding two array elements we goanna multiply them
- We need an extra for loop because as per matrix multiplication
- Here is the formula for matrix multiplication by using following

```
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
for(k=0;k<n;k++)
{
c[i][j]+=a[i][k]*b[k][j];
}
}
}
```

Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array without any brackets.
- The array size is usually passed to the function.

```
int myArray[ 24 ];
myFunction( myArray, 24 );
```

- Arrays are passed by-reference.
  - The name of the array is associated with the address of the first array element.
  - The function knows where the array is stored and it can modify the original memory locations.

Passing Arrays to Functions

- Individual array elements
  - Are passed by value.
  - Pass the subscripted name (i.e., `myArray[ 3 ]`) to function.
- Function prototype

```
void modifyArray( int b[], int arraySize );
```

  - Parameter names are optional in prototype.
    - `int b[]` could be written `int []`
    - `int arraySize` could be simply `int`

Passing Arrays to Functions

```
/* Arrays are passed using Call by Reference */
#include <math.h>
#define SIZE 6
void flip (float fray [], int fsize)
{
  float temp;
  int i,j;

  i = fsize - 1;
  for (j = 0; j < fsize/2 ; j++)
  {
    temp = fray[j];
    fray[j] = fray[i];
    fray[i] = temp;
    i--;
  }
  return;
}
```

Passing Arrays to Functions

```
int main ()
{
  float var[SIZE];
  int i,j;
  for (i=0; i < SIZE; i++)
  {
    var[i] = 1.0/pow (2.0,i);
    printf(" %5.3f", var[i]);
  }
  printf("\n");

  for (j=0; j < 2; j++)
  {
    flip (var, SIZE);
    for (i=0; i < SIZE; i++)
      printf(" %5.3f", var[i]);
    printf("\n");
  }
}
```

```
$./passray
 1.000 0.500 0.250 0.125 0.062 0.031
 0.031 0.062 0.125 0.250 0.500 1.000
 1.000 0.500 0.250 0.125 0.062 0.031
```

Character Array(String)

- In C, strings are arrays of char with a special characteristic:
  - the end of the string is denoted by the ASCII value 0
- This is called a null-terminated string.

Example

- char name[81] = "Fred Flintstone";
- This example has 81 bytes associated with it.
- Only 16 of those bytes are used: 15 for the name "Fred Flintstone" and 1 for the null-termination at the end of the string.
- The act of putting the name within double-quotes automatically null-terminates the string.

Declaring the String's Size

- Whenever you declare a string's size, you must allocate one **more** byte than you need for the characters you expect in the string.
  - This is because you need to account for space for the null-termination.

char str[10] has a **size** of:

33%	1. 9 bytes
33%	✓ 2. 10 bytes
33%	3. 11 bytes

NOTE: Don’t assume yet that str is used for a string.

If str is used as a string, char str[10] can contain ___ characters

33%	✓ 1. 9
33%	2. 10
33%	3. 11

The end of a string in C always contains:

33%		1. the character ‘0’
33%	✓	2. the ASCII value 0
33%		3. the string “0”

The null-termination at the end of a C string is used to:

25% 1. keep you from overflowing the array

25% [✓ 2. determine where the end of the string is]

25% 3. store the length of the string

25% 4. shorten the length of the array

If you try to put a 60 character long string in a 10 character long array:

25% 1. you will get a compiler error

25% [✓ 2. you will not get a compiler error but you will have a bug in your program]

25% 3. you will have a compiler warning and you might have a bug in your program

25% 4. the array will expand to handle it

Initializing an Empty String

- You can declare a string variable that has nothing in it:
  - e.g. char myName[81] = "";
  - This has nothing between the two double-quotes.
  - This puts a null-termination in myName[0].

What You **Can't** Do With Strings

- Except upon the variable declaration, you can't assign a string using an = :
  - e.g. you can't do: myName = "Fred";
    - You'll get an LValue Required (or something like that) error.
- You can't compare strings using == or any other comparison operator:
  - e.g. you can't do: if(myName == "Fred")
    - It's legal but wrong to do
- You can't add or append strings using a plus sign:
  - e.g. you can't do: myName = "Fred " + "Flintstone";

Use String Functions Instead

- To assign a string to an array of char, use strcpy():
  - e.g. strcpy(myName, "Fred");
  - This assumes that myName is an array of char with enough room to hold "Fred" and the null-termination.
  - If it doesn't have enough room, **it'll copy it anyway!**
- To append a string to another one in an array of char, use strcat():
  - e.g. strcat(myName, "Fred");
  - Same problem as above!

