What is data structures? What are the different types of data structures?

Definition: The study of

- how the data can be collected and stored in main memory during execution
- how the data can be represented,
- how the data is organized or how the data is categorized
- how efficiently the data can be retrieved and manipulated

and the possible ways in which different data items are logically related is called data structure. The data structures are classified into :

- Primitive data structures
- Non-primitive data structures

	Names	Marks	Grade
A[0]	AMITABH	100	'B'
A[1]	SACHIN	101	'B'
A[2]	ARJUN	102	'A'
A[3]	BHIM	103	'A'
A[4]	MODI	104	'A'

n = 5

arranged in ascending order

What are primitive data structures?

Definition: The data structures that can be manipulated directly by machine instructions are called primitive data structures. The primitive data structures are fundamental data types that are supported by any programming language. For example,

- integers (int)
- floating point numbers (float)
- characters (char)
- double values (double)
- pointers

are all primitive data structures in C language.

What are non primitive data structures?

Definition: The data structures that cannot be manipulated directly by machine instructions are called non-primitive data structures. The non-primitive data structures are created or constructed using primitive data structures.

For example,

- arrays
- stacks
- queues
- linked lists
- trees

are all non-primitive data structures in C language.

What are the operations that can be performed on data structures?

The various operations performed on data structures are:

- Traversing
- Inserting
- Deleting
- Searching
- Sorting

What is traversing?

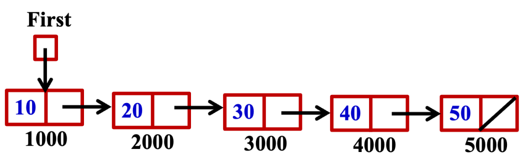
Definition: The process of accessing each item exactly once so that it can be processed and manipulated is called traversal.

For example, after traversing

- Print array elements // Output: 10 20 30 40 50
- Display each item in the list // Output: 10 20 30 40 50

a[0]	[1]	[2]	[3]	[4]
10	20	30	40	50

n = 5



When we use structures?

- An array is a collection of similar type of data items. Using arrays and other programming constructs, we can handle variety of situations.

- But, in real world, we can deal with entities that are collection of dissimilar data types. For example,
  - ▶ Name of the student (string type)
  - ▶ Marks scored (integer type)
  - ▶ Average marks (float type)

- Since, the above information is a collection of dissimilar data types, arrays cannot be used. In this situation, the structures are used.

- So, whenever we want to have a collection of similar or dissimilar data items that are logically related then we use structures.

What is a structure? What is the syntax to define a structure?

Definition: A structure is a collection of one or more declaration of variables of same data type or dissimilar data types, grouped together as a single entity.

- The variables defined inside the structure are called members of the structure or fields of the structure.
- All members are logically related data items.
- All the members can be accessed using a common name. It is a derived data type in C.

Syntax:

```
struct
{
     type1    member 1;
     type2    member 2;
     .......  .............
};
```

- struct is a keyword.
- member 1, member 2, etc., are the variables defined inside the structure. They are called members of the structure or fields of the structure.
- Semicolon is must at the end of the definition.

For example, a student information to be grouped may consist of

- name of the student //array of characters
- marks scored //integer
- average marks scored //float

```
struct
{
     char     name[20];
     int      marks;
     float    average;
};
```

Note:

- We know that all the variables are defined in the beginning of the function or before the function definition.
- On similar lines, the structures also should be defined either in the beginning of the function or before the function definition.

What are the different types of structures?

The structures can be classified as shown below:

- → Tagged structure (Named structure)
- → Tagless structure (Name less or Un-named structure)
- → Type defined structure

What is tagged structure? How to define tagged structure?

Definition: In the structure definition, the keyword struct can be followed by an identifier. This identifier is called tagname.

- The structure definition associated with tagname is called tagged structure or named structure.
- The syntax of tagged structure is shown below:

Syntax:

```
struct   tag_name
{
     type1     member 1;
     type2     member 2;
     .......   .............
};
```

Example:

```
struct   student
{
     char      name[10];
     int       marks;
     float     average;
};
```

What is structure declaration? How to declare structure variables?

- **By defining a structure, memory will not be reserved for members of a structure.**
- **Memory will be allocated for members of a structure, when the structure definition is associated with variables.**
- **The process of reserving the space for members of a structure is called structure declaration.**

Syntax:

```
struct  tag_name
{
      type1      member 1;
      type2      member 2;
      …….        ………….
};
```

Example:

```
struct  student
{
      char      name[10];
      int       marks;
      float     average;
};
```

struct tag_name v1, v2, …. vn; **structure declaration** **struct student cse;**

How and when memory is allocated for a structure?

- **A block of memory is allocated for structure variables.**
- **The memory for each member of a structure is allocated in the order specified within the braces.**
- **The size of block is the sum of individual sizes of all members of the structure.**

cse

	0	1	2	3	4	5	6	7	8	9
name										
marks										
average										

18 bytes

Example:

```
struct  student
{
      char      name[10];
      int       marks;
      float     average;
};
```

struct student cse;

How to initialize tagged structure?

Structure initialization: **The process of initializing the members of a structure is called structure initialization. During initialization, all the data items must be enclosed within braces i.e., '{' and '}' and are separated by commas.**

Syntax: struct tagname variable = { v1, v2, …… vn };

```
struct     student
{
      char      name[10];
      int       marks;
      float     average;
};
struct     student  a  =  { "ram",  25,  24.5 };
struct     student  b  =  { "mam", 24,  23.5 };
```

a

	0	1	2	3	4	5	6	7	8	9
name	r	a	m	\0						
marks	00	00	00	25						
average	00	00	00	24.5						

b

	0	1	2	3	4	5	6	7	8	9
name	m	a	m	\0						
marks	00	00	00	24						
average	00	00	00	23.5						

What is tagless or un-named structure? How to define tagless structure?

Definition: In the structure definition, the keyword **struct** is not followed by an identifier.

- That means, **there is no tag** associated with the structure.
- The structure definition without tagname is called **tagless structure** or **un-named structure** or **nameless structure.**
- The syntax of **tagless** or **un-named structure** is shown below:

Syntax:

```
struct
{
     type1     member 1;
     type2     member 2;
     …….       ………….
};
```

Example:

```
struct
{
     char      name[10];
     int       marks;
     float     average;
};
```

What is structure declaration? How to declare structure variables?

- By defining a structure, memory will not be reserved for members of a structure.
- Memory will be allocated for members of a structure, when the structure definition is associated with variables.
- The process of reserving the space for members of a structure is called **structure declaration.**

Syntax:

```
struct
{
     type1     member 1;
     type2     member 2;
     …….       ………….
}; v1, v2, ….  vn;
```

Example:

```
struct
{
     char      name[10];
     int       marks;
     float     average;
}; cse;
```

How and when memory is allocated for structure variables?

- A block of memory is allocated for structure variables.
- The memory for each member of a structure is allocated in the order specified within the braces.
- The size of block is the sum of individual sizes of all members of the structure.

cse

	0	1	2	3	4	5	6	7	8	9
name										
marks										
average										

18 bytes

```
struct
{
     char      name[10];
     int       marks;
     float     average;
} cse;
```

How to initialize one tagless structure?

Structure initialization: The process of initializing the members of a structure is called **structure initialization.** During initialization, all the data items must be enclosed within braces i.e., '{' and '}' and are separated by commas.

```
struct
{
     char      name[10];
     int       marks;
     float     average;
} cse = {"ram",  25, 24.5 };
```

cse

	0	1	2	3	4	5	6	7	8	9
name	r	a	m	\0						
marks	00	00	00	25						
average	00	00	00	24.5						

How to initialize more than one variable?

Structure initialization: The process of initializing the members of a structure is called **structure initialization.** During initialization, all the data items must be enclosed within braces i.e., '{' and '}' and are separated by commas.

only one variable is initialized

```
struct
{
    char    name[10];
    int     marks;
    float   average;
} a, b = { "ram",  25,  24.5 };
```

a

	0	1	2	3	4	5	6	7	8	9
name										
marks										
average										

b

	0	1	2	3	4	5	6	7	8	9
name	r	a	m	\0						
marks	00	00	00	25						
average	00	00	00	24.5						

More than one variable is initialized as shown below:

```
struct
{
    char    name[10];
    int     marks;
    float   average;
} a = { "ram",  25,  24.5 },  b = { "ham",  25,  24.5 };
```

a

	0	1	2	3	4	5	6	7	8	9
name	r	a	m	\0						
marks	00	00	00	25						
average	00	00	00	24.5						

b

	0	1	2	3	4	5	6	7	8	9
name	h	a	m	\0						
marks	00	00	00	25						
average	00	00	00	24.5						

What is type defined structure? How to define typedefined structure?

Definition: In the structure definition, the keyword struct is not followed by an identifier.

- It is a type of tagless structure. But, it is preceded by keyword typedef.
- The structure definition with the keyword typedef is called type defined structure.
- The type defined structure must be followed by an identifier ending with semicolon.
- This identifier acts as a data type. Using this type defined structure, we can declare variables.

Syntax:

```
typedef struct
{
    type1    member 1;
    type2    member 2;
    .......  .............
}TYPEID;

TYPEID v1, v2, .... vn;
```

Example:

```
typedef struct
{
    char    name[10];
    int     marks;
    float   average;
}STUDENT;

STUDENT  cse;
```

What are the different methods using which structure variables can be defined using typedef?

Method 1: Using tagged structure a structure variable can be declared as shown below:

```
struct student
{
    char    name[10];
    int     marks;
    float   average;
};
typedef struct student STUDENT;
STUDENT cse;
STUDENT ise;
```

Method 2: Using type-defined structure a structure variable can be declared as shown below:

Example:

```
typedef struct
{
    char    name[10];
    int     marks;
    float   average;
}STUDENT;

STUDENT cse;
```

How to initialize type-defined structure?

```
typedef struct
{
    char    name[10];
    int     marks;
    float   average;
} STUDENT;

STUDENT a = { "ram", 25, 24.5 };
STUDENT b = { "sam", 24, 23.5 };
```

a

	0	1	2	3	4	5	6	7	8	9
name	r	a	m	\0						
marks	00	00	00	25						
average	00	00	00	24.5						

b

	0	1	2	3	4	5	6	7	8	9
name	s	a	m	\0						
marks	00	00	00	24						
average	00	00	00	23.5						

How to initialize structure variables partially?

```
typedef struct
{
    char    name[10];
    int     marks;
    float   average;
} STUDENT;
STUDENT  a = { "ram" };
STUDENT  b = { "sam", 24, 23.5, 23.5 };  // Error
```

a

	0	1	2	3	4	5	6	7	8	9
name	r	a	m	\0						
marks	00	00	00	00						
average	00	00	00	00						

// The number of initial values should not exceed the number of members

```
STUDENT  c = { 24, 23.5 };  // Error
```

There is no way to initialize members in the middle of a structure without initializing the previous members.

Is it possible to initialize the members without declaring the variables?

Consider the following program segment:

Language

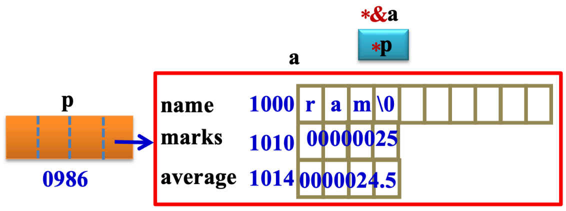
```
typedef struct
{
    char    name[10]= "RAMA";   // Error
    int     marks = 100;        // Error
    float   average = 14.5;     // Error
} STUDENT;
```

Note: It is not possible to initialize members without declaring a structure variable

What is pointer to a structure?

Definition: A variable which contains address of a structure variable is called **pointer to a structure.** For example, in the following program segment, the variable *p* holds the address of a structure variable. So, the variable *p* is **pointer to a structure.**

```
typedef struct
{
    char   name[10];
    int    marks;
    float  average;
} STUDENT;

STUDENT    a = {"ram", 25, 24.5 };
STUDENT   *p;
p = &a;
```



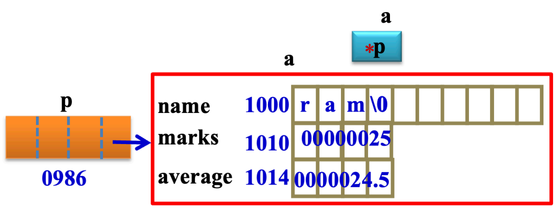
The members of a structure can be accessed using following ways:

- Using dot operator denoted by .
- Using de-referencing operator and dot operator denoted by * and .
- Using arrow operator denoted by ->

How to access the members of a structure using * and . operator?

- A member of a structure can be accessed by writing * followed by pointer variable but enclosed within parentheses followed by a dot and member name. SYNTAX: (*pointer_variable).member

```
#include <stdio.h>
void main ( )
{
    typedef struct
    {
        char   name[10];
        int    marks;
        float  average;
    } STUDENT;

    STUDENT    a = {"ram", 25, 24.5 };
    STUDENT   *p;
    p = &a;

    printf ("%s", (*p).name);
    printf ("%d", (*p).marks);
    printf ("%f", (*p).average );
}
```

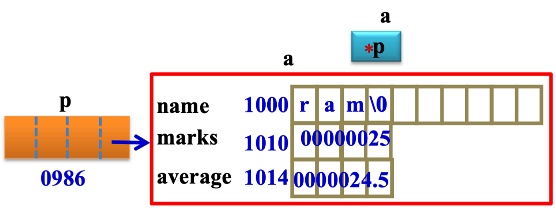


]

How to access the members of a structure using arrow operator?

- A member of a structure can be accessed by writing pointer variable followed by arrow operator in turn followed by member name. SYNTAX: pointer_variable -> member

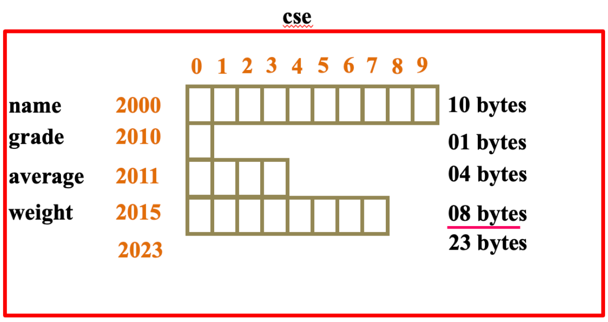
```
#include <stdio.h>
void main ( )
{
    typedef struct
    {
        char   name[10];
        int    marks;
        float  average;
    } STUDENT;
```



```
    STUDENT    a = {"ram", 25, 24.5 };
    STUDENT   *p;
    p =  &a;

    printf ("%s",  p -> name);
    printf ("%d",  p -> marks);
    printf ("%f",   p -> average );
}
```

What is the size of a structure?

- A block of memory is allocated.
- The memory for each member of a structure is allocated in the order specified within the braces.
- The size of a block is sum of individual sizes of members

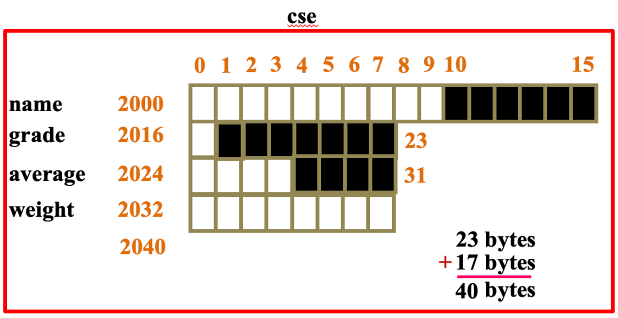



```
struct  student
{
    char      name[10];
    char      grade;
    float     average;
    double    weight;
};
struct  student  cse;
```

Note: The address of any member is greater than address of its previous member

What is the concept of slack bytes?

- A block of memory is allocated.
- The memory for each member of a structure is allocated in the order specified within the braces at word boundaries.
- The size of a block is sum of individual sizes of members and number of slack bytes.




```
struct  student
{
    char      name[10];
    char      grade;
    float     average;
    double    weight;
};
struct  student  cse;
```

- In the structure definition, double is the largest data type with size 8 bytes. So, the starting address of each member should be divisible by 8.
- In some machines, the memory for the members of a structure is allocated at certain boundaries called word boundaries.
- In such cases, extra bytes are padded at the end of each member whose size is less than the size of largest data type so that the address of each member starts at word boundary.
- The extra bytes that are inserted at the end of each member are called slack bytes.

Note: The slack bytes are shown using black boxes in the above figure.'

What is the advantage/disadvantage of slack bytes?

- The slack bytes do not contain any valid information and are useless wasting the memory space.
- In this situation, the size of structure may be greater than the size of individual members.
- But, the advantage is that data accessing at word boundaries is very fast.
- The size of a structure may be equal to the size of individual members. In such case, no slack bytes are used.

What are the operations that can be performed on structures?

The various operations that can be performed on structures are:

- → Copying of structure
- → Comparing members of a structure
- → Arithmetic operations on structures

- It is possible to assign a member of one structure to member of another structure if the type of those members is same.

```
a.marks   =  c.marks;        // OK
a.marks   =  c.average;      // ERROR
a.name    =  c.name;         // ERROR
```

How a structure can be copied?

- A copy of a structure can be obtained using assignment operator.
- But, one structure can be assigned to another structure of same structure type.

Example1:

```
struct
{   char    name[10];
    int     marks;
    float   average;
} a, b;

a = b;   // OK
b = a;   // OK
```

Example2:

```
struct
{   char    name[10];
    int     marks;
    float   average;
}  c, d;

c = d;   // OK
d = c;   // OK
```

```
a = c;   // Error
a = d;   // Error
c = b;   // Error
d = a;   // Error
```

Note: Even though the members of both structures are same in number and type, both structures are considered to be different.

- It is possible to assign a member of one structure to member of another structure if the type of those members is same.

```
a.marks   =  c.marks;          // OK
a.marks   =  c.average;        // ERROR
a.name    =  c.name;           // ERROR
strcpy (a.name,  c.name);      // OK
```

How to compare two structures?

- Comparing two structures is not allowed.

Example1:

```
struct
{       char      name[10];
        int       marks;
        float     average;
} a, b;

if ( a == b )      // Error
if ( a != b )      // Error
```

Example2:

```
struct
{       char      name[10];
        int       marks;
        float     average;
}  c, d;

if ( c == d )      // Error
if ( a != d )      // Error
```

- However, comparing members of different structures is allowed.

```
if ( a.marks == c.marks )          // OK
if ( a.name == c.name )            // ERROR
if ( strcmp(a.name, c.name) == 0 )// OK
```

How arithmetic operations are performed in a structure?

- Arithmetic operations on two structures is not allowed.
- However, arithmetic operations are allowed on members of a structure.

Example1:

```
struct
{       char      name[10];
        int       marks;
        float     average;
} a, b;
```

Example2:

```
struct
{       char      name[10];
        int       marks;
        float     average;
}  c, d;
```

For example,

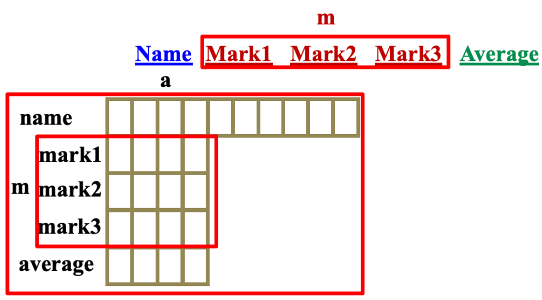
```
int       marks;
marks =  a.marks + c.marks;          // OK
```

What is nested structure?

```
typedef  struct
{   int        mark1;
    int        mark2;
    int        mark3;
} MARKS ;

typedef  struct
{   char       name[10];
    MARKS      m;
    float      average;
} STUDENT ;

STUDENT     a;
```

Definition: A structure inside a structure is called nested structure. As we declare variables inside a structure, a structure can also be declared inside a structure. So, a structure whose member itself is a structure is a nested structure.

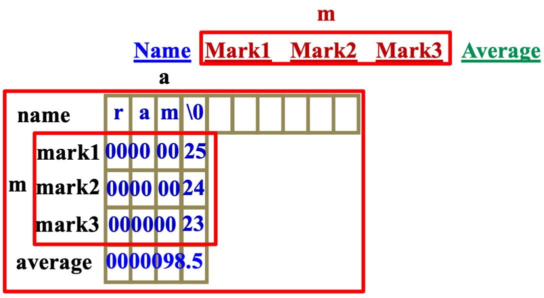
For example, the structure STUDENT is a nested structure.



How to initialize the members of a nested structures?

```
typedef  struct
{
    int        mark1;
    int        mark2;
    int        mark3;
} MARKS ;

typedef  struct
{
    char       name[10];
    MARKS      m;
    float      average;
} STUDENT ;

STUDENT    a = {  "ram",
                  {25,   24,   23 },
                  98.5
               };
```

- The variable in the declaration must be followed by '=' sign and followed by data items.
- The data items that are to be initialized must be separated by commas.
- The data items that are to be initialized must be enclosed within braces.
- The data items thus initialized are stored in memory as shown below:

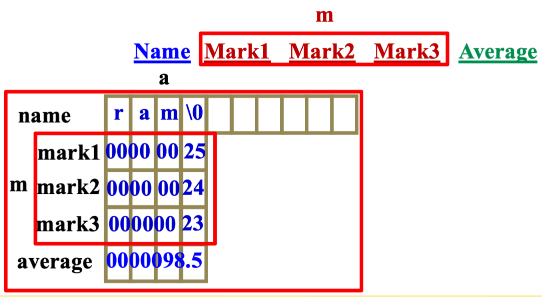


How to access the members of a nested structures?

```
typedef  struct
{
    int        mark1;
    int        mark2;
    int        mark3;
} MARKS ;

typedef  struct
{
    char       name[10];
    MARKS      m;
    float      average;
} STUDENT ;
```

- The data stored in each member can be accessed using dot operator as shown below:

```
a.name          // ram
a.m.mark1       // 25
a.m.mark2       // 24
a.m.mark3       // 23
a.average       // 98.5
```



How to read the information of students?

```
typedef  struct
{
    int        mark1;
    int        mark2;
    int        mark3;
} MARKS ;

typedef  struct
{
    char       name[10];
    MARKS      m;
    float      average;
} STUDENT ;
```

```
void  read_student_info ( STUDENT a [] ,  int  n )
{
    int    i;

    printf ("Name mark1 mark2 mark3 average\n");
    for ( i =  0;  i  <  n;  i++ )
    {
        scanf (" %s %d %d  %d  %f ", a[i].name,
                  &a[i].m.mark1,     &a[i].m.mark2,
                  &a[i].m.mark3,     &a[i].average ) ;
    }
}
```

m

	Name	Mark1	Mark2	Mark3	Average
a[0] =	RAMA	90	90	87	89.0
a[1] =	BAMA	85	85	85	85.0
a[2] =	SOMA	95	95	92	94.0
a[3] =	MAMA	98	98	95	97.0
a[4] =	YAMA	97	97	94	96.0

n = 5

How to sort the student details in increasing order of average marks?

```
typedef  struct
{
    int         mark1;
    int         mark2;
    int         mark3;
} MARKS ;

typedef  struct
{
    char        name[10];
    MARKS       m;
    float       average;
} STUDENT ;
```

// Input:array a with n items

```
∀ j = 1  to  n - 1
    ∀ i = 0 to  n - ( j + 1 )
        if  (a[i] > a[i + 1])
        {
            temp = a[i];
            a[i] = a[i + 1];
            a[i + 1]= temp;
        }
```

```
void  read_student_info ( STUDENT a [] ,  int  n )
{
    int    i;

    printf (“Name mark1 mark2 mark3 average\n”);
    for ( i =  0;  i  <  n;  i++ )
    {
        scanf (“  %s %d %d  %d  %f “, a[i].name,
                &a[i].m.mark1,    &a[i].m.mark2,
                &a[i].m.mark3,    &a[i].average ) ;
    }
}

void  sort_student_info (STUDENT   a [],  int  n)
{
    int  i,  j;    STUDENT   temp;

    for ( j = 1;   j  <  n;   j++ )
    {
        for ( i = 0;  i  <  n – j;  i++ )
        {
            if  ( a[i].average   > a[i + 1].average)
            {
                temp  =  a[i] ;
                a[i]  =  a[i + 1] ;
                a[i + 1]   =  temp ;
            }
        }
    }
}
```

The complete program to read student inform, sort student info and to print student info can be written as shown below:

```
#include   <stdio.h>

typedef  struct
{
    int         mark1;
    int         mark2;
    int         mark3;
} MARKS ;

typedef  struct
{
    char        name[10];
    MARKS       m;
    float       average;
} STUDENT ;

void  main  ( )
{
    STUDENT       a[10];
    int           n;

    printf (  “Enter no. of students : “ );
    scanf (  “%d”,  & n );

    read_student_info ( a,  n );
    sort_student_info  ( a,  n );
    print_student_info ( a,  n );
}

void  print_student_info ( STUDENT a [] ,  int   n )
{
    int    i;

    printf (“Name mark1 mark2 mark3 average\n”);
    for ( i =  0;  i  <  n;  i++ )
    {
        printf ( “%s %d  %d  %d  %f \n “, a[i].name,
              a[i].m.mark1,  a[i].m.mark2,  a[i].m.mark3,
               a[i].average ) ;
    }
}

void  read_student_info ( STUDENT a [] ,  int  n )
{
    int    i;

    printf (“Name mark1 mark2 mark3 average\n”);
    for ( i =  0;  i  <  n;  i++ )
    {
        scanf (“  %s %d %d  %d  %f “, a[i].name,
                &a[i].m.mark1,    &a[i].m.mark2,
                &a[i].m.mark3,    &a[i].average ) ;
    }
}

void  sort_student_info (STUDENT   a [],  int  n)
{
    int  i,  j;    STUDENT   temp;

    for ( j = 1;   j  <  n;   j++ )
    {
        for ( i = 0;  i  <  n – j;  i++ )
        {
            if  ( a[i].average   > a[i + 1].average)
            {
                temp  =  a[i] ;
                a[i]  =  a[i + 1] ;
                a[i + 1]   =  temp ;
            }
        }
    }
}
```

Can a structure contain array as a member name?

Yes. Definitely a structure can contain an array as the member name. Consider the following structure.

```
typedef  struct
{
    char        name[10];
    int         marks[3];
    float       average;
} STUDENT ;

STUDENT     a = {“RAMA” , { 10,  20,  30 },  20.0 };
```

name										marks [0]	marks [1]	marks [2]	average
R	A	M	A	\0						00000010	00000020	00000030	0000020.0

a

- The variable in the declaration must be followed by ‘=‘ sign and followed by data items.
- The data items that are to be initialized must be separated by commas.
- The data items that are to be initialized must be enclosed within braces.

Note: The members “name” and “marks” are arrays inside the structure “STUDNT”.

How to initialize the members of a structure when structure has array as the member name? A structure having array as the member can be initialized as shown in program segment below:

```
typedef  struct
{
     char       name[10];
     int        marks[3];
     float      average;
} STUDENT ;

STUDENT    a = {"RAMA" , { 10,  20,  30 },  20.0 };
```

name										marks [0]	marks [1]	marks [2]	average
R	A	M	A	\0						00000010	00000020	00000030	0000020.0

a

- The variable in the declaration must be followed by '=' sign and followed by data items.
- The data items that are to be initialized must be separated by commas.
- The data items that are to be initialized must be enclosed within braces.

- The data stored in each member can be accessed using dot operator as shown below:

```
a.name          // RAMA
a.marks[0]      // 10
a.marks[1]      // 20
a.marks[2]      // 30
a.average       // 20.0
```

How to initialize the structures having arrays as member name? The structure having array name as member can be initialized as shown below:

```
typedef  struct
{
     char       name[10];
     int        marks[3];
     float      average;
} STUDENT ;

STUDENT   a [] =
{
     {"RAMA" , { 90,  90,  87 },  89.0 },
     {"BAMA" , { 85,  85,  85 },  85.0 },
     {"SOMA" , { 95,  95,  92 },  94.0 },
     {"MAMA", { 98,  98,  95 },  97.0 },
     {"YAMA" , { 97,  97,  94 },  96.0 }
};
```

	name	marks [0]	marks [1]	marks [2]	average
a[0] =	RAMA	90	90	87	89.0
a[1] =	BAMA	85	85	85	85.0
a[2] =	SOMA	95	95	92	94.0
a[3] =	MAMA	98	98	95	97.0
a[4] =	YAMA	97	97	94	96.0

	name	marks [0]	marks [1]	marks [2]	average
a[0] =	RAMA	90	90	87	89.0
a[1] =	BAMA	85	85	85	85.0

The complete program to read the student info, sort the student info and to print the student info can be written as shown below:

```
#include    <stdio.h>

typedef  struct
{
     char       name[10];
     int        marks[3];
     float      average;
} STUDENT ;
```

```
void  main ( )
{
     STUDENT      a[10];
     int          n;

     printf (  "Enter no. of students : " );
     scanf (  "%d",   & n );

     read_student_info ( a, n );

     sort_student_info  ( a, n );

     print_student_info ( a, n );
}
```

```
void   sort_student_info ( STUDENT    a [],  int   n)
{
     int   i,  j;     STUDENT     temp;

     for ( j = 1;    j  <   n;    j++ )
     {
          for ( i = 0;   i  <   n – j;   i++ )
          {
               if  ( a[i].average   > a[i + 1].average)
               {
                    temp  =  a[i] ;
                    a[i]  =  a[i + 1] ;
                    a[i + 1]  =  temp ;
               }
          }
     }
}
```

```
void   print_student_info ( STUDENT a [] ,  int   n )
{
     int     i,  j;
     printf ("Name marks1 marks2 marks3 average\n");
     for ( i  = 0;  i < n; i++ )
     {
          printf ( "%s ",  a[i].name) ;
          for ( j  = 0;  j < 3; j++)
               printf ( " %d ",  a[i]. marks[j] ) ;
          printf ( " %f ",  &a[i].average ) ;
     }
}
```

```
void   read_student_info ( STUDENT a [] ,  int   n )
{    int     i,  j;
     printf ("Name mark1 mark2 mark3 average\n");
     for ( i  = 0;  i < n; i++ )
     {
          scanf (" %s ",  a[i].name) ;
          for ( j  = 0;  j < 3; j++)
               scanf ( " %d ",  &a[i]. marks[j]) ;
          scanf ( " %f ",  &a[i].average ) ;
     }
}
```

What are the different ways of passing structures/members to functions?

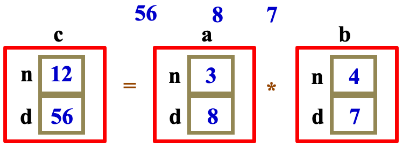
The various ways to pass structure or its members to the functions:

- Passing members of a structure
- Passing the structures
- Passing the address of structures

How to pass structure members as parameters?

$$\frac{12}{56} = \frac{3}{8} * \frac{4}{7}$$

```c
#include <stdio.h>

typedef  struct
{
    int     n;
    int     d;
}  FRACTION;

int  multiply ( int  x,   int  y)
{
     return      x * y;
}
```



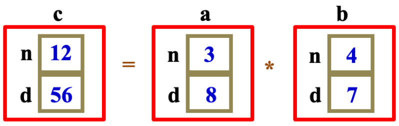

```c
void main()
{
    FRACTION   a,  b,  c;

    printf (“Fraction1:(x/y)”);
    scanf ( “%d/%d”,  &a.n, &a.d  );

    printf (“Enter fraction2:(x/y)”);
    scanf ( “%d/%d”,   &b.n, &b.d );

    c.n  =  multiply ( a.n,  b.n );
    c.d  =  multiply ( a.d,  b.d );

    printf ( “Result = %d/%d”,  c.n,  c.d );
}
```

Disadvantages

- The return address and the values of actual parameters are pushed on to the stack (Last in first out data structure).
- As the number of actual parameters increases, the size of the stack also increases.
- As the size of stack increases, the memory space utilized also increases.
- Hence, it is not a good practice to pass the members. It is not a good programming style and performance decreases.
- This method is inefficient as the number of members increases and require more memory.

How to pass structure to a function?

```c
#include <stdio.h>

typedef  struct
{
    int     n;
    int     d;
}  FRACTION;

FRACTION  multiply  (FRACTION   x,  FRACTION    y )
{
      FRACTION     z;

      z.n   =      x . n  *     y  . n;
      z.d   =      x . d  *     y  . d;

      return    z;
}
```




```c
void main()
{
    FRACTION   a,  b,  c;

    printf (“Fraction1:(x/y)”);
    scanf ( “%d/%d”,  &a.n, &a.d  );

    printf (“Enter fraction2:(x/y)”);
    scanf ( “%d/%d”,   &b.n, &b.d );

    c   =    multiply  (   a,    b );

    printf ( “Result = %d/%d”,  c.n,  c.d
}
```

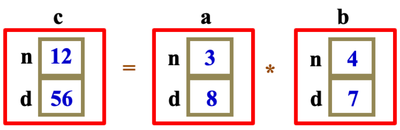
Disadvantages

- When a function is called the entire structure will be pushed on to the stack.
- The size occupied by the structure on the stack is equal to the sum of sizes of individual members. So, more time is required for copying it into stack and hence efficiency of the program decreases.

Note: The above disadvantages are overcome by passing addresses of structures as actual parameters.

How to pass address of a structure to a function?

```c
#include <stdio.h>

typedef  struct
{
    int     n;
    int     d;
}  FRACTION;
```




```c
void main()
{
    FRACTION   a,  b,  c;

    printf (“Fraction1:(x/y)”);
    scanf ( “%d/%d”,  &a.n, &a.d  );

    printf (“Enter fraction2:(x/y)”);
    scanf ( “%d/%d”,   &b.n, &b.d );
```

```
FRACTION multiply (FRACTION *x, FRACTION *y )
{
      FRACTION   z;

      z.n  =  (*x) . n  *  (*y) . n;
      z.d  =  (*x) . d  *  (*y) . d;

      return   z;
}
```

```
      c  =   multiply (&a, &b );

      printf ( “Result = %d/%d”, c.n, c.d
}
```

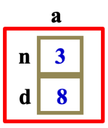
The above function multiply can also be written using array operator as shown below:

```
FRACTION multiply (FRACTION *x, FRACTION *y )
{
      FRACTION   z;

      z.n  =    x-> n  *   y-> n;
      z.d  =    y-> d  *   y-> d;

      return   z;
}
```

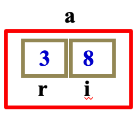
What are the advantages of using structures?

- Structures are used to represent more complex data types. For example, derived data types such as FRACTION, COMPLEX etc can be easily represented using structures.

```
typedef  struct
{
     int      n;     3
     int      d;     8
} FRACTION;

FRACTION   a;
```

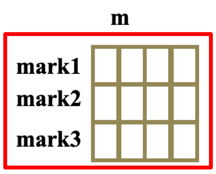



```
typedef  struct
{
     int      r;      3  +  8 i
     int      i;
} COMPLEX;

COMPLEX   a;
```



- Related data items of same data type can be logically grouped under a common name.

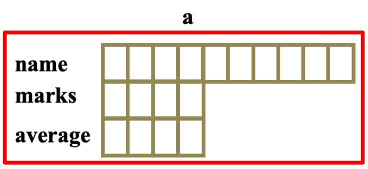
```
typedef  struct
{
     int          mark1;
     int          mark2;
     int          mark3;
} MARKS ;

MARKS          m;
```



Mark1 Mark2 Mark3

- Related data items of dissimilar data types can also be logically grouped under a common name. For example,

```
typedef  struct
{
     char    name[10];
     int     marks;
     float   average;
} STUDENT;

STUDENT     a;
```



- A function always returns a single value. When we want to return more than one value, we use structures.
- Extensively used in applications involving database management.