String Compare

- To compare two strings, use strcmp():
  - e.g. if(strcmp(myName, "Fred") == 0)
  - **VERY IMPORTANT NOTE:** strcmp() returns one of **three** values:
    - 0 if the strings are equal
    - a number less than 0 if the first argument is less than the second one
    - a number greater than 0 otherwise
  - Do NOT leave out the explicit comparison

String Comparison

- **IMPLICATION:** You cannot use strcmp() as if it were telling you if the arguments were equal to each other.
  - e.g. if(strcmp(myName, "Fred") == TRUE)
    - This will give you the **wrong answer** if you're assuming that TRUE returned means that the strings are equal!
  - e.g. if(strcmp(myName, "Fred"))
    - This will also give you the **wrong answer** if you're assuming that TRUE returned means that the strings are equal!

Assigning a value to a string is done like this:

50%	1. str = “blah”;
50%	✓ 2. strcpy(str, “blah”);

I know that assigning a value to a string is done with strcpy() and not by using = and I didn’t just answer the obvious answer on the previous slide

50%	1. TRUE
50%	2. FALSE

String equality comparison is done using:

33%		1. ==
33%	✓	2. strcmp()
33%		3. strcpy()

Using if(strcmp(str, “fred”)):

25% 1. will produce a compiler error

25% 2. will produce a compiler warning

25% [✓ 3. is legal but inadvisable because you can’t tell if the person knew what they were doing]

25% 4. is legal and a good idea

Comparing Strings

- Strings are compared "lexicographically".
  - i.e. the ASCII values of the characters are numerically compared, in order, until a mismatch is found or the end of a string is found.
  - Thus, "a" (ASCII value 97) is greater than "A" (ASCII value 65)
    - If you want to do a case-insensitive comparison, use stricmp().

More String Functions

- To find out how long a string is, use strlen():
  - e.g. len = strlen(myName);
- NOTE: len must be of type *size_t*.
  - *size_t* is a data type that can contain the longest length of a string in that particular operating environment.
  - On our systems, it is equivalent to unsigned int, so we can have rather long strings.

Example of String Usage

```
char str[81] = "", str2[81] = "Barney";
    strcpy(str, "Fred");
    printf("%s\n", str);
    strcat(str, " Flintstone");
    printf("%s\n", str);
    if( strcmp(str, str2) == 0 )
        printf("str:%s and str2:%s are both equal:", str,str1);
    else
        printf("%s and %s are different\n", str, str2);
```

OUTPUT:

Its doesn’t equal, so else part will be executed

Getting Keyboard Input

- There are several ways that you can use to get keyboard input in C.
- Some are better than others.

Getting A Single String From The Keyboard

- gets() can be used to get a string from the keyboard.
- It takes all the input you enter until you press ENTER and puts it into an array of char that you specify.
- e.g. gets(str1);

Problem With gets()

- gets() does not check for overflowing the array you specify.
- This is dangerous!

A Better, Slightly More Complicated Alternative

- fgets() is another function that you can use that will get a string from a file, while specifying a maximum size for the string.
- We can treat the keyboard as a file!
  - The keyboard has a special system variable set up for it: stdin.
  - We can use that variable with file-related functions.

Use of fgets()

- fgets() takes three parameters:
    1. the array of char in which you will put the string
    2. the number of characters you are willing to accept, including the null-termination
    3. the file from which you want to get the string (which would always be stdin if you want to get the string from the keyboard)

Example of fgets()

- e.g. fgets(str1, 81, stdin);
- This will get up to 80 (not 81) characters from the keyboard and put them in str1.
- If the user enters less than 80 characters before pressing ENTER, the string entered will get put into str1 **and** a carriage return (‘\n’) will be put at the end of the string, before the null-termination.
- If the user enters 80 characters or more, the carriage return is left out but the null-termination is put in.
  – This allows you to tell if the user has entered more characters than you will accept into the string.

Major Difference with gets()

- gets() will not put in the ‘\n’. fgets() will.
  - Remember this!

fgets() is better than gets() because:

33%

1. it allows you to specify a format string for input

33%

2. it allows you to get numbers as input

33%

[✓ 3. it gives you the opportunity to limit the amount of input so that you don’t overflow the array]

One important problem with using fgets() is:

33%

1. you can’t enter as much input as you want

33%

[✓ 2. it puts a ‘\n’ at the end of most input so you need to take that into account]

33%

3. you can’t use it for getting input from the keyboard

What If You Don’t Want To Hardcode The Array Size?

- You can automatically get the array size by using *sizeof* operator.
- e.g. fgets(str1, sizeof str1, stdin);
  - The compiler will figure out how big str1 is and provide that value as the second parameter to this function call.
- This is a Very Good Thing.

gets() vs. fgets()

- Due to the overflow protection from fgets(), you should probably always use fgets() with the stdin variable as the last parameter, instead of using gets().

What If You Want To Get Something Besides One String?

- There are other options for formatted input.

scanf()

- scanf() is a function that is heavily used in textbooks but is mostly avoided in real world situations.
- scanf() gets formatted input from the keyboard.
- It uses a format string similar to that found in printf().
  - scanf() scans the input string gotten from the keyboard, looking for the items specified in the format string.
  - By default, the items are separated by spaces.

Examples of Format Strings

Format String	**Meaning**
"%d %d %d"	Look for three consecutive integers, separated by spaces.
"%d %f %d"	Look for an integer followed by a floating point number, followed by an integer.
%s %d	Look for a string followed by an integer.
%d %s %d	Look for an integer, followed by a string, followed by an integer.

So, What's a String to scanf()?

- scanf() considers a string to be any sequence of characters terminated by whitespace or the end of the line.
- Thus, in user input, an integer can also be a string!
- Also, if you have two words as input, the space between them will separate them into two strings!

Examples of Matches to Format Strings

Format String	Sample Match
"%d %d %d"	170 200 201
"%d %f %d"	170 20.3 10
%s %d	fred 20
%s %d	10 20
%s %d %s	10 20 fred
%d %s %d	20 fred 40
%s %s %s	fred twinkletoes flintstone

A Better Way

- There is another function that is related to scanf(): sscanf().
- sscanf() scans strings, instead of the keyboard.
- It takes an additional string parameter **before** the format string.
- e.g. sscanf(str1, “%d %d”, &i, &j);

Using fgets() and sscanf()

- If we use fgets() to get a whole line of input from the user and sscanf() to scan that line once it’s put in a string, we can be much safer.

Example

```
fgets(str1, sizeof(str1), stdin);
if( sscanf(str1, "%d", &i) != 1 )
{
    printf(“Need to enter a number\n”);
}
```

Other Input Functions That Don't Work As Well As You Might Think

- getc()
- getchar()
- getch()
- getche()

Structure

- A **structure** has components, called ***members***, which can be of various types
- Structure is used for record maintaining like database in early days
- The ***declaration of a structure*** captures the information needed to represent the related structure.
- It is one of the user defined data type
- The keyword is ***struct***