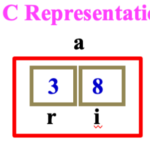
How to represent a complex number in C? A complex number 3 + 8i can be represented using structures as shown below:

```
typedef   struct
{
     int     r;
     int     i;
}  COMPLEX;
```

Mathematical Representation

3 + 8 i

C Representation



The function to read a complex number can be written as:

```
COMPLEX   read_complex  ()
{
      COMPLEX   a;
      scanf ( “%d  %d “, &a. r ,  &a.i) ;

}     return   a;
```

The function to print a complex number can be written as shown below:

```
void  print_complex ( COMPLEX   a )
{
      printf ("%d ",  a. r );

      if ( a.i  >  0 )
             printf ("+%d i ",  a. i );
      else
             printf ("%d i ",  a. i );
}
```

3 + 8 i

a: r = 3, i = 8

3 - 8 i

a: r = 3, i = -8

The program to read a complex number, print a complex number and to add two complex numbers can be written as shown below:

```
#include   <stdio.h>

typedef   struct
{
     int      r;
     int      i;
}  COMPLEX;

COMPLEX   read_complex  ()
{
     COMPLEX   a;
     scanf ( "%d   %d ", &a. r ,  &a.i) ;

     return   a;
}

void  print_complex ( COMPLEX   a )
{
     printf ("%d ",  a. r );

     if ( a.i  >  0 )
           printf ("+%d i ",  a. i );
     else
           printf ("%d i ",  a. i );
}

COMPLEX   add_complex( COMPLEX   a , COMPLEX   b )
{
     COMPLEX   c;

     c.r   =   a.r  +  b.r;
     c.i   =   a.i  +  b.i ;

     return   c;
}
```

a = 3 + 4 i
b = 4 + 2 i
c = 7 + 6 i

```
void  main ( )
{
     COMPLEX   a,  b,  c;

     printf ( "Enter complex number 1:  ");
     a  =  read_complex  ( ) ;

     printf ("Enter complex number 2:  ");
     b  =  read_complex  ( ) ;

     c  =  add_complex ( a, b ) ;

     printf (" a = ");     print_complex (a) ;
     printf (" b = ");     print_complex (b) ;
     printf (" c = ");     print_complex (c) ;
}
```


Write a C program to search for given student name in a student record consisting of name and marks

The given student called "key" has to be compared with all records as shown below:

Design

```
typedef   struct
{
     char      name[10];
     int       marks;
     char      grade;
}  STUDENT ;
```

key =   BABA

	Names	Marks	Grade
A[0]	AMITABH	100	'B'
A[1]	SACHIN	101	'B'
A[2]	ARJUN	102	'A'
A[3]	BHIM	103	'A'
A[4]	MODI	104	'A'

n = 5

The algorithm and equivalent code can be written as shown below:

Design

// Input: key, array a with n items

∀ i = 0 to   n - 1

if (key == A[i].name) return   i

return   -1

```
int   search ( char   key  [] ,  STUDENT a [] ,  int   n )
{
     int      i;

     for ( i  =   0; i   <   n;   i++ )
     {
          if ( strcmp  ( key,  a[i].name ) ==  0 )
               return   i ;
     }

     return   -1 ;
}
```

The complete C program to search for a given key in an array of student records can be written as shown below:

How to search for an item in an array of structures?

```c
#include <stdio.h >
#include <string.h>

typedef struct
{
    char    name[10];
    int     marks;
    char    grade;
} STUDENT ;

int search ( char key [], STUDENT a [], int n )
{
    int i;

    for ( i = 0; i < n; i++ )
    {
        if ( strcmp ( key, a[i].name ) == 0 )
            return i ;
    }

    return -1 ;
}
```

```c
void read_student_info ( STUDENT a [] , int n )
{
    int i;

    for ( i = 0; i < n; i++ )
    {
        scanf (" %s %d %c" , a[i].name, &a[i].marks, &a[i].grade ) ;
    }
}

void main ( )
{
    int n, pos; char key[10]; STUDENT a[10];

    printf ( "Enter number of students : " ) ;
    scanf ( " %d " , &n) ;

    printf ( "Name  Marks Grade\n" ) ;
    read_student_info (a, n ) ;

    printf ( "Enter key to search:\n" );
    scanf ( " %s", key );

    pos = search( key, a, n);

    if ( pos != -1 )
        printf ( "Successful search\n" );
    else
        printf ( "Unsuccessful search\n");
}
```

Padma Reddy A.M | Sai Vidya Institute of Technology | Bangalore | download: www.saividya.ac.in www.nandipublications.com

Write a program to print student information who got above average marks and who got below average marks separately.

The structure representation is shown below:

```c
typedef struct
{   char    name[10];
    int     marks;
} STUDENT ;
```

The student record where number of students n = 5 is shown below:

	Names	Marks
A[0]	AMITABH	100
A[1]	SACHIN	101
A[2]	ARJUN	102
A[3]	BHIM	103
A[4]	MODI	104

n = 5

The algorithm to find the average can be written as shown below:

// Input: array a with n students

sum = 0

$\boxed{\text{sum = sum + a[i].marks}}$ ∀ i = 0 to n - 1

return sum / n

The complete program to print the student info who got more than average marks and who got less than average is shown below:

How to print student details who are above and below average?

```
#include  < stdio.h >
typedef   struct
{     char      name[10];
      int       marks;
}  STUDENT ;

void  print_student_info ( STUDENT a [] ,  int  n )
{
    int        i;
    float      average;
    average  =  find_average ( a , n ) ;

    printf ( "Marks Names above %f ", average ) ;
    for ( i  =  0;  i  <  n;  i++ )
    {
        if (  a[i].marks  >  average)
            printf ( "%d   %s" , a[i].marks ,  a[i].name ) ;
    }

    printf ( "Marks Names below %f ", average ) ;
    for ( i  =  0;  i  <  n;  i++ )
    {
        if ( a[i].marks  <  average )
            printf ( "%d   %s" , a[i].marks ,  a[i].name ) ;
    }
}

void  read_student_info ( STUDENT a [] ,  int  n )
{
    int    i;
    for ( i  =  0;  i  <  n;  i++ )
        scanf ( "  %s  %d " , a[i].name, &a[i].marks ) ;
}

float  find_average( STUDENT   a[],  int   n )
{
    int      i ;
    float    sum;

    sum  =  0;
    for (  i  =  0;  i  <  n;  i++ )  sum  +=  a[ i].marks ;

    return  sum  /  n ;
}

void  main ( )
{
    int    n;      STUDENT     a[10];

    printf ( "Enter number of students : " ) ;
    scanf ( " %d " ,  &n ) ;

    printf ( "Name  Marks \n" ) ;
    read_student_info ( a,   n ) ;

    print_student_info  ( a,   n ) ;
}
```


What is union? What is the syntax for defining union?

Definition: A union is a collection of one or more declaration of variables of same data type or dissimilar data types, grouped together as a single entity.

- The variables defined inside the union are called members of the union or fields of the union.
- All members can be accessed using a common name. It is a derived data type in C.

For example, a student information to be grouped may consist of

- name of the student //array of characters
- marks scored //integer
- average marks scored //double

Syntax:

```
union
{
    type1      member 1;
    type2      member 2;
    .......    .............
};
```

Example:

```
union
{
    char      name[20];
    int       marks;
    float     double;
};
```

What is tagged union? What is the syntax?

Definition: In the union definition, the keyword **union** can be followed by an identifier.

- This identifier is called **tag name.**
- The union definition associated with tag name is called **tagged/named union.**

Syntax:

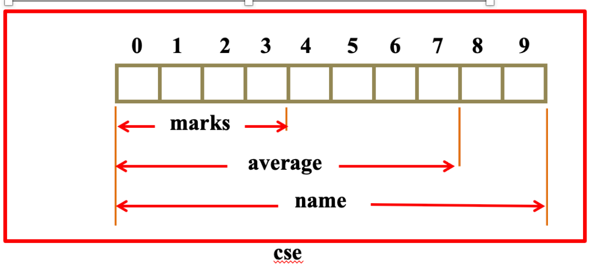
```
union   tag_name
{
      type1      member 1;
      type2      member 2;
      …….        ………….
};
union   tag_name  v1,  v2,  ….  vn;
```

Example:

```
union   student
{
      char       name[10];
      int        marks;
      double     average;
};
union   student cse;
```

How memory is allocated for union?

- A block of memory is allocated.
- The memory allocated by the compiler is large enough to hold the largest member of the union.
- So, the size of block is the size of the largest member of the union.
- All the members share the same set of memory locations.
- At any point of time only one member can be accessed and change of one member affects the other member.



Example:

```
union   student
{
      char       name[10];
      int        marks;
      double     average;
};
union   student cse;
```

What is tagless union or unnamed union?

Definition: In the union definition, the keyword **union** is not followed by an identifier.

- That means, there is no tag associated with the union.
- The union definition without tag name is called **tagless union.**
- Since, there is no name associated with keyword **union**, it is also called **name less union** or **unnamed union.**

Syntax:

```
union
{
      type1      member 1;
      type2      member 2;
      …….        ………….
}  v1,  v2,  ….  vn;
```

Example:

```
union
{
      char       name[10];
      int        marks;
      float      average;
}  cse;
```

How memory is allocated?

- A block of memory is allocated.
- The memory allocated by the compiler is large enough to hold the largest member of the union.
- So, the size of block is the size of the largest member of the union.
- All the members share the same set of memory locations.
- At any point of time only one member can be accessed and change of one member affects the other member.

Example:

What is type-defined union?

Definition: In the union definition, the keyword union is not followed by an identifier.

- It is a type of tagless union. But, it is preceded by a keyword typedef.
- The union definition with keyword typedef is called type-defined union. The type-defined union must be followed by an identifier ending with semicolon.
- This identifier acts as a data type. Using this type defined union we can declare variables.

Syntax:

```
typedef    union
{
     type1    member 1;
     type2    member 2;
     .......  .............
} TYPE_ID;

TYPE_ID     v1,  v2,  ....  vn;
```

Example:

```
typedef    union
{
     char     name[10];
     int      marks;
     float    average;
} STUDENT;

STUDENT      cse;
```

How memory is allocated?

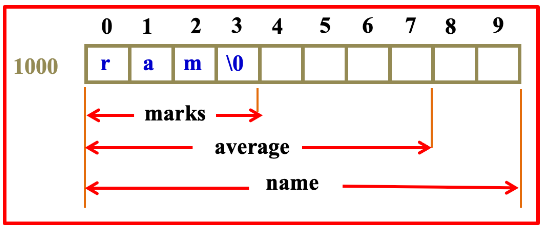
- A block of memory is allocated.
- The memory allocated by the compiler is large enough to hold the largest member of the union.
- So, the size of block is the size of the largest member of the union.
- All the members share the same set of memory locations.
- At any point of time only one member can be accessed and change of one member affects the other member.

Example:

How to initialize the members of union?

Method 1: Tagless union initialization

```
union
{
     char      name[10];
     int       marks;
     double    average;
} cse  =  {"ram",   25,  24.5  };
```



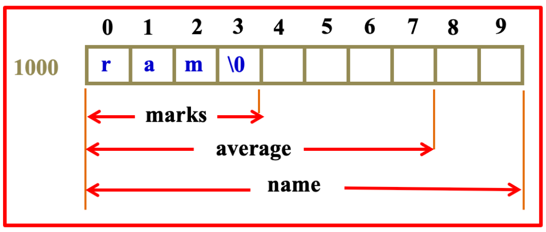
cse

Note:
- Only the first member of union can be initialized.
- It is not possible to initialize subsequent members of union

How to initialize the members of union?

Method 2: Tagged union initialization

```
union   student
{
     char      name[10];
     int       marks;
     double    average;
};

union   student  a  =  {"ram", 25,  24.5  };
```

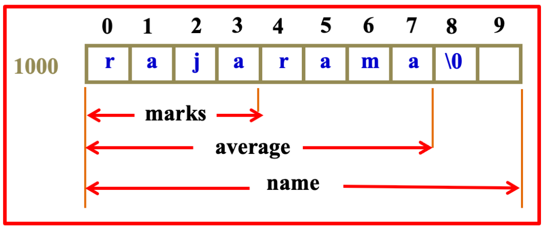


cse

How to initialize the members of union?

Method 3: Type-defined union initialization

```
typedef struct
{
     char    name[10];
     int     marks;
     double  average;
} STUDENT;

STUDENT a = { "rajarama", 25, 24.5};
```

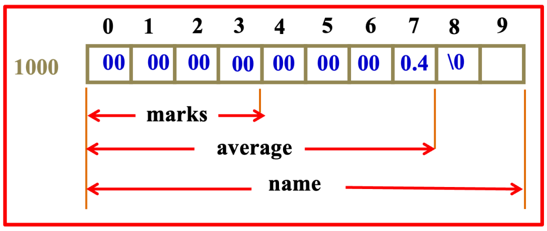


cse

How to access the members of union?

Method 3: Type-defined union initialization

```
typedef struct
{
     char    name[10];
     int     marks;
     double  average;
} STUDENT;

STUDENT a = { "rajarama", 25, 24.5};
```



cse

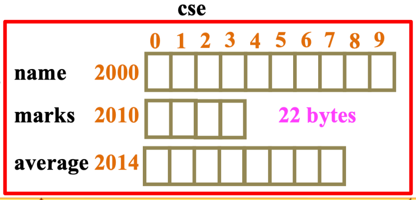
```
printf ("%s", a .name);      // rajarama
a .marks = 25;
printf ("%d", a .marks );    // 25
a .average = 0.4;
printf ("%f", a.average );   // 0.4
```

What are the differences between structures and unions?

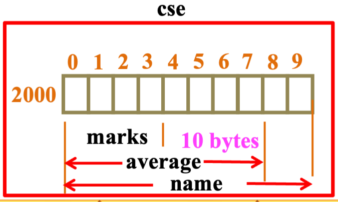
Separate memory locations are allocated for every member of the structure.	The memory is allocated and its size is equal to maximum size of a member.
Each member within a structure is assigned unique address	The address is same for all members
The address of each member is greater than the address of its previous member	The address is same for all members
Altering the value of one member will not affect other other members of the structure	Altering the value of one member affects other member as the memory is shared.
Several members of a structure can be initialized	Only the first member of the union can be initialized.
Size of structure is >= sum of sizes of its members. (Greater because of slack bytes)	Size of union is = size of largest member

```
struct
{
  char    name[10];
  int     marks;
  double  average;
} cse;
```

cse



cse




```
union
{
  char    name[10];
  int     marks;
  double  average;
} cse;
```

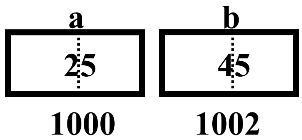
Chapter 12: Pointers

What are we studying in this chapter?

- Pointers and address
- Pointers and function arguments
- pointers and arrays, address arithmetic
- character pointer and functions
- Pointer to pointer , Initialization of pointer arrays
- Understanding complex declarations
- dynamic allocation methods
- Array of pointers and programming examples. - 7 hours

Example 12.1: Program to print the values using variables and their addresses

```
#include <stdio.h>
void main()
{
        int     a = 25;
        int     b = 45;

        /* Accessing the data using variables */
        printf("Value of a = %d\n", a);
        printf("Value of b = %d\n", b);

        /* Accessing the address of variables */
        printf("Address of a = %d\n", &a);
        printf("Address of b = %d\n", &b);
        /* Accessing the data using de-referencing operator */
        printf("Value of a = %d\n", *&a);
        printf("Value of b = %d\n", *&b);
}
```




```
Value of a = 25
Value of b = 45

Address of a = 1000
Address of b = 1002

Value of a = 25
Value of b = 45
```

Note: Observe that the operator pair *& gets cancelled each other. So,

- *&a is same as *a*
- *&b is same as *b*

Now, the question is *"Is it possible to store the address of a variable into memory?".* Yes, it is possible. As we store the data using *assignment operator,* we can store address of variable using assignment operator as shown below:

```
p = &a;     // Now, the variable p contains address of variable a
x = &b;     // Now, the variable x contains address of variable b
```

Note: Please see that, the variables *p* and *x* in above two statements are not normal variables, as they do not contain the data. Instead the variables *p* and *x* contain addresses of the data. These variables *p* and *x* which contain the addresses are called *pointers* or *pointer variables.*

Now, once we know what are pointer variables, the next question is *"How to declare the pointer variables?"* It is very simple and can be done as shown below:

If a variable *p* contains address of *int variable* its declaration is:	**int**	*p;
If a variable *x* contains address of *float variable* its declaration is:	**float**	*x;
If a variable *y* contains address of *char variable* its declaration is:	**char**	*y;
If a variable *z* contains address of *double variable* its declaration is:	**double**	*z;

Note: *The address operator can be used with any variable that can be placed on the left side of an assignment operator.* Since constants, expressions and array names cannot be used on the left hand side of the assignment and hence accessing address is invalid for constants, expressions and array names.   The following are invalid:

Usage	Valid/Invalid	Reasons for invalidity
&100	Invalid	Address of a constant cannot be obtained
&(p + 10)	Invalid	Address of an expression cannot be obtained
&(p + q)	Invalid	Address of an expression cannot be obtained
int a[10]; &a	Invalid	Address of entire array cannot be obtained
register a; &a	Invalid	Address of a register variable cannot be obtained

Definition: A variable that contains the address of another variable or address of a memory location is called a *pointer*. A *pointer* is also called a *pointer variable.*

Once we know the concept of pointers, let us see "What are the steps to be followed to use pointers?" The following sequence of steps have to be followed by the programmer:

Steps to be followed while using pointers:

- Declare a data variable — **Ex: int a;**
- Declare a pointer variable — **Ex: int *p;**
- Initialize a pointer variable — **Ex: p = &a;**
- Access data using pointer variable — **Ex: printf("%d",*p);**

- The variables along with pointer variables have to be declared in the beginning of a function. These declarations can be in any order.
- Only point we have to remember is that before using pointers to access anything, the pointers have to be initialized with appropriate addresses.

12.2.1 Pointer declaration and Definition

In C language, we know that all the variables should be declared before they are used. Pointer variables also should be declared before they are used. In this section, let us see "How to declare pointer variables?" The syntax to declare a pointer variable is shown below:

type * **identifier** ;

- identifier → Name given to the pointer variable
- * → The asterisk (*) in between *type* and *identifier* tells that the *identifier* is a pointer variable
- type → **type** can be any data type such as **int**, **float**, **char** etc. It can be derived/user-defined data type also.

For example,

- If a variable *p* contains address of *int variable*, its declaration is: **int** *p;
- If a variable *x* contains address of *float variable*, its declaration is: **float** *x;
- If a variable *y* contains address of *char variable*, its declaration is: **char** *y;
- If a variable *z* contains address of *double variable*, its declaration is: **double** *z;
- If a variable *fp* contains address of FILE variable, its declaration is: FILE *fp;

Note: In the above declarations we say:

1) *p* is a pointer to an **int**
2) *x* is a pointer to a **float**
3) *y* is a pointer to a **char**
4) *z* is a pointer to a **double**
5) *fp* is a pointer to FILE (details are in FILE HANDLING: CHAPTER 14)

Example 12.2: In the declaration, the position of * is immaterial. For example, all the following declarations are same:

```
int    *pa;
int  *  pa;
int*    pa;
```

Any of the above declaration informs that the variable *pa* is a pointer variable and it should contain address of integer variable.

Example 12.3: Consider the multiple declarations as shown below:

```
int*    pa, pb, pc;
```

Observe the following points:

- In the above declaration, most of the readers *wrongly assume* that the variables *pa, pb* and *pc* are pointer variables. This is because * is attached to **int.**
- This assumption is wrong. Only *pa* is a pointer variable, whereas the variables *pb* and *pc* are ordinary integer variables.
- For better readability, the above declaration can be written as shown below:

  ```
  int     *pa, pb, pc;
  ```

  Now, we can easily say that *pa* is pointer variable because of * operator, whereas *pb* and *pc* are integer variables and are not pointer variables.
- It is still better if the variables are declared in separate lines as shown below:

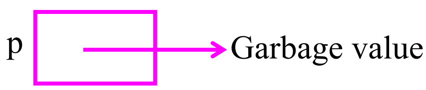
  ```
  int     *pa;
  int     pb;
  int     pc;
  ```

12.2.2 Dangling pointers

In the previous section, we have seen the method of declaring a pointer variable. For example, consider the following declaration:

```
int     *p;
```

This indicates that *p* is a pointer variable and the corresponding memory location should contain address of an integer variable. But, the declaration will not initialize the memory location and memory contains garbage value as shown below:



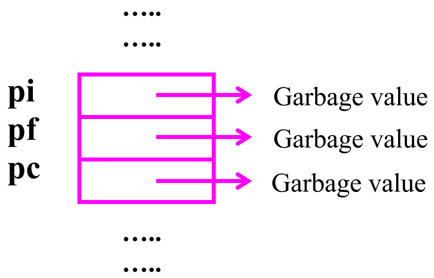
Here, the pointer variable *p* does not contain a valid address and we say that it is a dangling pointer. Now, let us see "What is a dangling pointer?"