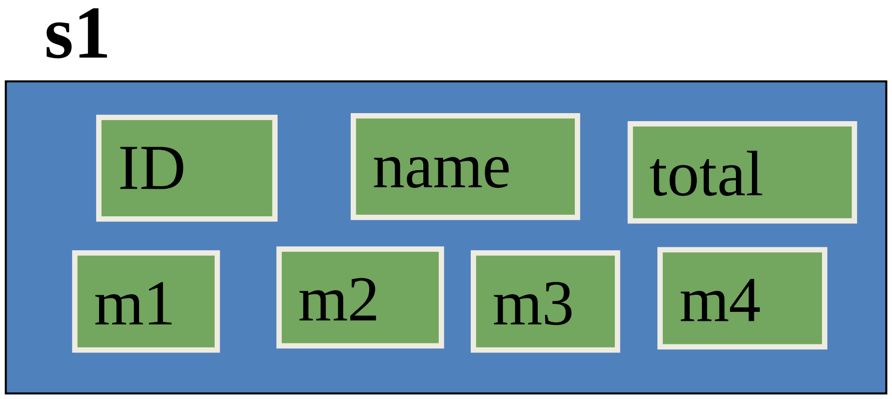
Structure

- Declaration:
  - struct userdefinedname obj/var;//obj is used to call the structure or refers the members of structures
- Definition:
  - struct userdefinedname {data type1 mem1;data type2 mem2};(or)
  - Struct userdefinedname{data type1 mem1;data type2 mem2;}obj;
- Accessing: (By using dot operator)
  - Obj.mem1;// we can create any number of object/variable for accessing the above all struct members

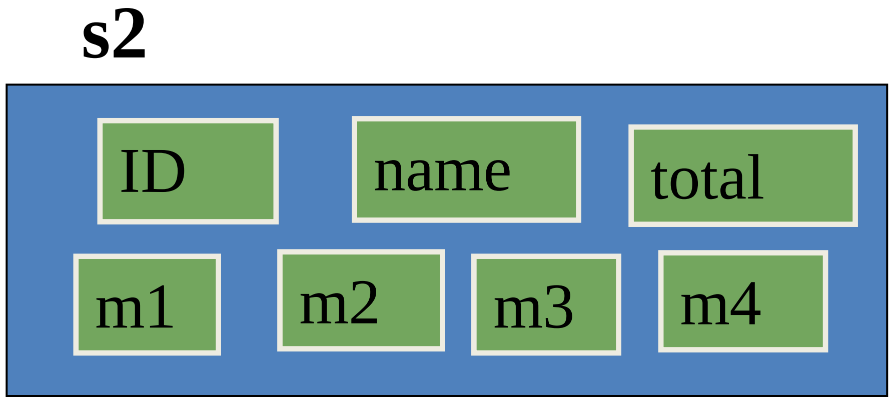
Example Structure

struct record { int ID; char name[10]; int total;int m1,m2,m3,m4; };

struct record s1;



struct record s2;



Example of Structure allocation

```
struct record
{
int ID;
char name[20];
int total,m1,m2,m3,m4;
};
void main()
{
struct record r;
clrscr();
printf("Getting values for struct elements from user");
scanf("%d",&r.ID);
scanf("%s",r.name);//when getting elements for string in structure we do not need to specify the '&' symbol
scanf("%d",&r.m1);
scanf("%d",&r.m2);
scanf("%d",&r.m3);
scanf("%d",&r.m4);
r.total=r.m1+r.m2+r.m3+r.m4;
printf("\nPrinting struct values");
printf("\n%d",r.ID);
printf("\n%s",r.name);
printf("\n%d\t%d\t%d\t%d",r.m1,r.m2,r.m3,r.m4);
printf("\n%d",r.total);
getch();
}
```

Array of Structure

```
struct record
{
int ID;
char name[20];
int total,m1,m2,m3,m4;
};
void main()
{
struct record r[10];
int i;
clrscr();
printf("Getting values for struct elements of ten records from
user");
for(i=0;i<10;i++)
{
scanf("%d",&r[i].ID);
scanf("%s",r[i].name);
scanf("%d",&r[i].m1);
scanf("%d",&r[i].m2);
scanf("%d",&r[i].m3);
scanf("%d",&r[i].m4);
r[i].total=r[i].m1+r[i].m2+r[i].m3+r[i].m4;
}
printf("\nPrinting struct values");
for(i=0;i<10;i++)
{
printf("\n%d",r[i].ID);
printf("\n%s",r[i].name);
printf("\n%d\t%d\t%d\t%d",r[i].m1,r[i].m
2,r[i].m3,r[i].m4);
printf("\n%d",r[i].total);
}
getch();
}
```