Definition: A pointer variable which does not contain a valid address is called *dangling pointer.*

Example 12.4: Consider following declarations and assume all are *local variables.*

```
int     *pi;          /* Pointer to an integer */
float   *pf;          /* Pointer to a float number */
char    *pc;          /* Pointer to a character */
```

- The local variables are not initialized by the compiler during compilation. This is because, the local variables are created and used only during execution time.
- The pointer variables also will not be initialized and hence they normally contain some garbage values and hence are called dangling pointers.
- The memory organization is shown below:



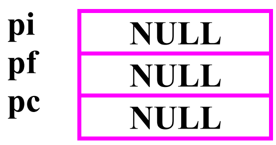
The pointer variables **pi**, **pf** and **pc** does not contain valid addresses and hence they are **dangling pointers.**

Note: Most of the errors in programming are due to un-initialized pointers. These errors are very difficult to debug. So, it is the responsibility of the programmer to avoid dangling pointers. Hence, it is necessary to initialize the pointer variables so that they always contain valid addresses.

Example 12.5: Consider following declarations and assume all are *global variables.*

```
int     *pi;          /* Pointer to an integer */
float   *pf;          /* Pointer to a float number */
char    *pc;          /* Pointer to a character */
```

All global variables are initialized by the compiler during compilation. The pointer variables are initialized to NULL indicating they do not point to any memory locations as shown below:



12.2.4 Initializing a pointer variable

Now, the question is *"How to initialize a pointer variable?"* Initialization of a pointer variable is the process of assigning the address of a variable to a pointer variable. The initialization of a pointer variable can be done using following three steps:

- **Step 1:** Declare a data variable
- **Step 2:** Declare a pointer variable
- **Step 3:** Assign address of a data variable to pointer variable using & operator and assignment operator

Note that the steps 1 and 2 can be interchanged i.e., we can first declare a pointer variable, then declare a data variable and then initialize the pointer variable. The three ways using which initialization can be done is described below:

Method 1: *Declaring a data variable, pointer variable and initializing pointer variable in separate statements.* For example, consider the following three statements:

```
int     x;        /* Step 1: x is declared as an integer data variable */
int     *px;      /* Step 2: px is declared as a pointer variable */
px = & x;         /* Step 3: copy address of data variable to pointer variable */
```

Method 2:*Declaring a pointer and initializing a pointer in a single statement:* Using this method, the above three statements can be written as shown below:

```
int     x;
int     *px = &x;
```

Method 3: *Declaring a data variable, pointer variable and initializing a pointer variable in a single statement:* Using this method, the above two statements can be written as shown below:

```
int     x, *px = &x;
```

Example 12.6: Consider the following statements:

```
int     p, *ip;
float   d, f;

ip = p;         /* ERROR: The ip should contain address of a variable */
ip = &d;        /* ERROR: ip is pointer to int. But, it contains address of
                   double variable */
```

```
ip = &p;        /* OK */
```

Observe the following points:

- Consider the first statement:

  ```
  ip = p;
  ```

  Here, *ip* is a pointer to integer. It should contain the address. But, we are not storing the address. Hence, *it is an error.*

- Consider the second statement:

  ```
  ip = &d;
  ```

  Here, *ip* should contain address of integer variable. But, we are storing address of **float** variable. So, *it results in error.*

Now, let us write some programs using two pointers

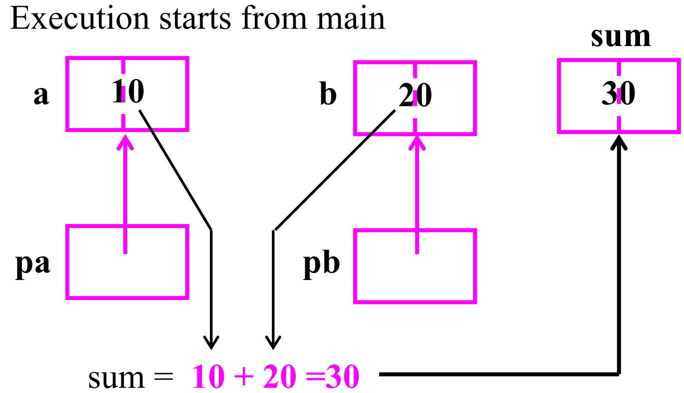
Example 12.7: Write a program to add two numbers using pointers

PROGRAM

```
#include <stdio.h>

void main()
{
      int a = 10, b = 20, sum;

      int *pa, *pb;

      pa = &a;
      pb = &b;

      sum = *pa + *pb;


      printf("Sum = %d\n", sum);
}
```

TRACING

Execution starts from main



Output

Sum = 30

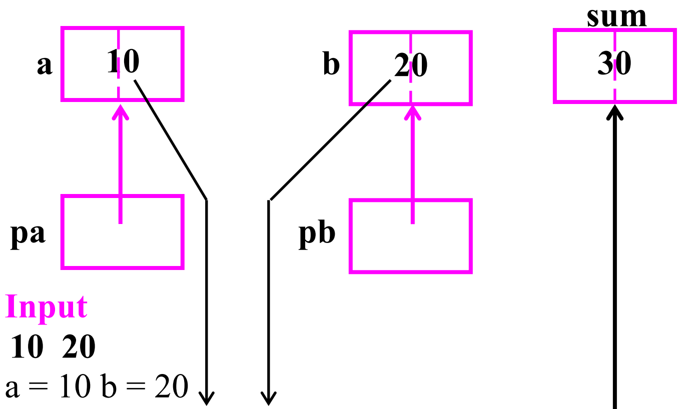
Example 12.8: Program to read two numbers and add two numbers using pointers

PROGRAM

```
1. #include <stdio.h>
2.
3. void main()
4.{
5.      int a, b, sum;
6
7.      int *pa, *pb;
8.
9.      pa = &a;
10.     pb = &b;
11.
12.     scanf("%d %d",&a, &b);
13.
14.     sum = *pa + *pb;
15.
16.     printf("Sum = %d\n", sum);
17.}
```

TRACING

Execution starts from main



Input

10 20

a = 10 b = 20

sum = **10 + 20 =30**

Output

Sum = 30

Note: After executing statement 12, the values 10 and 20 which are read from the keyboard are copied into memory locations identified by *a* and *b*. Then those values are accessed using pointer variables *pa* and *pb*, added and result is stored in the variable *sum*.

Note: In the statement in line 12 i.e., **scanf**("%d %d", **&a, &b**);

we are using &a and &b. In line 9 and 10, &a and &b are already copied into pointer variables *pa* and *pb*. So, in place of &a and &b, we can use the pointer variables *pa* and *pb* as shown below:

```
scanf("%d %d", pa, pb );    /* Since pa contains &a, pb contains &b */
```

Note: there is no need of writing &pa and &pb, since *pa* and *pb* already contains the addresses.

12.3 Pointers are flexible

The pointers are very flexible and can be used in variety of situations as shown below:

- A pointer can point to different memory locations
- Two or more pointers can point to same memory location

- Altering functional arguments using pointers (Pointers and function arguments)
- Functions returning pointers
- Pointers to pointers
- Arrays and pointers
- Pointer can point to a single dimensional array
- Arrays of pointers
- Pointer can point to a function

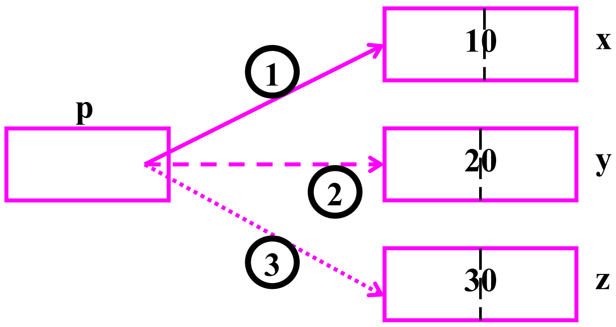
12.3.1 A pointer pointing to different memory locations

A pointer can point to different data variables by storing the address of appropriate variables. This can be explained using the following program segment:

```
int     x = 10, y = 20, z = 30;
int     *p;
```

Output

```
1   p = &x;
    printf("%d\n", *p);     //10

2   p = &y;
    printf("%d\n", *p);     //20

3   p = &z;
    printf("%d\n", *p);     //30
```



Observe the following points:

1. The pointer variable *p* contains the address of *x*. So, output is 10
2. The pointer variable *p* contains the address of *y*. So, output is 20
3. The pointer variable *p* contains the address of *y*. So, output is 30

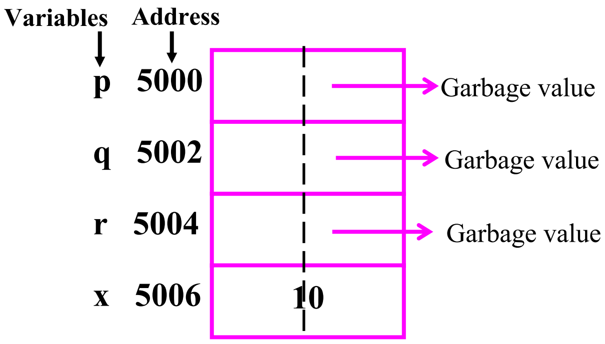
It is observed from above example that the pointer variable *p* points to different memory locations by storing the addresses of different variables. But, at any point of time, *p* points to only one memory location. Thus, same pointer can be pointed to different data variables.

12.3.2 Two or more pointers can point to same memory locations

Consider the statements shown below:

1

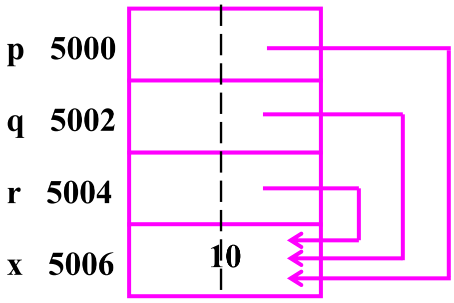
```
int     *p;

int     *q;
```



```
int     *r;
int     x = 10;
```

2

```
p = &x;

q = &x;

r = &x;
```



3

printf("&p =%u, p = %u, *p = %d\n",&p, p, *p); /* **Output** */
&p = 5000, p = 5006, *p = 10

printf("&q =%u, q = %u, *q = %d\n",&q, q, *q); /* **Output** */
&q = 5002, q = 5006, *q = 10

printf("&r =%u, r = %u, *r = %d\n",&r, r, *r); /* **Output** */
&r = 5004, r = 5006, *r = 10

Observe the following points from the above program segment:

- 1 In the first set of instructions, memory is allocated for all pointer variables but the pointers are not initialized. Hence, they contain garbage values and hence they are called dangling pointers. Only the variable *x* is initialized.
- 2 After executing the second set of statements, the pointer variables *p*, *q* and *r* contains the address of integer variable *x* and logical representation is shown in above figure.
- 3 After executing the third set of instructions, even though various pointers have different addressed, all of them points to same set of memory locations. So, the output is 10.

Note: Even though the variables **p**, **q** and **r** have different addresses, they contain address of **x** only. So, different pointer variables (**p**, **q** and **r** in this example) contain address of one variable (**x** in this example). So, the value of **x** can be accessed and changed using the variables **p**, **q**, **r** and **x**. In general, there can be multiple pointers to a variable.

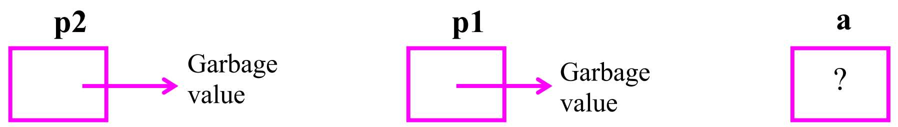
12.4.2 Pointers to Pointers

We have used pointers which directly points to data. In this section, let us see "What is pointer to a pointer?"

Definition: A variable which contains address of a pointer variable is called *pointer to a pointer*. For example, consider the following declarations:

```
int     a;
int     *p1;
int     **p2;
```

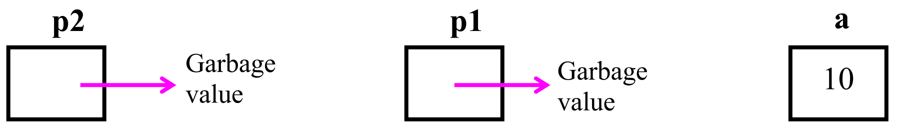
- The first declaration instructs the compiler to allocate the memory for the variable *a* in which integer data can be stored.
- The second declaration tells the compiler to allocate a memory for the variable *p1* in which address of an integer variable can be stored.
- The third declaration tells the compiler to allocate a memory for the variable *p2* in which address of a pointer variable which points to an integer can be stored. The memory organization for the above three declarations is shown below:



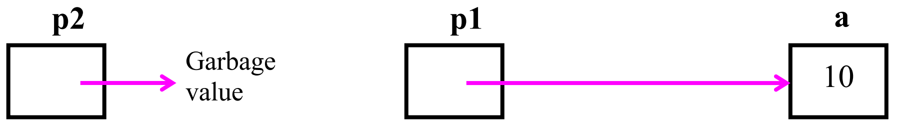
Assume the above declarations are followed by the following assignment statements:

```
a = 10;
p1 = &a;
p2 = &p1;
```

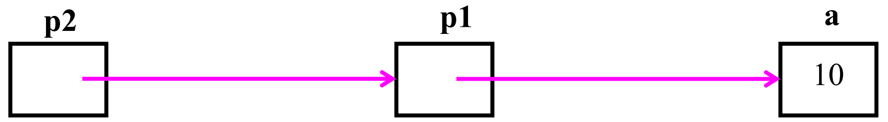
The memory organization after executing the statement **a = 10** is shown below:



The memory organization after executing the statement **p1 = &a** is shown below:



The memory organization after executing the statement **p2 = &p1** is shown below:



12.12 Pointers

The data item 10 can be accessed using three variables *a*, *p1* and *p2* are shown below:

a refers to the data item 10.

***p1** refers to the data item 10. Here, using *p1* and one indirection operator, the data item 10 can be accessed.

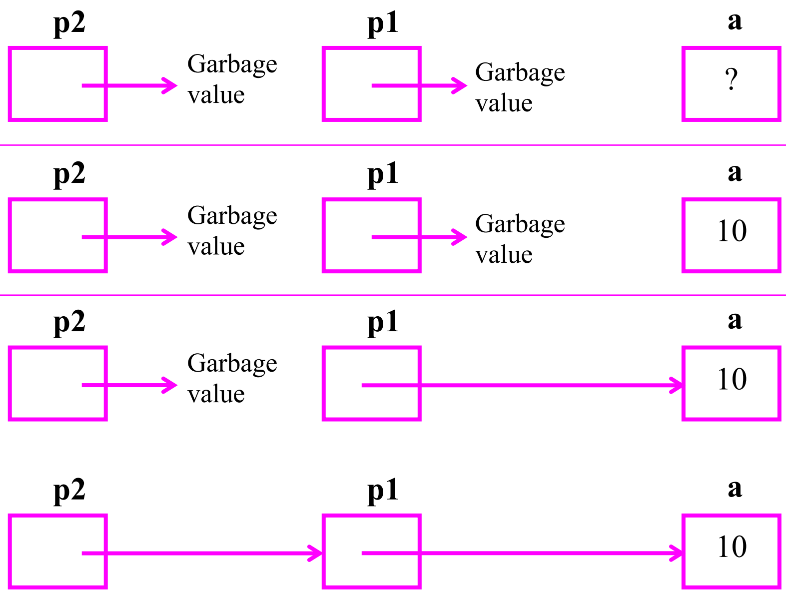
****p2** refers to the data item 10. Here, using *p2* and two indirection operators the data item 10 can be accessed (i.e., *p2 refers to p1 and **p2 refers to **a**)

The following program illustrates the way the data item 10 can be accessed using the variable **a**, using a pointer variable **p1** and pointer to a pointer variable **p2**.

Example 12.14: Program to access 10, using a variable, pointer variable and pointer to a pointer variable

```
#include <stdio.h>

void main()
{
        int     a;
        int     *p1;
        int     **p2;

        a = 10;

        p1 = &a;

        p2 = &p1;

        printf("a = %d\n",a);
        printf("*p1 = %d\n", *p1);
        printf("**p2 = %d\n", **p2);
}
```

TRACING



Output

```
a = 10
*p1 = 10
**p2 = 10
```

Note: If x is declared as integer, which of the following statements is true and which is false?

a. The expression ***&x** and **x** are the same. // it is true
b. The expression ***&x** and **&*x** are the same. // it is false
⇩
illegal

Example 12.15: Given the following declarations:

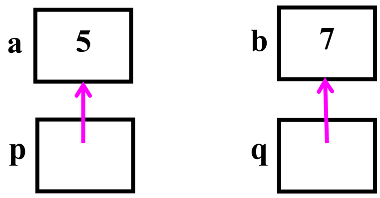
```
int     a = 5;
int     b = 7;
int     *p = &a;
int     *q = &b;
```

What is the value of each of the following expressions?

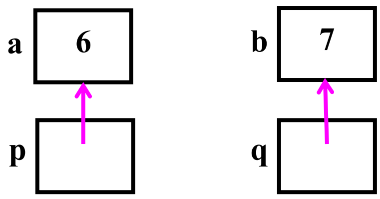
a. ++a;
b. ++(*p);
c. – – (*q);
d. – –b;

Solution: The tracing of the above program segment is shown below:

```
/* Memory representation for the declarations */
int     a = 5;
int     b = 7;
int     *p = &a;
int     *q = &b;
```

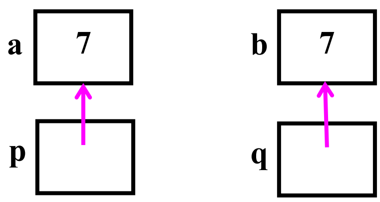



```
/* Increment a */
 ++a;
```



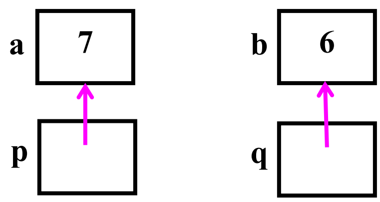
/* Access **a** using pointer variable **p** */

```
++(*p);
```

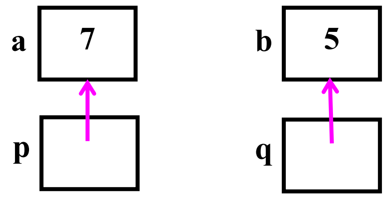


/* Access ***b*** using pointer variable **q***/

```
--(*q);
```



/* Decrement b */
--b;



Output **a = 7** **b = 5**
***p = 7** ***q = 5**

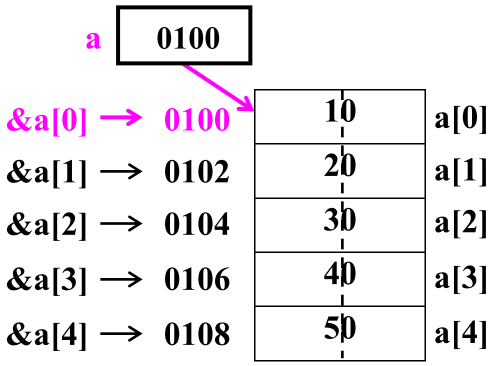
12.5 Arrays and pointers

Consider the following declaration:

int a[5] = {10, 20, 30, 40, 50};

Observe the following points:

- The compiler treats the array *a* as a pointer and memory is allocated for variable *a*.
- It then allocates 5 memory locations and address of the first memory location (say 0100) is copied into pointer variable *a* as shown in the diagram.
- The compiler then initializes all five memory locations with values 10, 20, 30, 40 and 50 respectively as shown in figure below:



Note: Assuming size of integer is 2 bytes, two bytes are reserved for each memory location

Note: The starting address of the first byte of the array is called *base address which is 0100.*

Note: The address of the 0^{th} memory location **0100** stored in ***a*** cannot be changed. So, even though ***a*** contains an address, since its value cannot be changed, *we call* ***a*** *as pointer constant.* Observe that **&a[0]** and ***a*** are same.

$$\mathbf{a} \xleftrightarrow[\text{same}]{} \mathbf{\&a[0]} \xleftrightarrow[\text{same}]{} \mathbf{(a + 0)}$$

To justify above points, now let us see "What is the output of the following program?"