Nested Structure

Creating object of one struct inside another struct is called nested structure.

```
struct dob
{
int date;
char month[15];
int year;
};
struct record
{
int ID;
char name[20];
int total,m1,m2,m3,m4;
struct dob d;
};
void main()
{
struct record r[10];
int i;
clrscr();
printf("Getting values for struct elements of ten records from user");
for(i=0;i<10;i++)
{
scanf("%d",&r[i].ID);
scanf("%s",r[i].name);
scanf("%d",&r[i].m1);
scanf("%d",&r[i].m2);
scanf("%d",&r[i].m3);
scanf("%d",&r[i].m4);
scanf("%d",&r[i].d.date);
scanf("%s",r[i].d.month);
scanf("%d",&r[i].d.year);
r[i].total=r[i].m1+r[i].m2+r[i].m3+r[i].m4;
}
printf("\nPrinting struct values");
for(i=0;i<10;i++)
{
printf("\n%d",r[i].ID);
printf("\n%s",r[i].name);
printf("\n%d\t%d\t%d\t%d",r[i].m1,r[i].m2,r[i].m3,r[i].m4);
printf("\n%d",r[i].d.date);
printf("\n%s",r[i].d.month);
printf("\n%d",r[i].d.year);
printf("\n%d",r[i].total);
}
getch();
}
```

Function With Structure

- Declaration:
  - Return_type function_name(struct userdefinedname);
- Definition:
  - Return_type function_name(struct userdefinedname obj/var)

  {

  ///////

  }
- Calling:
  - Function_name(struct obj/var);

Example of Function with Structure

```
struct dob
{
int date;
char month[15];
int year;
};
struct record
{
int ID;
char name[20];
int total,m1,m2,m3,m4;
struct dob d;
}r[10];
void settingValue(struct record r[],int s)
{
int i;
for(i=0;i<s;i++)
{
scanf("%d",&r[i].ID);
scanf("%s",r[i].name);
scanf("%d",&r[i].m1);
scanf("%d",&r[i].m2);
scanf("%d",&r[i].m3);
scanf("%d",&r1[i].m4);
scanf("%d",&r[i].d.date);
scanf("%d",r[i].d.month);
scanf("%d",&r[i].d.year);
r[i].total=r[i].m1+r[i].m2+r[i].m3+r[i].m4;
}
}
```

```
void printingValue(struct record r[],int s)
{
int i;
for(i=0;i<s;i++)
{
printf("\n%d",r[i].ID);
printf("\n%s",r[i].name);
printf("\n%d\t%d\t%d\t%d",r[i].m1,r[i].m2,r[i].m3,r[i].m4);
printf("\n%d",r[i].d.date);
printf("\n%d",r[i].d.month);
printf("\n%d",r[i].d.year);
printf("\n%d",r[i].total);
}
}
void main()
{
int s=10;
clrscr();
printf("Getting values for struct elements of ten records from user");
settingValue(r,s);//calling setting function
printf("\nPrinting struct values");
printingValue(r,s);//calling printing function
getch();
}
```

union

- The **union**, like the **struct,** is a keyword to define a new data type.
- Structure members had its unique members size
- A **union** has the same syntax as a **struct** but has members that share the storage.
- Which means the highest size member in union shares its size to all other member
- A **union** type declaration, presents a set of alternative values that may be stored in a shared portion of memory.

union type declaration

```
#include<stdio.h>

union int_or_double  /* A NEW UNION TYPE DECLARATION*/
{    /* to be used as an integer OR a double*/
          int i;     /*needs 2 bytes*/
          double d; /*needs 8 bytes*/
};

struct int_and_double
{    /* to be used as an integer AND a double*/
          int i;     /*needs 2 bytes*/
          double d; /*needs 8 bytes*/
};

int main()
{
          printf ("%d\n", sizeof(int));
          printf ("%d\n", sizeof(double));
          printf ("%d\n", sizeof(union int_or_double));//8 bytes, this is why because the highest size double is the size of union
          printf ("%d\n", sizeof(struct int_and_double));//10 Bytes
          return 0;
}
```

union example

```
#include<stdio.h>

union int_or_char {
        char c;
        int i;
};

int main()
{
        union int_or_char test;
        test.i= 83;
        printf("i= %d\n", test);
        printf("c= %c\n",test);
        test.c= 'A';
        printf("i= %d\n",test);
        printf("c= %c\n", test);
        return 0;
}
```

bit fields

- The **bit field** is used to pack the bits
- User defined size can be allocated for primitive data types
- An **int** or **unsigned** member of a **structure** or **union** can be declared to consist of a specified number of bits. Such a member is called a **bit field**, and the number of associated bits is called its **width**. The **width** is at most the number of bits in a machine **word(4 bits).**
- The compiler packs the **bit field**s into a minimal number of machine **word**s.
- 0<8→1 byte, 8<16→2 byte like this calculated

Example of Bit fields

```
struct one
{
int i:4;
double d:4;
char c:4;
};
union two
{
int i:4;
double d:4;
char c:4;
}
void main()
{
struct one o;
union two t;
printf("Size of Structure:%d",sizeof(o));//2 bytes
printf("size of Union :%d",sizeof(t));//1 bytes, because highest is comman 4its so there is no bigger than 4 bits
}
```

Enumerator

- Enumerator is also an user defined data type like structure and union
- Declaration:
  - enum userdefinedname{m1,m2,m3};
- Definition:
  - enum userdefinedname obj/var;
  - Obj/var=m1;//output will be 0
  - Obj/var=m2;//output will be 1

Enumerator Example

```
enum colors
{
red,blue,green
};
void main()
{
enum colors c;
c=red;
printf("%d",c);//0
c=blue;
printf("%d",c);//1
c=green;
printf("%d",c);//2
}
```

Pointers

- Pointer is an variable, but stores address of another variable not storing values like normal variable
- A pointer is an derived data type from primitive types such as, int float double char
- Pointer used in dynamic memory allocation
- Easy and fast access to the memory
- A pointer variable distinguish from normal by specifying * in front of variable name
- Eg: int *p;→Pointer variable

Pointers

- If int a; which stored in the memory address 4040 in the sense we are assigning value to a=7,
- Then int *p; which value is p=&a, that is address of ‘a’, which is 4040 so the value of p is 4040
- When we try to print p like, printf(“%d”,p);//4040, we will get
- When we try to print *p, i.e printf(“%d”,*p);//then the value of the statement is 7, because it represent the value which stored in the address 4040

Pointers

- Declaration:
  - Data_type *ptr_var;
- Definition:
  - ptr_var=&another_var;
- Accessing:
  - &normal_var-→ address of particular normal variable in memory
  - *ptr_var-→ value which has been assigned in normal variable
  - Normal_var-→value of normal variable
  - ptr_var-→address of normal_var

Example of Pointers

```
void main()
{
int a=5,*i;
float b=12.5,*f;
clrscr();
printf("%d",a);//5
printf("\n%d",&a);//for just assumption may be 320
printf("\n%f",b);//12.5;
printf("\n%d",&b);//for just assumption may be 450
i=&a;
f=&b;
printf("\n%d",i);//320
printf("\n%d",f);//450
printf("\n%d",*i);//5
printf("\n%f",*b);//12.5
getch();
}
```

Pointers Integration

- Pointers with array
- Pointers with function
- Pointers with String(char array)
- Pointers with Structure

Pointers with Array

- In this circumstance the we need to make pointer to points initial index of array
- Eg: int a[3]={4,5,7},*p;p=&a[0];
- Advantage of using pointer in array is which reduce the usage of external variables like iteration variables in loops, so the memory space wastage avoided
- So linking iteration with array taking time and much space too

Example of Pointers with array

```
void main()
{
int a[5]={10,20,40,60,90},*p;
p=&a[0];
clrscr();
//Using pointer
while(*p!=NULL)
{
printf("\n%d",*p);
p++;
}
//normal iteration, which we used to do
for(int i=0;i<5;i++)
{
printf("\n%d",a[i]);// here we are using 'i' extra variable which always combine with array a
}
getch();
}
```

Pointers with function

- When we pass the value through the function it doesn’t change the original
- So the variable is local scope default, its value is valid particular block only
- But when we pass the address of variable instead of variable it always changes the original because value may change but address is same

Example of Pointers with function

```
//it accepts the integer value
void changev(int a)
{
printf("\n%d",a);//7
a+=40;
printf("\n%d",a);//47
}
void changep(int *p)
{
printf("\n%d",*p);//7
*p=(*p)+10;
printf("\n%d",*p);//17
}
void main()
{
int i;
clrscr();
i=7;
printf("\n%d",i);//i value 7
changev(i); //passing the value of i, throw to the changev function where int paramater catches the value
printf("\nAfter changev function:%d",i); //7 because we passed value of i, so doesn't changed here this is called call/pass by value
changep(&i); //here we pass address of i, the function changep catch the int address of i in pointer variable
printf("\nafter calling changep function:%d",i);//17 because value changed through the address passing this called call/pass by
reference
getch();
}
```