PROGRAM | **TRACING**

```
#include <stdio.h>

void main()
{
        int     a[5] = {10, 20, 30, 40, 50};

        printf("%u %u %u\n", &a[0],  a,  a+0);
}
```

Output

0100   0100 0100

Note: We may get different answer in our computer. But, whatever it is, observe that the value of ***&a[0]*** or ***a*** or ***a+0*** are same.

Now, let us see "How to access the address of each element?" The address of each item can be accessed using two different ways:

address operator with index	base address with index
&a[**0**] i.e, 1000	(a+**0**) i.e., 0100
&a[**1**] i.e., 1002	(a+**1**) i.e., 0102
&a[**2**] i.e., 1004	(a+**2**) i.e., 0104
&a[**3**] i.e., 1006	(a+**3**) i.e., 0106
&a[**4**] i.e., 1008	(a+**4**) i.e., 0108

In general,

&a[**i**] ⟷ **is same as** (a +**i**) where i = 0 to 4
0 to 5-1
0 to n – 1 (in general)

Note: The various ways of accessing the address of **i**th item in an array **a** is shown below:

&a[i] is same as **a + i**

↕ same ↕ same

&i[a] is same as **i + a**

So, address of a[i] can be obtained using any of the following notations:

&a[i] or **a+i** or **i+a** or **&i[a]**

The data in those addresses can be obtained using the indirection operator * as shown below:

***&a[i]** or ***(a+i)** or ***(i+a)** or ***&i[a]**

a[i] **i[a]**

Note: The pair *& get cancelled each other

So, *(**a+i**) or *(**i+a**) or **a[i]** or **i[a]** or **a[i]** or **i[a]** are one and the same.

To justify this answer, consider the following program:

```
#include <stdio.h>

void main()
{
        int     a[5] = {10, 20, 30, 40, 50};
        int     i = 3;

        printf("%d %d %d %d %d %d \n", *(&a[i]), a[i], *(a+i), *(i+a), i[a], *&i[a]);
}
```

a = 0100

&a[0] →	**0100**	10	**a[0]**
&a[1] →	**0102**	20	**a[1]**
&a[2] →	**0104**	30	**a[2]**
&a[3] →	**0106**	40	**a[3]**
&a[4] →	**0108**	50	**a[4]**

Output **40** **40** **40** **40** **40** **40**

Note: It is observed from the above example that: a[i] is same as *(a+i) denoted using pointer concept. So, any array program can be written using pointers.

12.5.1 Largest of N numbers

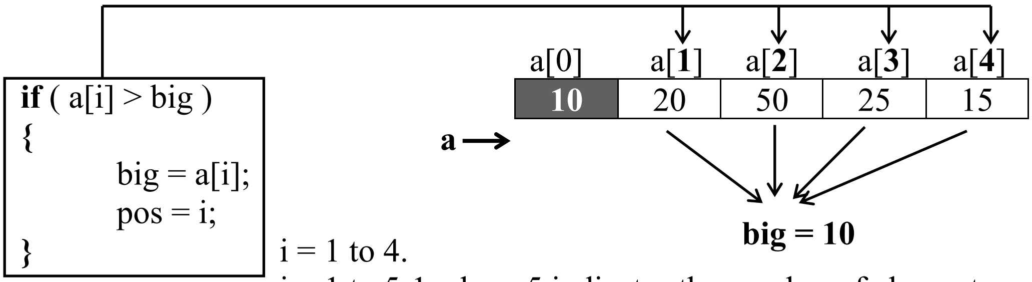
Consider 5 elements 10, 20, 50, 25 and 15. It is required to find the largest of these 5 numbers. Now, let us see "How to write the program to find largest of N numbers?"

Design: Assume the variable ***big*** contains **10** which is the 0th element of the array and ***pos*** is 0 which is the position of that element. The equivalent code can be written as:

```
big = a[0];     /* Assume first item is big */
pos = 0;        /* Store 0 as the position of 0th item */
```

Initialization

Since 0th item **10** is in **big**, the rest of the items such as **a[1]**, **a[2]**, **a[3]** and **a[4]** should be compared with **big** as shown in figure:



i = 1 to 4.
i = 1 to 5-1 where 5 indicates the number of elements
In general, i = 1 to n-1 where n = 5 indicates the number of elements.

So, the code can be written as shown below:

```
for ( i = 1; i <= n-1; i++)
{
        if  ( a[i] > big )
        {
                big = a[i];
                pos = i;
        }
}
```

Main logic

Note: When we know the program using arrays, we can easily write the program using pointers. We have seen that **a[i]** is same as ***(a+i) or *(i+a) or i[a].** So, *replace **a[i]** by ***(a+i)** to get the program using pointers.*

Now, the complete program to find the largest of N elements *using an array* and *using pointer with indexing* is shown below:

Example 12.16: Program to compute largest and its position

Using Arrays

```
#include <stdio.h>

void main()
{
    int  a[10], n, i, big, pos;

    printf("Enter number of elements\n");
    scanf("%d",&n);

    printf("Enter the elements\n");
    for ( i = 0; i <= n-1; i++)
        scanf("%d",&a[i]);

    big = a[0];
    pos = 0;

    for ( i = 1; i <= n-1; i++)
    {
        if ( a[i] > big )
        {
                big = a[i];
                pos = i;
        }
    }

    printf("Largest = %d\n",big);
    printf("Position = %d\n", pos+1);
}
```

Using Pointer with indexing

```
#include <stdio.h>

void main()
{
    int  a[10], n, i, big, pos;

    printf("Enter number of elements\n");
    scanf("%d",&n);

    printf("Enter the elements\n");
    for ( i = 0; i <= n-1; i++)
        scanf("%d",a+i);   /* (a+i) = &a[i] */

    big = *(a+0);
    pos = 0;

    for ( i = 1; i <= n-1; i++)
    {
        if ( *(a+i) > big )
        {
            big = *(a+i);
            pos = i;
        }
    }

    printf("Largest = %d\n",big);
    printf("Position = %d\n", pos+1);
}
```

12.5.2 Pointers and other operators

Like normal variables in an expression, pointer variables in expressions can also be used. If **p1** and **p2** are pointer variables that are declared and initialized properly, ***p1** and ***p2** represent the values to be manipulated. So, operations such as relational, arithmetic, logical etc., can be performed on ***p1** and ***p2.**

Example 12.17: Valid statements with operations such as multiplication and addition

```
x = *p1 *  *p2;        // The values pointed by p1 and p2 are multiplied
sum = sum + *p1;       // The value pointed to by p1 is added to sum
*p1 = *p1 + 1;         // value pointed to by p1 is incremented by 1
```

- x = *p1 /*p2; // **Error:** This expression is wrong because /* before p2 // is treated as beginning of the comment in C

Note: The error in the above statement can be eliminated by inserting the space between / and * as shown below:

- x = *p1 / *p2 // **Correct:** Since there exists space between / and *, it is // treated as division operation not as beginning of the // comment

Note: Even though various operations can be performed on ***p1** and ***p2** (since they represent the values to be manipulated), the operations are restricted on **p1** and **p2** since they contain only the addresses. The various operations that can be performed on pointer variables are shown below:

Operations performed on pointers
- → Adding an integer to a pointer
- → Subtracting an integer from pointer
- → Subtracting two pointers
- → Comparing two pointers

12.5.3 Adding an integer to a pointer

An integer can be added to a pointer. This can be explained using the following example.

Example 12.18: Consider the following declaration:

```
int     a[5] = {10, 20, 30, 40, 50};
int     *p1, *p2;
p1 = a;
p2 = a;
```

The various valid and invalid statements are shown below:

- p1 = p1 + 1; /* **Valid: Points to next element** */
- p1 = p1 + 3; /* **Valid**: Points to 3rd element from p1*/
- p1 + p2; /* **Invalid:** Two pointers cannot be added */
- p1++; /* **Valid:** Same as p1 = p1 + 1 */

Example 12.19: Pointer arithmetic using increment operator

```
int     a[5]={10, 20, 30, 40, 50};
int     *p;

p = a;  /* p points to a */
```

Assuming base address of **a is 0100,** the variable **p** points to first item as shown below:

	a ↘	
p ⟶	**0100**	10
	0102	20
	0104	30
	0106	40
	0108	50

After executing the statement:

p++;

the pointer variable **p** points to the next integer.

Note: Each time p++ is executed, its value will be incremented by 2 because size of integer is 2 bytes. In other words, **p** points to the next item.

```
float   a[5]={10.555, 20, 30, 40, 50};
float   *p;

p = a;  /* p points to a */
```

Assuming base address of **a is 0100,** the variable **p** points to first item as shown below:

	a ↘	
p ⟶	**0100**	10.555
	0104	20.0
	0108	30.0
	0112	40.0
	0116	50.0

After executing the statement:

p++;

the pointer variable **p** points to the next floating point number.

Note: Each time p++ is executed, its value will be incremented by 4 because size of floating point number is 4 bytes. In other words, **p** points to the next item.

Note: In general, if **p** is a pointer variable pointing to an array, after executing the statement:

p++;

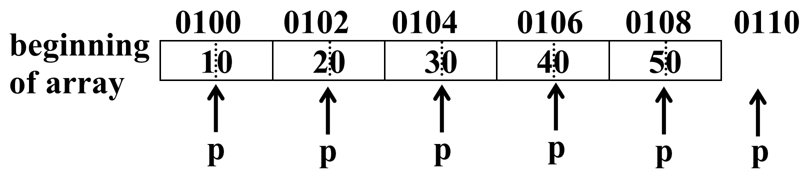
the pointer variable is incremented by:

- 1 for character array
- 2 for integer array
- 4 for floating point array and 8 for double and so on. In other words, the pointer points to the next item of an array.

12.5.4 Display array elements using pointers

Now, let us see "How to write a program to display array elements using pointer?"

Design: Consider the following array and assume **p** points to the beginning of the array. To start with **p** points to **0100** and ***p** refers to **10.** Let us observe the outputs in various iterations shown below:



Iterations: i = 0   1   2   3   4

print 10 and increment p using
```
printf("%d\n", *p);        // 10
p++;                       // 0102
```
print 20 and increment p using
```
printf("%d\n", *p);        // 20
p++;                       // 0104
```
print 30 and increment p using
```
printf("%d\n", *p);        // 30
p++;                       // 0106
```
print 40 and increment p using
```
printf("%d\n", *p);        // 40
p++;                       // 0108
```
print 50 and increment p using
```
printf("%d\n", *p); q      // 50
p++;                       // 0110
```

In general, observe that the following two statements:

```
printf("%d\n", *p);
p++;
```

are repeatedly executed for i = 0 to 4, to get the output **10**, **20**, **30**, **40** and **50**. The C equivalent statement using for loop is shown below:

```
for (i = 0; i <= 4; i++)        /* i <= 4 is same as i < 5 */
{
        printf("%d\n", *p);
        p++;
}
```

The complete program is shown below:

Example 12.20: Program to display array elements using pointer

```
#include <stdio.h>

void main()
{
        int     a[] = {10, 20, 30, 40, 50 };
        int     *p;
        int     i;

        p = a;          /* same as p = &a[0] */

        for ( i = 0; i <= 4; i++)
        {
                printf("%d  ",*p);
                p++;
        }
        printf("\n");
}
```

12.5.5 Sum of N numbers using pointers

In the previous example, instead of printf() within the for loop, if we use the statement

sum = sum + *p;

then we add all the elements of the array. The complete program to add *n* elements is shown below:

Example 12.21: Program to compute sum of elements of array

```
#include <stdio.h>

void main()
{
        int     a[] = {10, 20, 30, 40, 50 };
        int     *p;
        int     i, sum;

        p = a;          /* point p to the first element */
```

```
    sum = 0;        /* Initialize sum to 0 */

    for ( i = 0; i <= 4; i++)
    {
        sum = sum + *p;    } /* Same as */
        p++;               } sum = sum + *p++; or sum = sum + *(p++);
    }

    printf("Sum of all the numbers = %d\n",sum);
}
```

Note: Observe that by executing **p++**, we can point **p** to the next element. On similar lines by executing **p--,** we can point **p** to the previous element in an array.

12.5.6 Subtracting an integer from a pointer

Subtraction can be performed when *first operand is a pointer* and the *second operand is an integer.* This can be explained by considering the following example.

Example 12.22: Consider the following declaration and initialization:

```
    int     a[5] = {10, 20, 30, 40, 50};
    int     *p1;
    p1 = &a[4];
```

The various valid and invalid statements are shown below:

- p1 = p1 – 1;   /* **Valid** */
- p1 = p1 – 3;   /* **Valid** */
- p1--;   /* **Valid:** Same as p1 = p1 – 1 */
- --p1;   /* **Valid:** Same as p1 = p1 – 1 */
- p1 = 1 – p1;   /* **Invalid:** The first operand should be a pointer

---

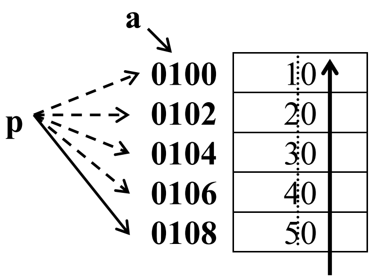
Example 12.23: Write a program to display array elements using pointer from last element to first element.

Note: As we execute p++, pointer variable **p** points to next element, if we execute p--, pointer variable **p** points to the previous element.

Design: To get the array elements in reverse order, point the variable **p** to point to the end of the array and replace p++ by p– – in the previous program. The complete program is shown below:

```
#include <stdio.h>
```

```
void main()
{
        int     a[] = {10, 20, 30, 40, 50 };
        int     *p;
        int     i;

        p = &a[4];    /* point p to the last element*/

        for ( i = 0; i <= 4; i++)
        {
                printf("%d ",*p);
                p--;
        }
        printf("\n");
}
```



Output

p points to previous element after executing p-- and items are accessed from bottom to top.

12.5.7 Subtracting two pointers

If two pointers are associated with the same array, then subtraction of two pointers is allowed. But, if the two pointers are associated with different arrays, even though subtraction of two pointers is allowed, the result is meaningless.

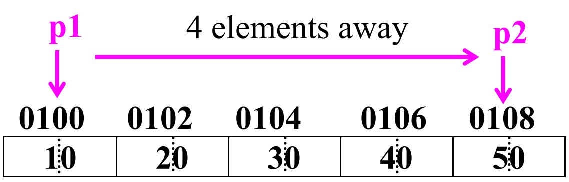
Consider the following declaration and initialization:

```
int     a[5] = {10, 20, 30, 40, 50};
int     *p1;
int     *p2;
float   *f;
p1 = a;         /* same as p1 = &a[0] */
p2 = &a[4];
```

The various valid and invalid statements are shown below:

- p2 – p1; /* **Valid** */
- p1 – p2; /* **Valid** */
- f – p1; /* **Invalid:** Since type of both operands is not same */

The memory map for the above declaration is shown below:



Note: Observe following facts from above figure:

- **p2** has an address 0108 and **p1** has address 0100.
- But, **p2 – p1 is not 0108-0100.** Actually, it is (0108-0100)/sizeof(int) i.e., (0108-0100)/2 = 4
- So, **p2 – p1** gives us 4 which indicates that **p2** is at a distance of 4 elements away from **p1.**
- So, **p2 – p1 +1** gives the number of elements in the array

12.5.8 Comparing two pointers

If two pointers are associated with the same array, then comparison of two pointers is allowed using relational operators. But, if the two pointers are associated with different arrays, even though comparisons of two pointers is allowed, the result is meaningless.

Consider the following declaration and initialization:

```
int     a[5] = {10, 20, 30, 40, 50};
int     *p1;
int     *p2;
float   *f;

p1 = a;          /* or p1 = &a[0] */
p2 = &a[4];
```

The various valid and invalid statements are shown below:

- p2 != p1;   /* **Valid** */;
- p1 == p2;   /* **Valid** */;
- p1 <= p2;   /* **Valid** */;
- p1 >= p2;   /* **Valid** */
- f != p1;   /* **Invalid:** Since type of both operands is not same */

Note: Multiplying and dividing a pointer variable with any other variable or integer is not allowed.

Now, let us see "How to write a program to display array elements by comparing of two pointers?"

Design: Let us use two pointers **p** and **q** where **p** points to the first element of array **a** and **q** points to the last element of array **a** as shown below:

	p ↓				q ↓
	0100	**0102**	**0104**	**0106**	**0108**
a	**10**	**20**	**30**	**40**	**50**

Observe from the above figure that as long as **p <= q**, value pointed to by **p** can be printed and updated using the following statements:

```
while (p <= q)
{
        printf("%d ",*p);
        p++;
}
```

Output

10 20 30 40 50

So, the complete program is shown below:

Example 12.24: Program to display array elements by comparing pointers

```
#include <stdio.h>

void main()
{
        int     a[] = {10, 20, 30, 40, 50 };
        int     *p;
        int     *q;

        p = &a[0];      /* point p to the first element */
        q = &a[4];      /* point q to the last element*/
        while (p <= q)          /* Comparing two pointer values */
        {
                printf("%d ",*p);
                p++;
        }
        printf("\n");
}
```

Note: Two pointer subtractions and two pointer comparisons are generally performed if both the pointers point to the same array.

12.6 Passing an array to a function / character pointer and functions

As we pass various parameters to functions, we can also pass name of an array as a parameter. **Note:** Name of an array is a pointer to the first element. So, when we pass an array to a function we should not use the address operator. The syntax of a function call is:

```
function_name (a);    /* Here a should have been declared as array */
```

The two ways of declaring and using the array in the called function are:

- using pointer declaration
- using array declaration

```
void function_name(int a[])
{
        /* ith item can be accessed
           using a[i]
        */
}
```

```
void function_name(int *a)
{
        /* ith item can be accessed
           using *(a+i)
        */
}
```

Note: Easier way of writing a program using pointers

- write a program using arrays i.e., may be using a[i] or a[j] etc.
- Then replacing a[i] by *(a+i) and a[j] by *(a+j) we get the program using pointers.