Pointers with String(char array)

- Here we are using pointers in char array
- But we are not goanna specify the size because when we using pointer in char array its referred as dynamic allocation
- i.e it grows its memory whenever values are added with it
- Here we goanna see about the string copy concatenation manually without using string functions
- Declaration: char * var_name;

Example of Pointers with string

- **String copy program**

```
void main()
{
char *name="JAVA",*copyname;
clrscr();
while((*name)!='\0')// ‘\0’→NULL
{
*copyname=*name;
printf("%c",copyname);
name++;
copyname++;
}
printf("\n%s",*copyname);
}
```

- **String concatenation**

```
void main()
{
char *c1="JAVA",*c2="J2EE",*c3;
clrscr();
//copying c1 to c3
while((*c1)!=NULL)
{
*c3=*c1;
printf("%c\n",*c3);
c1++;
c3++;
}
//copying c2 to c3
while((*c2)!=NULL)
{
*c3=*c2;
printf("%c\n",*c3);
c3++;
c2++;
}
printf("\n%s",*c3);
getch();
}
```

Pointers with Structure

- Here we goanna use pointer object/var instead of normal one
- While using normal variable as obj/var in structure we refers/access the structure member with help of (.)dot operator
- Whereas in pointer object we are referring/accessing structure member using (->).
- Eg: struct record *r; r->name;

Example of Pointers with structure

```
struct student
{
char *name;
int roll;
int total;
};
void main()
{
struct student *s;
scanf("%s",s->name);
scanf("%d",&s->roll);
scanf("%d",&s->total);
printf("\n%d",s->name);
printf("\n%d",s->roll);
printf("\n%d",s->total);
getch();
}
```

Storage Clauses/Type qualifiers

Type	Life time	Scope
auto(default qualifier)	Only current block	Within block
static	Till end of program	Entire program
extern	Till end of program	Entire program
register	Only current block	Within block

Type Conversions

- Conversion of one data type value into another one
- There are two ways of type conversion
  - Automatic(Implicit)→ Which lower data type to higher converted automatically by compiler, lower higher are depends on the size,

    Eg: int(2bytes) →float(4bytes)
  - Explicit→ It needs an manual conversion done by user,i.e higher to lower,

    Eg:float(4bytes)→int(2bytes)
    - » Syntax: (data_type)variable;

Example of type conversions

- Implicit conversions
  - Eg: int a=7;printf("%f",a);// here we used float format string and it prints like 7.0000, automatically converted
- Explicit conversions
  - Eg: float f=5.2;

```
        int i=(int) f;
        printf("%d",i);
```

Preprocessors

- Before compiling the program the preprocessor(#) considered first
- Preprocessor comes along with library files
- Preprocessor used as global variable declaration
- Eg: #define a 7

Example of Preprocessor

```
#define a 7
void fun()
{
printf("\n%d",a);//7
}
void main()
{
clrscr();
printf("%d",a);//7
fun();
getch();
}
```

Example of Preprocessor

```
#define square(x) x*x
void main()
{
int a,b;
clrscr();
a=square(7);
printf("%d",a);//49
b=square(5+2);
printf("\n%d",b);
getch();
}
```

- Always works as BUDMAS(bracket,unary, division,multiplication,a ddtion,subtraction) priority

5+2*5+2

5+10+2

15+2

17

b=17

Dynamic Memory Allocation

- Allocates the memory dynamically i.e heap memory storage
- Usually memory allocation done on stack portion in RAM, which automatically frees the contents(variables, function,..etc) in stack memory when program ended
- Whereas in dynamic memory contents stored in heap portion in RAM, which should be clear from memory manually
- Here we use some of function which used allocate, reallocate, free the memory from heap

Dynamic memory allocation Function

- malloc()→allocate the memory in heap
- realloc()→reallocate(alter) the memory which we previously allocated
- free()→remove the content from heap memory

Syntax of Dynamic Memory Allocation

- **malloc():**
- Syntax: ptr_var=(type *)malloc(size);
- Eg: i=(int *)malloc(sizeof(int));
- **realloc():**
- Syntax: ptr_var=realloc(ptr_var,resize);
- Eg: i=realloc(I,sizeof(int));
- **free():**
- Syntax: free(ptr_var);
- Eg: free(i);

FILE Operation

- Essential and important concept of C
- A file storage and manipulation on those file done on using C file functions
- In C FILE is a keyword which used to create files
- It is one of data structure in c
- Any type of text based file created by using C concept
- Normally we do copy, move, rename, …..etc., with the file
- These operation can be done by using C operations here

FILE Operations

- FILE declaration
- Opening file
- Writing on the file
- Reading contents from file
- Closing the file

FILE Declaration

- Syntax:
  - FILE *fptr_var;
- Eg:
  - FILE *f1

Opening FILE

- Syntax: fptr_var=fopen(“file”,”mode”);
- Where first parameter refers the filr which we need to open, it may new / existence one
- Second parameter defines the mode to open the file,
  - r→ Reading content from file
  - w→ Writing on the file, where if we specify existence file name in first parameter means this mode going be overwrite the contents, or if we specify the new file which is not in disk means, while using this mode its created on disk
  - a+→ Appending, which means instead of overwriting the contents in existence file, this mode to concatenate the contents

Writing Contents on FILE

- Functions are following used to write the contents on the FILE

- fputc()→for writing character on FILE
- fprintf(fptr_var,”Format string”,var)→ writing any data type
- fputs()→writing String on the FILE
- putc(character, fptr_var)→copy the character from first parameter to second parameter which is FILE pointer

Reading Contents from FILE

- The following are the functions used to reading the contents from FILE
- fscanf()→ reading any type of data type
- fgets()→reading apt content from FILE
- getchar()→reading character from FILE
- fgetc()→reading the character from FILE

FILE Accessing

- **FILE Accessing:**
- fseek()→used to navigate us to particular location what we wish
- frewind()→move to the beginning (like stop button in media players)
- Syntax: frewind(fptr);
- ftell()→ tell us location where we are In file now, it returns an integer value so we need to catch it
- Syntax:int position=ftell(fptr);
- Syntax:fseek(fptr,size,seeking);-->3 parameters are,
  - 1.file pointer
  - 2.size→-1,0,1 these are size used
  - 3.SEEK_CUR,SEEK_BEGIN,SEEK_END

Closing the FILE

- After the operation we have done FILE should be closed, it also depends on mode of opening file too
- Eg: opening file in write mode for write some thing on file, here if we want read something in the sense first close that write mode opening then open file in read mode
- Syntax closing file: fclose(fptr);

Example of Writing contents on FILE

```
void main()
{
FILE *f1;
char c;
clrscr();
f1=fopen("java.txt","w");//here java.txt not exists so after this statement it automatically created
//getting characters from user until pressing $ key
while(c=getchar()!='$')
{
putc(c,f1);
}
fclose(f1);
getch();
}
```

output:

```
JAVA Document is here
So Click $
```

// you can see this file following path

//tc→bin→java.txt

Copying contents from one file to another

```
void main()
{
FILE *f1,*f2;
char c;
clrscr();
f1=fopen("java.txt","r");//here assume java.txt already in disk then we goanna read the content
//getting characters from java.txt until EOF i,e End Of File, and loads into
f2=fopen("j2ee.txt","w");//here if j2ee.txt not exists in the sense it automatically created
otherwise contents going to be over writed
while(c=fgetc(f1)!=EOF)
{
putc(c,f2);
}
fclose(f1);
fclose(f2);
getch();
}
//go to TC→bin folder and see the file j2ee.txt contents whether it copied contents from java.txt
```

Appending Contents in FILE

```
void main()
{
FILE *f2;
char c;
clrscr();
f2=fopen("j2ee.txt","a+");//here if j2ee.txt contents going to be concatenated
with user input values, the user can give input until pressing #
while(c=getchar()!='#')
{
putc(c,f2);
}
fclose(f2);
getch();
}
//go to tc→bin, checks your file whether it appened
```

Thank You
For your
Patience