Now, using the above technique, any array program can be converted into a program using pointers.

12.6.1 strlen(str) – String Length

Consider the function to find the length of the string (Refer example 10.11, section 10.5.1 for design details). Various versions of the functions are written side by side to show the difference:

Example 12.25: Function returning the string length

Using arrays

```
int my_strlen(char str[])
{
    int  i = 0;

    /* compute the  length */
    while (str[i] != '\0')
        i++;

    return i;
}
```

Using pointers

```
int my_strlen(char *str)
{
    int   i = 0;

    /* compute length */
    while (*(str +i) )
        i++;

    return i;
}
```

```
int my_strlen(char *str)
{
    char *ptr = str;

    while (*ptr++)
        ;

    return ptr-str;
}
```

The C program to access any of the above functions can be written as shown below:

Example 12.26: Program using the user-defined function my_strlen()

```
#include <stdio.h>
/* Include: Example 12.25: to compute the length */

void main()
{
        char str[20];
        int   i;

        printf("Enter the string\n");
        gets(str);

        i = my_strlen(str) ;

        printf("Length = %d\n", i);
}
```

Input

Enter the string
Rama

i = 4

Output

Length = 4

12.6.2 strcpy(dest, src) – string copy

Now, let us write a function to implement *strcpy* (Refer section 10.5.2, example 10.14 for design details). So, the final function to copy the contents of source string **src** to destination string **dest** using arrays as well as using pointers is shown below:

Example 12.27: Function to copy string **src** to string **dest** using 3 methods.

Using arrays

```
void my_strcpy(char dest[], char src[])
{
        int     i = 0;

        /* Copy the string */
        while (src[i] != '\0')
        {
                dest[i] = src[i];
                i++;
        }

        /* Attach null character at the end */
        dest[i] = '\0';
}
```

Using Pointers

```
void my_strcpy(char *dest, char *src)
{
        /* copy the string */
        while ( *src != '\0')
                *dest++ = *src++;

        /* attach null character at end */
        *dest = '\0';
}
```

Note: Following is most efficient one

```
void my_strcpy(char *dest, char *src)
{
        while (*dest++ = *src++)
                ;
}
```

Note: Observe the null statement ";" in the third version of ***my_strcpy***. It does nothing. The condition in the while loop i.e., ***dest++ = *src++** is repeatedly executed and each character of the source is copied into destination including '\0'. Once '\0' is reached, the condition fails and control comes out of the loop. The complete program which uses the user defined function is shown below:

Example 12.28: Program using the user-defined function my_strcpy()

```
#include <stdio.h>

/* Include: Example 12.27 to compute the length */

void main()
{
        char src[20],  dest[20];

        printf("Enter the string\n");
        gets(src);

        my_strcpy(dest, src);

        printf("Dest string = %s\n", dest);
}
```

TRACING

```
Enter the string
RAMA

dest = "RAMA"

Dest string = RAMA
```

12.6.3 strcmp(s1, s2) – string compare

This function is used to compare two strings. The design details are given in section 10.5.7, example 10.25. The function using arrays and pointers are given side by side below:

Example 12.29: Function to compute two strings.

Using arrays

```
int my_strcmp(char  s1[], char  s2[])
{
        int  i;

        i = 0;
        while (s1[i] == s2[i])
        {
                if (s1[i] == '\0') break;

                i++
        }

        return s1[i] – s2[i];
}
```

Using Pointers

```
int my_stcmp(char *s1, char *s2)
{


        while (*s1 == *s2)
        {
                if (*s1 == '\0') break;

                s1++, s2++;
        }

        return *s1 - *s2;
}
```

The above function returns one of the following values:

- zero if **s1** = **s2**
- positive if **s1** > **s2**
- negative if **s1** < **s2**

The complete program showing the usage of ***my_strcmp*** is shown below:

Example 12.30: C program showing the usage of ***my_strcmp***

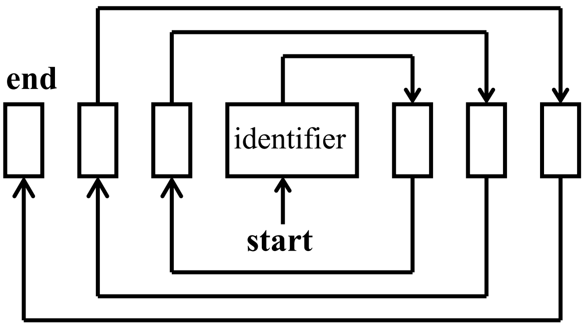
```
#include <stdio.h>
/* Include: example 12.29: Function my_strcmp */
void main()
{
        char s1[] = "RAMA";
        char s2[] = "KRISHNA";
        int     difference;

        difference = my_strcmp(s1, s2);

        if (difference == 0)
                printf("String s1 = string s2\n");
        else if (difference >0)
                printf("String s1 > string s2\n");
        else
                printf("String s1 < string s2\n");
}
```

12.7 Understanding complex declarations

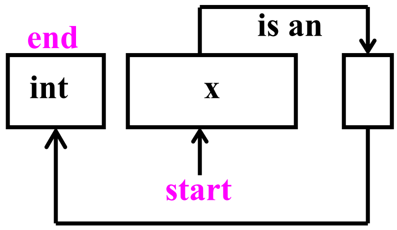
Note that it is very difficult to interpret and understand the declarations especially related to pointers. To read and understand the complicated declarations, we can follow the right-left rule. Now, let us see "What is right-left rule?"

Definition: The right-left rule can be stated as follows:

- Start with the identifier in the center of declaration
- Read the declarations in a spiral manner once going right and then left, again right and left and so on till all entities are read i.e, right-left reading of each symbol is done alternatively spinally. This concept can be represented pictorially as shown on the right hand side.



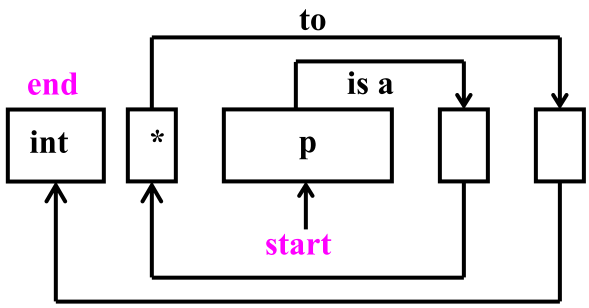
Example 12.31: Interpret the declaration: **int x.** The declaration can be pictorially represented as shwon below:



Note: Read and interpret the entity in each box in the direction of the arrow mark along with labels.

i.e., **x** is an **int**. In other words, **x is an integer**

Example 12.32: Interpret the declaration: **int *p.** The declaration can be pictorially represented as shwon below:



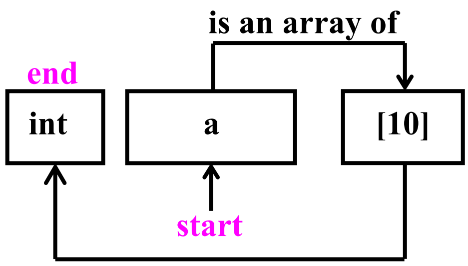
Reading in the direction of arrow along with the labels we have:

p is a * to int

i.e., **p is a pointer to an integer**

[By reading * as pointer, **int** as integer]

Example 12.33: Interpret the declaration: **int a[10].** The declaration can be pictorially represented as shwon below:



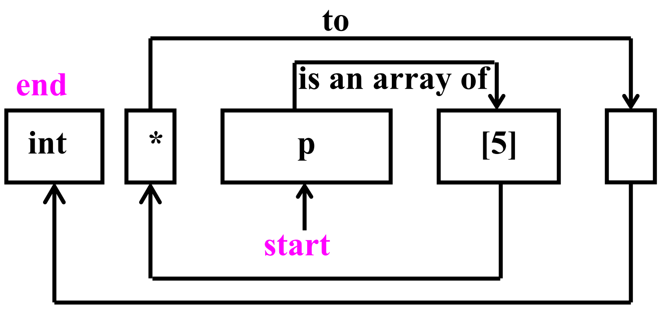
Reading in the direction of arrow along with the labels we have:

a is an array of 10 int

i.e., a is an array of 10 integer

[By reading int as integer]

Example 12.34: Interpret the declaration: **int *p[5].** The declaration can be pictorially represented as shwon below:

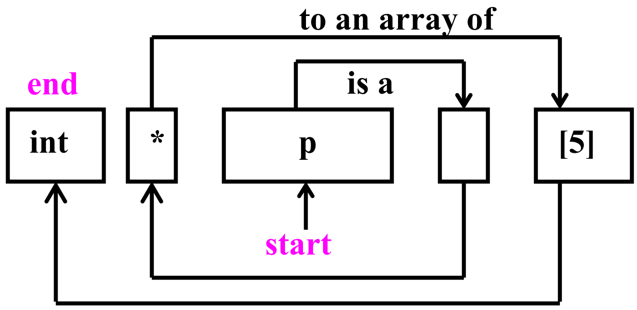


Reading in the direction of arrow along with the labels we have:

p is an array of 5 * to int

i.e., **p is an array of 5 pointers to integers** where * is pointer, **int** is integer

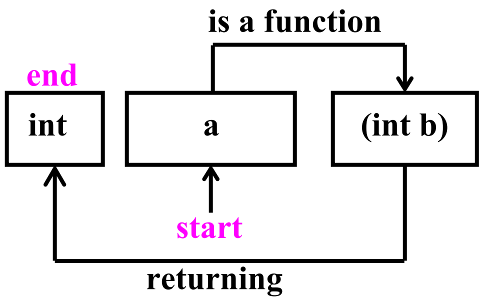
Example 12.35: Interpret the declaration: **int (*p) [5].** The declaration can be pictorially represented as shwon below:



Note: Preference is given for the expression withing parantheses. Reading in the direction of arrow along with the labels we have:

p is a * to an array of [5] int i.e., **p is a pointer to an array of 5 integers** where * is pointer, **int** is integer

Example 12.36: Interpret the declaration: **int a (int b).** The declaration can be pictorially represented as shwon below:



with b as integer parameter

Note: If an identifier is followed by (….), it indicates a function call or function declaration. So, reading in the direction of arrow along with the labels we have:

a is a function with (int b) and returning int

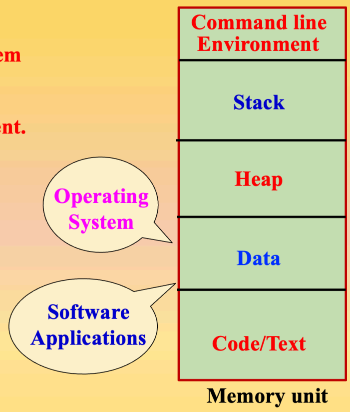
i.e., *a* is a function which accepts *b* an integer as a parameter and returning an integer

12.8 Memory allocation functions

What are memory allocation techniques?

Definition: Memory is a hardware unit where the data or instructions (programs) are stored. The process or mechanism by which memory space is allocated to variables and constants during run time i.e., when the program is being executed is called memory allocation.

- **Memory allocation is a hardware operation that is managed by Operating System and software applications.**
- **Memory allocation is achieved through a process known as memory management.**
- **The two memory allocation techniques are:**
  - **Static Memory Allocation**
  - **Dynamic Memory Allocation**



Now, let us see "What is static memory allocation?"

What is static memory allocation technique?

Definition: The process of allocating the memory space in the stack area during run-time as decided by the compiler during compile time is called static memory allocation.

- The size of the memory space to be allocated for various types of data is decided by the compiler during compile time since the compiler knows the size of each data type.

```
int     a;   // Need to allocate 4 bytes
float   b;   // Need to allocate 4 bytes
double  c;   // Need to allocate 8 bytes
```

- The compiler generates necessary machine instructions to allocate the memory space based on size of each data type in the stack area of memory. When these instructions are executed during run-time, memory is allocated for these data items in the stack area of the memory.

Note: Compiler will not allocate memory space for variables. It generates necessary machine instructions to allocate the memory space during run-time.

What are the disadvantages of static memory allocation technique?

- The size of the memory space to be allocated is fixed during compilation time.

Ex: `int a [5] = { 50, 40, 20, 90, 70 };` // Instruction given by programmer to the compiler

// Memory space allocated by the compiler during compilation

a [0]	[1]	[2]	[3]	[4]
50	40	20	90	70

- Once the memory space is fixed during compilation time, it size cannot be increased to accommodate more data.
- If more space is allocated during compilation time and only few elements are stored, it results in wastage of more space. Its size cannot be decreased to accommodate less data.

Ex: `int a [10] = { 50, 40, 20, 90, 70 };`

a [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
50	40	20	90	70					

Note: All the disadvantages of static memory allocation are overcome using dynamic memory allocation technique.

Note: Even though the memory is allocated for local variables on the stack during run time, the size of memory to be allocated is decided during compilation time.

Dr. Padma Reddy A.M | Sai Vidya Institute of Technology | Bengaluru | download: nandipublications.com, saividya.ac.in

Note: If there is an unpredictable storage requirement, then static allocation technique is not at all used. This is the point where the concept of dynamic allocation comes into picture. Now, the question is "What is dynamic memory allocation?"

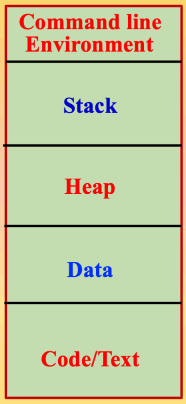
What is dynamic memory allocation technique?

Definition: **The process or mechanism by which memory space is allocated to store data during run time i.e., when the program is being is executed is called dynamic memory allocation. In this technique, the user can request the Operating System to allocate the specified memory space to store the data from the heap area. In C/C++ language,**

- **Memory allocation is done using functions such as: malloc(), calloc() and realloc()**
- **Memory de-allocation is done using functions such as: free()**

Advantages

- **Enables us to use as much storage as we want without worrying any wastage.**
- **Enables us to enter required amount of data during run time.**
- **Enables us to remove the required amount of data during run time.**



Memory layout

Now, let us see "What are the differences between static memory allocation and dynamic memory allocation?" The various differences between static allocation and dynamic allocation technique are shown below:

Static allocation technique	Dynamic allocation technique
1. Memory is allocated during compilation time	1. Memory is allocated during execution time
2. The size of the memory to be allocated is fixed during compilation time and cannot be altered during execution time	2. When required memory can be allocated and when not required memory can be de-allocated
3. Used only when the data size is fixed and known in advance before processing	3. Used only for unpredictable memory requirement.
4. Execution is faster, since memory is already allocated and data manipulation is done on these allocated memory locations	4. Execution is slower since memory has to be allocated during run time. Data manipulation is done only after allocating the memory.
5. Memory is allocated either in stack area (for local variables) or data area (for global and static variables).	5. Memory is allocated only in heap area

6. **Ex:** arrays	6. **Ex:** Dynamic arrays, linked lists, trees

Now, let us see “What are the various memory management functions in C?” Th

The various memory management functions available in C are:

- **malloc()**
- **calloc()**
- **realloc()**
- **free()**

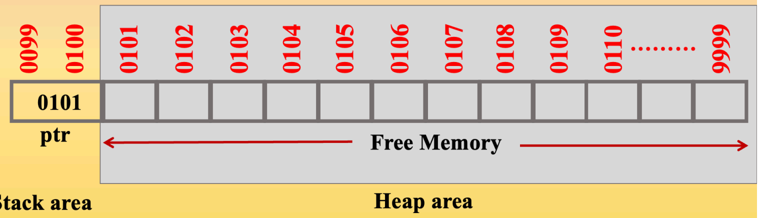
- **The required memory space for the data is allocated during run-time in heap area.**
- **If specified memory space is not available, the function malloc() returns NULL.**
- **If memory space is successfully allocated, the function malloc() returns address of the first byte.**

Syntax:

```
#include <stdlib.h>
void  *malloc ( size_t  size )
```

For example,

```
int   *ptr;
ptr  = ( int * )  calloc  ( 2, sizeof (int) ) ;
if (  ptr == NULL )
{
      printf ( " Insufficient memory\n" ) ;
      exit (0);
}
..................
free ( ptr ) ;
```



What is malloc()?

Definition: malloc() is the shorthand name for memory allocation which is a built in function in C. It is declared in the header file “stdlib.h”. Using this function the programmer can request the Operating System to allocate a block of contiguous memory according to the size specified in the argument.

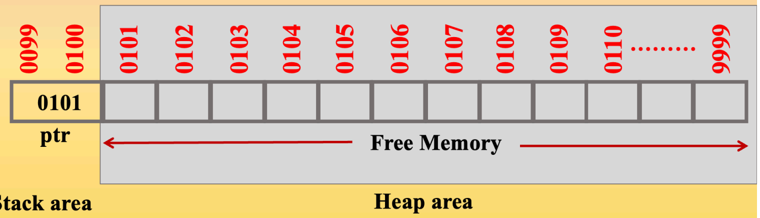
- **The required memory space for the data is allocated during run-time in heap area.**
- **If specified memory space is not available, the function malloc() returns NULL.**
- **If memory space is successfully allocated, the function malloc() returns address of the first byte.**

Syntax:

```
#include <stdlib.h>
void  *malloc ( size_t  size )
```

For example,

```
int   *ptr;
ptr  = ( int * )  calloc  ( 2, sizeof (int) ) ;
if (  ptr == NULL )
{
      printf ( " Insufficient memory\n" ) ;
      exit (0);
}
..................
free ( ptr ) ;
```



What are the differences between malloc() and calloc()?

	malloc()	calloc()
■ Syntax:	#include <stdlib.h> data_type *ptr; ptr = (dat_type *) malloc(size); size : Number of bytes to be allocated	#include <stdlib.h> data_type *ptr; ptr = (dat_type *) calloc(n, size) ; Takes two arguments: n : Number of blocks to be allocated size : Number of bytes to be allocated for each block
■ Memory allocation	Allocates a single block of memory of size bytes.	Allocates multiple blocks of memory where ■ Each block is of same size ■ Size represent the number of bytes to be allocated for each block.
■ Memory Initialization	Allocated memory space will not be initialized to any value.	Allocated memory space is initialized to 0.
■ Initializing allocated memory to 0's	p = (int *) malloc (sizeof (int) * n) ; memset (p, 0, sizeof (int) * n) ;	p = (int *) calloc (n, sizeof (int)) ;

Dr. Padma Reddy A.M | Sai Vidya Institute of Technology | Bengaluru | download: nandipublications.com, saividya.ac.in

How to read an array of N elements dynamically?

```
#include <stdio..h>
#include <stdlib.h>

int * allocate_memory ( int  n )
{
     int  *ptr;

     ptr  =  ( int * ) malloc ( n * sizeof ( int ) );

     if ( ptr == NULL )
     {
          printf ( " Insufficient memory\n") ;
          exit (0);
     }
     return  ptr;
}
```

```
void  main  ( )
{
     int     n,  *a;

     printf ( "Enter no. of items:\n" );
     scanf ("%d",  &n);
     a  = allocate_memory ( n ) ;
     printf ("Enter array items:\n" );
     for  ( i  =  0;  i  <  n;  i++)  scanf ("%d" , (a + i) ) ;

     printf ("Array items:\n" );
     for  ( i  =  0;  i  <  n;  i++) printf ("%d\n",  * (a + i) ) ;

     free ( a );
}
```

12.8.1 malloc(size)

Now, let us see "What is the purpose of using malloc?" This function allows the program to allocate memory explicitly as and when required and the exact amount needed during execution. This function allocates a block of memory. The size of the block is the number of bytes specified in the parameter. The syntax is shown below:

```
#include <stdlib.h>          /* Prototype definition of malloc() is available */
……..
ptr = (data_type *) malloc(size);
…….
```

where

- *ptr* is a pointer variable of type **data_type**

- **data_type** can be any of the basic data type or user defined data type
- *size* is the number of bytes required

Observe the following points:

- On successful allocation, the function returns the address of first byte of allocated memory. Since address is returned, the return type is a **void** pointer. By ***type casting*** appropriately we can use it to store integer, float etc.
- If specified size of memory is not available, the condition is called **"overflow of memory".** In such case, the function returns NULL. It is the responsibility of the programmer to check whether the sufficient memory is allocated or not as shown below:

```
void function_name()
{
        ……..
        ptr = (data_type *) malloc(size);

        if (ptr == NULL)
        {
                printf("Insufficient memory\n");
                exit(0);
        }
        ……..
        ……..
}
```

Example 12.37: Program showing the usage of malloc() function

TRACING	

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
        int     i,n;
        int     *ptr;

        printf("Enter the number of elements\n");
        scanf("%d",&n);

        ptr = (int *) malloc (sizeof(int)* n);

        /* If sufficient memory is not allocated */
        if (ptr == NULL)
        {
                printf("Insuffient memory\n");
                return;
```

TRACING

Execution starts from here

Inputs

Enter the number of elements
5

```
    }

    /* Read N elements */
    printf("Enter N elements\n");            Enter N elements
    for (i = 0; i < n; i++)
        scanf("%d", ptr+i);                      10 20 30 40 50

    printf("The given elements are\n");
    for (i = 0; i < n; i++)                      The given elements are
        printf("%d  ", *(ptr+i));                10  20  30  40  50
}
```

12.8.2 calloc(n, size)

Now, let us see "What is the purpose of using calloc?" This function is used to allocate multiple blocks of memory. Here, **calloc** – stands for contiguous allocation of multiple blocks and is mainly used to allocate memory for arrays. The number of blocks is determined by the first parameter **n.** The size of each block is equal to the number of bytes specified in the parameter i.e., **size.** Thus, total number of bytes allocated is n*size and all bytes will be initialized to 0. The syntax is shown below:

```
#include <stdlib.h>          /* Prototype definition of calloc() is available */
……..
ptr = (data_type *) calloc(n, size);
……..
```

where

- *ptr* is a pointer variable of type **data_type**
- **data_type** can be any of the basic data type or user defined data type
- **n** is the number of blocks to be allocated
- *size* is the number of bytes in each block

Observe the following points:

- On successful allocation, the function returns the address of first byte of allocated memory. Since address is returned, the return type is a **void** pointer. By **type casting** appropriately we can use it to store integer, float etc.
- If specified size of memory is not available, the condition is called **"overflow of memory".** In such case, the function returns NULL. It is the responsibility of the programmer to check whether the sufficient memory is allocated or not as shown below:

```
void function_name()
{
    ……..
    ptr = (data_type *) calloc(size);
```

```
        if (ptr == NULL)
        {
                printf("Insufficient memory\n");
                exit(0);
        }
        ........
}
```

Example 12.38: Program to find maximum of n numbers using dynamic arrays

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
        int *a, i, j, n;

        printf("Enter the no. of elements\n");
        scanf("%d",&n);

        /* Allocate the required number of memory locations dynamically */
        a = (int *) calloc( n, sizeof(int) );

        if (a == NULL)                  /* If required amount of memory  */
        {                               /* is not allocated */
                printf("Insufficient memory\n");
                return;
        }

        printf("Enter %d elements\n", n);  /* Read all elements */
        for ( i = 0; i < n; i++)
        {
                scanf("%d",&a[i]);
        }

        j = 0;                          /* Initial position of the largest number */
        for ( i = 1; i < n; i++)
        {
                if ( a[i] > a[j] ) j = i;   /* obtain position of the largest element*/
        }

        printf("The biggest = %d is found in pos = %d\n",a[j], j+1);
```

```
        free(a);        /* free the memory allocated to n numbers */
}
```

Observe the following points:

- The variable *a* is a pointer to an **int.**
- Once memory is allocated dynamically using *calloc()*, the address of the first byte is copied into *a*.
- From this point onwards the variable *a* can be used as an array or used as a pointer. If *a* is used as an array, the i^{th} element can be accessed by a[i] and the address of i^{th} element can be obtained using &a[i]
- If *ptr* is used as a pointer, the i^{th} element can be accessed by *(a + i) and the address of i^{th} element can be obtained using (a + i)
- In the above program in place of (a + i) we can use &a[i]. At the same time, in place of *(a + i) we can use a[i]

12.8.3 realloc(ptr, size)

Now, let us see "What is the purpose of using realloc?"

Before using this function, the memory should have been allocated using malloc() or calloc(). Sometimes, the allocated memory may not be sufficient and we may require additional memory space. Sometimes, the allocated memory may be much larger and we want to reduce the size of allocated memory. In both situations, the size of allocated memory can be changed using realloc() and the process is called *reallocation* of memory. The reallocation is done as shown below:

- realloc() changes the size of the block by extending or deleting the memory at the end of the block.
- If the existing memory can be extended, **ptr** value will not be changed
- If the memory cannot be extended, this function allocates a completely new block and copies the contents of existing memory block into new memory block and then deletes the old memory block. The syntax is shown below:

```
    #include <stdlib.h>         /* Prototype definition of realloc() is available */
    ……..
    ……..
    ptr = (data_type *) realloc(ptr, size);
    …….
    …….
```

where

- ptr is a pointer to a block of previously allocated memory either using malloc() or calloc().

```
    ♦ size is new size of the block
    if (ptr == NULL)
    {                                   /* Memory is not allocated */
        printf("Insufficient memory\n");
        return;
    }
```

Now, let us see "What does this function return?" This function returns the following values:

- On successful allocation, the function returns the address of first byte of allocated memory.
- If specified size of memory cannot be allocated, the condition is called **"overflow of memory".** In such case, the function returns NULL.

Example: 12.39: C program showing the usage of realloc() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
{
    char *str;

    str = (char *) malloc(10);          /* allocate memory for the string */
    strcpy(str, "Computer");

    str = (char *) realloc(str, 40);
    strcpy(str, "Computer Science and Engineering");
}
```

12.8.4 free(ptr)

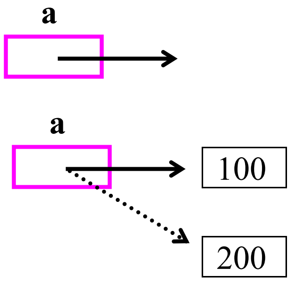
Now, let us see "What is the purpose of using free()?"

This function is used to de-allocate (or free) the allocated block of memory which is allocated by using the functions calloc(), malloc() or realloc(). It is the responsibility of a programmer to de-allocate memory whenever it is not required by the program and initialize **ptr** to NULL. The syntax is shown below:

```
    #include <stdlib.h>         /* Prototype definition of free() is available */
    ……..
    free(ptr);
    ptr = NULL;
    …….
```

Example: 12.40: Sample program to show the problems that occur when free() is not used.

```
1. #include <stdlib.h>
2.
3. void main()
4. {
5.      int *a;
6.
7.      a = (int *) malloc(sizeof(int));
8.      *a = 100;
9.
10.     a = (int *) malloc(sizeof(int));
11.     *a = 200;
12. }
```



Now, let us see "What will happen if the above program is executed?" The various activities done during execution are shown below:

- When control enters into the function main, memory for the variable ***a*** will be allocated and will not be initialized.
- When memory is allocated successfully by malloc (line 7), the address of the first byte is stored in the pointer **a** and integer **100** is stored in the allocated memory (line 8).
- But, when the memory is allocated successfully by using the function malloc in line 10, address of the first byte of new memory block is copied into **a** (shown using dotted lines.)

Observe that the pointer **a** points to the most recently allocated memory, thereby making the earlier allocated memory inaccessible. So, memory location where the value **100** is stored is inaccessible to any of the program and is not possible to free so that it can be reused. *This problem where in memory is reserved dynamically but not accessible to any of the program is called* ***memory leakage.*** So, care should be taken while allocating and de-allocating the memory. It is the responsibility of the programmer to allocate the memory and *de-allocate the memory when no longer required.*

Note: Observe the following points:

- It is an error to free memory with a NULL pointer
- It is an error to free a memory pointing to other than the first element of an allocated block
- It is an error to refer to memory after it has been de-allocated

Note: Be careful, if we dynamically allocate memory in a function. We know that local variables vanish when the function is terminated. If **ptr** is a pointer variable used in a function, then the memory allocated for **ptr** is de-allocated automatically. But, the space allocated dynamically using malloc, calloc or realloc will not be de-allocated automatically when the control comes out of the function. But, the allocated memory cannot be accessed and hence cannot be used. This unused un-accessible memory results in memory leakage.

12.12 Advantages and disadvantages of pointers

By this time, we should have understood the full concepts of C pointers and given any problem we should be in a position to solve. After understanding the full concepts of pointers, we should be in a position to answer the question *"What are the advantages and disadvantages of pointers?"*

Advantages

- More than one value can be returned using pointer concept (pass by reference).
- Very compact code can be written using pointers.
- Data accessing is much faster when compared to arrays.
- Using pointers, we can access *byte* or *word* locations and the CPU *registers* directly. The pointers in C are mainly useful in processing of non-primitive data structures such as arrays, linked lists etc.

Disadvantages

- Un-initialized pointers or pointers containing invalid addresses can cause system crash.
- It is very easy to use pointers incorrectly, causing bugs that are very difficult to identify and correct.
- They are confusing and difficult to understand in the beginning and if they are misused the result is not predictable.

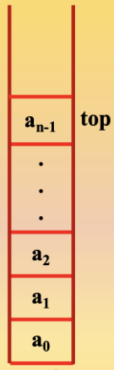
What is a stack? How stack can be represented?

Definition: Stack is a special type of data structure where elements are inserted from one end and elements are deleted from the same end.

- Using the above approach, the Last element Inserted is the First element to be deleted Out.
- Hence, stack is also called LIFO data structure.
- The stack s = { a_0, a_1, a_2, a_3, ….. a_{n-1} } is pictorially represented as shown in fig:
- The elements are inserted onto the stack in the order a_0, a_1, a_2, a_3, ….. a_{n-1}
  i.e., item a_0 is inserted first, item a_1 is inserted next, and so on. Finally, a_{n-1} is inserted.
- Since a_{n-1} is on top of the stack, it is removed first, then a_{n-2} and so on.

Stack can be represented using:

- Array representation
- Linked representation



stack

stack s = { a_0, a_1, a_2, a_3, ….. a_{n-1} }

What are the operations that can be performed on stack?

The various operations that can be performed on stack are:

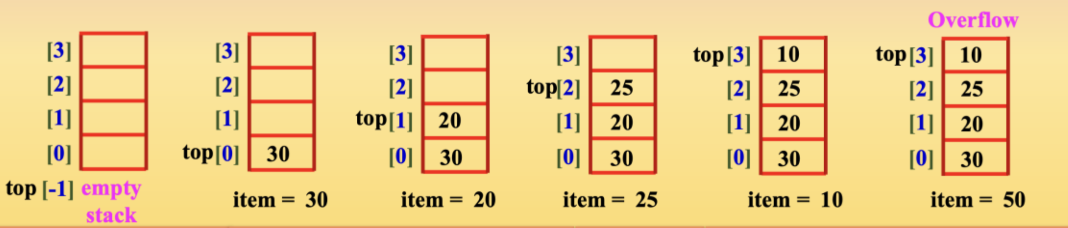
- Insertion : Inserting an element into the stack is called PUSH operation.
- Deletion : Deleting an element from the stack is called POP operation.
- Display : Contents of the stack are displayed.

How to insert an element into the stack?

Insertion

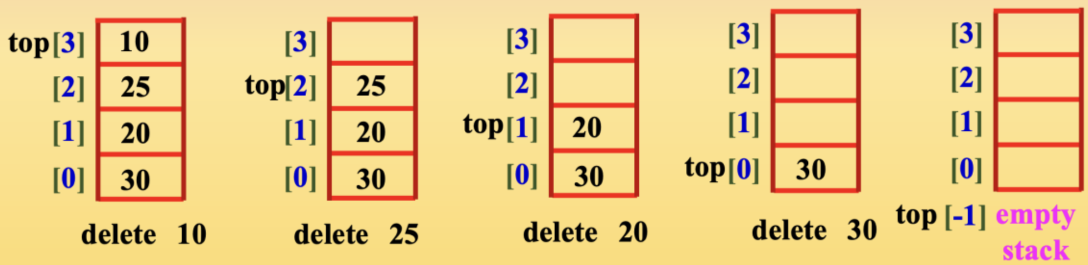
- Only one element is inserted at a time.
- An element is inserted only on top of the stack.
- Inserting an element into the stack is called PUSH operation.
- When stack is full it is not possible to insert any element into the stack.
- Trying to insert an element into the stack when the stack is full is called OVERFLOW of stack.

STACK_SIZE = 4



Deletion

- Only one element is deleted at a time.
- An element is deleted only from top of the stack.
- Deleting an element from the stack is called POP operation.
- When stack is empty it is not possible to delete any element from the stack.
- Trying to delete an element from the empty stack is called UNDERFLOW of stack.



How to write a C function to insert an item into the stack?

```
// Function to insert item into the stack
void push ( int item )
{
    // Check for overflow of stack
    if ( top == STACK_SIZE - 1 )
    {
        printf ("Stack Overflow" );
        return;
    }
```

```
Algorithm push
// Input:  item : element to be inserted
// Global/Parameters : top, stack [ 10 ]
// Check for overflow of stack
if ( top == STACK_SIZE - 1 )
    print ( "Stack Overflow" )
    return
```

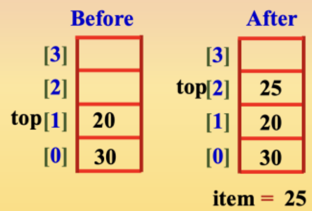
Case 1: Insertion not possible

STACK_SIZE = 4


```
    // Increment top by 1
    top++;
    // Insert an item into the stack
    stack [ top ] = item;

OR

    // Insert an item into the stack
    stack [++top ] = item;
}
```

```
// Increment top by 1
top = top + 1 /
top += 1      /
top++ / ++top

// Insert an item into the stack
stack [ top ] = item
```

Case 2: Insertion possible



How to write a C function to delete an item from stack?

```
// Function to delete an item from the stack
void  pop  ( )
{
      // Check for underflow of stack
      if (  top  ==  - 1 )
      {
            printf (“Stack Underflow”);
            return;
      }

      printf ( “Item deleted = %d “, stack[top]);
      // Decrement top by 1
      top =  top - 1;

                    OR

      printf (“Item deleted = %d “, stack[top--]);

      printf ( “\n“ ) ;
}
```

Algorithm pop

// Input: none

// Global/Parameters: top, stack[10]

```
// Check for underflow of stack
if (  top  ==  - 1 )
    print (“Stack Underflow”)
    return

print ( “Item deleted =  “, stack [top] )
// Decrement top by 1
top =  top - 1   /
top -= 1         /
top--   /  --top
```

Case 1: Deletion not possible

[3]	
[2]	
[1]	
[0]	

top [-1] empty stack — Underflow

Case 2: Deletion possible

Brfore

[3]	
top[2]	25
[1]	20
[0]	30

After

[3]	
[2]	
top[1]	20
[0]	30

Delete 25

How to write a C function to display stack contents?

```
// Function to display the contents of stack
void  display ( )
{
      int     i;

      // Check for empty stack
      if (  top  ==  - 1 )
      {
            print ( “Stack is empty” ) ;
            return;
      }

      printf ( “Stack :  “ );

      for ( i  =  0;  i <= top;  i++)
            printf ( “ %d    “, stack [i] );

      printf ( “\n “ );
}
```

Algorithm displaly

// Input: none

// Global: top, stack [10]

```
// Check for empty stack
if (  top  ==  - 1 )
    print ( “Stack is empty”  )
    return

print ( “Stack :  “ )
print stack [i]   ∀ i  =  0 to top
print  “\n”
```

Case 1: Display not possible

[3]	
[2]	
[1]	
[0]	

top [-1] empty stack — Stack is empty

Case 2: Display possible

[3]	
top[2]	25
[1]	20
[0]	30

How to write a C program to implement stack operations using global variables?

```
#include <stdio..h>
#include <stdlib.h>

#define  STACK_SIZE     5

int      top  =  -1 ;
int      stack  [10];

// Function to insert an item into the stack
void  push ( int  item  )
{
     // Write the complete function
}

// Function to delete an element from the stack
void  pop   ( )
{
     // Write the complete function
}

// Function to display the contents of stack
void  display  ( )
{
     // Write the complete function
}

void  main ( )
{
     int          choice, item;
     // Perform stack operations any number of times
     for   ( ;; )
     {
          printf ( “ 1:Push   2:Pop   3:Display   4:Exit : “ );
          scanf ( “ %d “, &choice );
          switch ( choice )
          {
               case 1 :  printf (“ Enter the item : “);
                         scanf ( “ %d “, &item );
                         push ( item ) ;
                         break;

               case 2 :  pop ( );
                         break;

               case 3 :  display ( );
                         break;

               default: exit ( 0 );
          }
     }
}
```

How to implement stack using dynamic arrays? (Using global variables)

```
#include <stdio..h>
#include <stdlib.h>

int     STACK_SIZE = 1;

int     top = -1 ;
int     *stack ;

// Function to insert item into the stack
void push ( int item )
{     // Check for overflow of stack
      if ( top == STACK_SIZE - 1 )
      {
           printf ("Stack Overflow" );
           STACK_SIZE++;
           stack = realloc (stack, STACK_SIZE * sizeof ( int ) );
      }

      // Insert an item into the stack
      stack [++top ] = item;
}
```

```
void main ( )
{
     int      choice, item;
     stack = ( int * ) malloc ( STACK_SIZE * sizeof ( int ) );
     for ( ;; )
     {
          printf ( "1:Push  2:Pop  3:Display  4:Exit : " );
          scanf ( "%d ", &choice );
          switch ( choice )
          {
               case 1 :  printf (" Enter the item : ");
                         scanf ( "%d ", &item );

                         push ( item ) ;

                         break;

               case 2 :  pop ( );
                         break;

               case 3 :  display ( );
                         break;

               default: exit ( 0 );
          }
     }
}
```

How to write a C function to insert an item into the stack? by passing parameters

```
// Function to insert item into the stack
void push (int item, int *top, int stack[])
{
     // Check for overflow of stack
     if ( *top == STACK_SIZE - 1 )
     {
          printf ("Stack Overflow" );
          return ;
     }

     // Increment top by 1
     (*top) ++;

     // Insert an item into the stack
     stack [*top ] = item;

                  OR

     // Insert an item into the stack
     stack [++ (*top)] = item;
}
```

Algorithm push

// Input: item : element to be inserted

// Global/Parameters: top, stack [10]

```
// Check for overflow of stack
if ( top == STACK_SIZE - 1 )

print ( "Stack Overflow" )
return

// Increment top by 1
top = top + 1 /
top += 1      /
top++ / ++top

// Insert an item into the stack
stack [ top ] = item
```

Case 1: Insertion not possible

STACK_SIZE = 4

top[3]	10
[2]	25
[1]	20
[0]	30

Overflow

Case 2: Insertion possible

Before

[3]	
[2]	
top[1]	20
[0]	30

After

[3]	
top[2]	25
[1]	20
[0]	30

item = 25

How to write a C function to delete an item from stack? – By passing parameters

```c
// Function to delete an item from the stack
void  pop ( int *top, int stack[])
{
    // Check for underflow of stack
    if (*top == -1 )
    {
        printf ("Stack Underflow" );
        return ;
    }
    printf ("Item deleted = %d ", stack[*top]);
    // Decrement top by 1
    *top = *top - 1;

                OR

    printf ("Item deleted =%d",stack[(*top)--]);

    printf ( "\n" ) ;
}
```

Algorithm pop

// Input: none

// Global/Parameters: top, stack[10]

```
// Check for underflow of stack
if ( top == - 1 )
    print ( "Stack Underflow")
    return

print ("Item deleted = ", stack [top] )
// Decrement top by 1
top = top - 1  /
top -= 1       /
top--  /  --top
```

Case 1: Deletion not possible

[3]	
[2]	
[1]	
[0]	
top [-1]	

Underflow

empty stack

Case 2: Deletion possible

Brfore		After	
[3]		[3]	
top[2]	25	[2]	
[1]	20	top[1]	20
[0]	30	[0]	30

Delete 25

Dr. Padma Reddy A.M | Sai Vidya Institute of Technology | Bengaluru | download: nandipublications.com, saividya.ac.in

How to write a C function to display stack contents? – By passing parameters

```c
// Function to display the contents of stack
void  display ( int  top, int stack[] )
{
    int    i;
    // Check for empty stack
    if (  top  ==  -1 )
    {
        printf ("Stack is empty" );
        return ;
    }
    printf ( "Stack :  " );

    for ( i = 0;  i <= top;  i++)
        printf ( " %d   ", stack [i] );

    printf ( "\n " );
}
```

Algorithm displaly

// Input: none

// Global/Parameters: top, stack[10]

```
// Check for empty stack
if ( top == - 1 )
    print ( "Stack is empty" )
    return

print ("Stack : ")
print stack [i]   ∀ i = 0 to top
print "\n"
```

Case 1: Display not possible

[3]	
[2]	
[1]	
[0]	
top [-1]	

Stack is empty

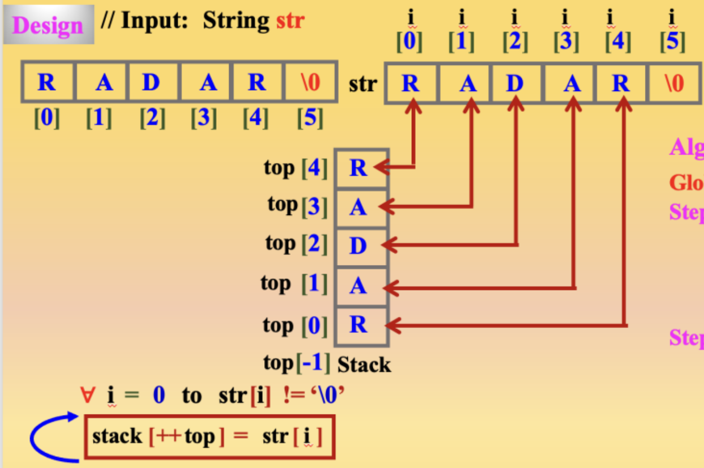
empty stack

Case 2: Display possible

[3]	
top[2]	25
[1]	20
[0]	30

What is a palindrome? How to check whether the string is palindrome or not?

Design // Input: String str




```
Algorithm Palindrome ( String )
Global / Parameters : stack, top
Step 1: // Insert each character on to stack
        ∀ i = 0 to str[i] != '\0'
            stack [++top] = str [ i ]
        end for

Step 2: // compare each character of string with stack
        ∀ i = 0 to str[i] != '\0'
            if ( str[i] == stack [ top -- ] ) continue

            print ( str, " : Not a Palindrome")
            return
        end for

        print ( str, " : is a Palindrome")

Step 3: // Finished
        return
```


What is a palindrome? How to check whether the string is palindrome or not?

```c
#include < stdio.h >

void palindrome ( char str [ ] )
{
    int    i;

    // Insert each character on to stack
    for ( i = 0; str[i] != '\0' ; i++ )
    {
        stack [++top ] = str [ i ] ;
    }

    // compare each character of string with stack
    for ( i = 0; str[i] != '\0' ; i++ )
    {
        if ( str[i] == stack [ top -- ] ) continue;

        printf ( " %s : Not a Palindrome", str );
        return ;
    }

    printf ( " %s : Palindrome ", str );
}
```

```c
void main ( )
{
    char str [10] ;

    printf ( "Enter the string: " );
    scanf ( "%[^\n]" , str );

    palindrome ( str ) ;
}
```

What is an infix expression? What is postfix expression? What is prefix expression?

Infix expression: In an expression, if an operator is in between two operands, the expression is called an infix expression.

- The expressions may be parenthesized or un-parenthesized.
- Parenthesized infix expressions : (a + b), (6 + (3 - 2) * 5) ^ 2 + 3
- Un-parenthesized infix expressions : a + b , X ^ Y ^ Z - M + N + P / Q

Postfix expression: In an expression, if an operator follows the two operands, the expression is called a postfix expression.

- Postfix expression is also called suffix expression or reverse polish expression.
- The expressions are always un-parenthesized.
- For example, a b + , A B C - D * + E ^ F +, X Y Z $ $ M - N + P Q / +

Prefix expression: In an expression, if an operator precede the two operands, the expression is called a prefix expression.

- Prefix expression is also called polish expression.
- The expressions are always un-parenthesized.
- For example, + a b, + $ + A * - B C D E F, + + - $ X $ Y Z M N / P Q

How to evaluate an infix expression? What are the disadvantages?

Evaluation of infix expression is not recommended because of the following reasons:

- We need to repeatedly scan from left to right and right to left to identify the part of the expression to be evaluated.
- Requires the knowledge of precedence of operators and associativity of the operators
- The problem becomes more complex with the introduction of parentheses in the expressions because they change the order of precedence.
- Designing the algorithm or the program is very difficult using this traditional technique.

Advantage:

Easy for us to read and understand these type of expressions.
So, when we write the expressions, we use these type of expressions.

Infix: $(6 + \underbrace{(3 - 2)}_{1} * 5) \wedge 2 + 3$

$(6 + \underbrace{1 * 5}_{5}) \wedge 2 + 3$

$\underbrace{(6 + 5)}_{11} \wedge 2 + 3$

$\underbrace{11 \wedge 2}_{121} + 3$

$\underbrace{121 + 3}_{124}$

Dr. Padma Reddy A.M | Sai Vidya Institute of Technology | Bengaluru | download: nandipublications.com, saividya.ac.in

How to convert from infix to postfix?

$((A + \underbrace{(B - C)}_{T1} * D) \wedge E + F)$ T1 = B C -

$((A + \underbrace{T1 * D}_{T2}) \wedge E + F)$ T2 = T1 D *

$(\underbrace{(A + T2)}_{T3} \wedge E + F)$ T3 = A T2 +

$(\underbrace{T3 \wedge E}_{T4} + F)$ T4 = T3 E ^

(T4 + F)

T4 F +

T3 E ^ F +

A T2 + E ^ F +

A T1 D * + E ^ F +

$\boxed{A\ B\ C\ -\ D\ *\ +\ E\ \wedge\ F\ +}$ Postfix Expression

or

$\boxed{A\ B\ C\ -\ D\ *\ +\ E\ \$\ F\ +}$ Postfix Expression

How to convert from infix to postfix?

$\text{X} \wedge \underbrace{\text{Y} \wedge \text{Z}}_{\text{T1}} - \text{M} + \text{N} + \text{P} / \text{Q}$ T1 = Y Z ^

$\underbrace{\text{X} \wedge \text{T1}}_{\text{T2}} - \text{M} + \text{N} + \text{P} / \text{Q}$ T2 = X T1 ^

$\text{T2} - \text{M} + \text{N} + \underbrace{\text{P} / \text{Q}}_{\text{T3}}$ T3 = P Q /

$\underbrace{\text{T2} - \text{M}}_{\text{T4}} + \text{N} + \text{T3}$ T4 = T2 M -

$\underbrace{\text{T4} + \text{N}}_{\text{T5}} + \text{T3}$ T5 = T4 N +

$\underbrace{\text{T5} + \text{T3}}$

T5 T3 +

T5 T3 +

T4 N + P Q / +

T2 M - N + P Q / +

X T1 ^ M - N + P Q / +

$\boxed{\text{X Y Z ^ ^ M - N + P Q / +}}$ Postfix Expression

or

$\boxed{\text{X Y Z \$ \$ M - N + P Q / +}}$ Postfix Expression

What is the logic used while converting from infix to postfix?

Infix: (A + (B - C) * D)

Stack[]	F (s[top])	>	G (infix[i])	Postfix[]
#	-1	>	9	
# (	0	>	7	
# (A	8	>	1	A
# (	0	>	1	
# (+	2	>	9	
# (+ (	0	>	7	
# (+ (B	8	>	1	B
# (+ (	0	>	1	
# (+ (-	2	>	7	
# (+ (- C	8	>	0	C
# (+ (-	2	>	0	-
# (+ (	0	>	0	
# (+ (	2	>	3	
# (+ *	4	>	7	
# (+ * D	8	>	0	D
# (+ *	4	>	0	*
# (+	2	>	0	+
# (	0	>	0	
# (				A B C - D * +

Fully Parenthesized

	infix[i]	F Stack	G I/P
	#	-1	-
	)	-	0
L	+ -	2	1
L	* /	4	3
R	^ $	5	6
L	operands	8	7
L	(	0	9

Precedence table

4 + 2 - 3

4 * 2 / 8

2 ^ 3 ^ 2 $\quad 2^{3^{2}}$

How to design the algorithm to convert from infix to postfix?

Infix: A + (B - C) * D

Stack[]	F (s[top])	>	G (infix[i])	Postfix[]
#	-1	>	7	
# A	8	>	1	A
#	-1	>	1	
# +	2	>	9	
# + (	0	>	7	
# + (B	8	>	1	B
# + (	0	>	1	
# + (-	2	>	7	
# + (- C	8	>	0	C
# + (-	2	>	0	-
# + (	0	>	0	
# +	2	>	3	
# + *	4	>	7	
# + * D				A B C - D
# + *				*
# +				+
#				
				A B C - D * +

```
Algorithm   infix_2_postfix ( infix, postfix )
top  =  -1
s [ ++top ] = '#'

∀ i = 0  to  infix [ i ]  != '\0'
     while ( F ( s[top] ) > G ( infix [ i ] ) )
          postfix [ j++ ] = s [ top -- ] ;

     if ( F ( s[top] ) != G ( infix[i] ) )
          s [++top ] = infix [ i ]
     else
          top --

while ( s[top]  !=  '#' )
     postfix [ j++ ] = s [ top -- ] ;

postfix [ j ] = '\0'
```

Dr. Padma Reddy A.M | Sai Vidya Institute of Technology | Bengaluru | download: nandipublications.com, saividya.ac.in

```
/* Insert the function: infix_2_postfix (infix, postfix) */
void  infix_2_postfix ( char  infix[],   char  postfix [] ) ;
{
        int       top,  i,  j = 0;
        char     s[20];

        top  =   -1;
        s [ ++top ] =  '#' ;

        for (   i = 0;   infix [ i ]  !=  '\0' ;  i++ )
        {
             while (  F ( s[top] )  >  G ( infix [ i ] ) )
                    postfix [ j ++ ]  =  s [ top --] ;

             if ( F ( s[top] )  !=  G ( infix[i] ) )
                    s [++top ]  =  infix [ i ];
             else
                    top -- ;
        }

        while ( s[top]   !=   '#' )
             postfix [ j ++ ]  =  s [ top -- ] ;

        postfix [ j ]   =  '\0' ;
}
```

```
/* Insert the function: infix_2_postfix (infix, postfix) */
void infix_2_postfix ( char infix[], char postfix [] ) ;
```

```
/* Stack precedence function: F */
int F ( char symbol )
{
  switch ( symbol )
  {
    case '#' : return -1;

    case '+' :
    case '-' : return 2;

    case '*' :
    case '/' : return 4;

    case '$' :
    case '^' : return 5;

    default  : return 8;

    case '(' : return 0;
  }
}
```

```
/* Input precedence function: G */
int G ( char symbol )
{
  switch ( symbol )
  {
    case ')' : return 0;

    case '+' :
    case '-' : return 1;

    case '*' :
    case '/' : return 3;

    case '$' :
    case '^' : return 6;

    default  : return 7;

    case '(' : return 9;
  }
}
```

	infix[i]	F Stack	G I/P
	#	-1	-
	)	-	0
L	+ -	2	1
L	* /	4	3
R	^ $	5	6
L	operands	8	7
L	(	0	9

Precedence table

```
void main ( )
{
        char   infix[20],  postfix[20] ;

        printf ("Enter infix expr: " );
        scanf ("%s" , infix );

        infix_2_postfix ( infix,  postfix ) ;

        printf ("Postfix: " );
        printf ("%s\n" , postfix );
}
```

Test Case 1:

Enter infix expression: ((A+(B-C) *D)^E+F)

Postfix: ABC-D*+E^F+

Test Case 2:

Enter infix expression: X^Y^Z-M+N+P/Q

Postfix: XYZ^^M-N+PQ/+

How to convert from infix to prefix?

((A + $\underbrace{\text{(B - C)}}_{T1}$ * D) ^ E + F)

T1 = - B C

(($\underbrace{\text{A + T1 * D}}_{T2}$) ^ E + F)

T2 = * T1 D

($\underbrace{\text{(A + T2)}}_{T3}$ ^ E + F)

T3 = + A T2

($\underbrace{\text{T3 ^ E + F}}_{T4}$)

T4 = ^ T3 E

(T4 + F)

\+ T4 F

\+ ^ T3 E F

\+ ^ + A T2 E F

\+ ^ + A * T1 D E F

$\boxed{\text{+ ^ + A * - B C D E F}}$ Prefix Expression

or

$\boxed{\text{+ \$ + A * - B C D E F}}$ Prefix Expression

How to convert from infix to prefix?

X ^ $\underbrace{\text{Y ^ Z}}_{T1}$ - M + N + P / Q

T1 = ^ Y Z

$\underbrace{\text{X ^ T1}}_{T2}$ - M + N + P / Q

T2 = ^ X T1

T2 - M + N + $\underbrace{\text{P / Q}}_{T3}$

T3 = / P Q

$\underbrace{\text{T2 - M}}_{T4}$ + N + T3

T4 = - T2 M

$\underbrace{\text{T4 + N}}_{T5}$ + T3

T5 = + T4 N

$\underbrace{\text{T5 + T3}}$

\+ T5 T3

\+ + T4 N / P Q

\+ + - T2 M N / P Q

\+ + - ^ X T1 M N / P Q

$\boxed{\text{+ + - ^ X ^ Y Z M N / P Q}}$ Prefix Expression

or

$\boxed{\text{+ + - \$ X \$ Y Z M N / P Q}}$ Prefix Expression

What is the logic used while converting from infix to prefix?

Infix: A + (B - C) * D

D	*	)	C	-	B	(	+	A					

Stack[]	F (s[top])	>	G (infix[i])	Prefix[]
#	-1	>	7	
# D	8	>	4	D
#	-1	>	4	
# *	3	>	9	
# *)	0	>	7	
# *) C	8	>	2	C
# *)	0	>	2	
# *) -	1	>	7	
# *) - B	8	>	0	B
# *) -	1	>	0	-
# *)	0	>	0	
# *	3	>	2	*
#	-1	>	2	
# +	1	>	7	
# + A				D C B - * A
# +				+
#				+ A * - B C D

A + $\underbrace{\text{(B - C)}}_{T1}$ * D

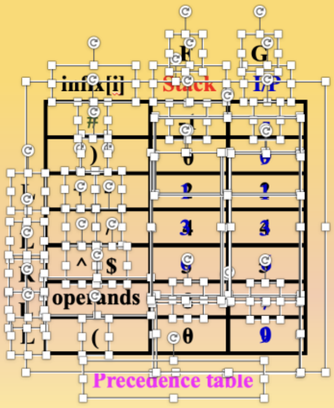
A + $\underbrace{\text{T1 * D}}_{T2}$

A + T2

\+ A T2

\+ A * T1 D

$\boxed{\text{+ A * - B C D}}$



How to design the algorithm to convert from infix to prefix?

Infix: A + (B - C) * D

D	*	)	C	-	B	(	+	A					

Stack[]	F (s[top]) > G (infix[i])	Prefix[]
#	-1 > 7	
# D	8 > 4	D
#	-1 > 4	
# *	3 > 9	
# *)	0 > 7	
# *) C	8 > 2	C
# *)	0 > 2	
# *) -	1 > 7	
# *) - B	8 > 0	B
# *) -	1 > 0	-
# *)	0 > 0	
# *	3 > 2	*
#	-1 > 2	
# +	1 > 7	
# + A		A
# +		+
#		

Prefix: D C B - * A +

Reversed: + A * - B C D

```
Algorithm infix_2_prefix ( infix, prefix )
top = -1
s [ ++top ] = '#'

∀ i = 0 to infix [ i ] != '\0'
    while ( F ( s[top] ) > G ( infix [ i ] ) )
        prefix [ j++ ] = s [ top -- ] ;

    if ( F ( s[top] ) != G ( infix[i] ) )
        s [++top ] = infix [ i ]
    else
        top --

while ( s[top] != '#' )
    prfix [ j++ ] = s [ top -- ] ;

prefix [ j ] = '\0'
```

```
/* Insert the function: infix_2_prefix (infix, prefix) */
void infix_2_prefix ( char infix[], char prefix [] )
{
        int     top, i, j = 0;
        char    s[20];

        top = -1;
        s [++top ] = '#' ;

        for ( i = 0; infix [ i ] != '\0'; i++ )
        {
            while ( F ( s[top] ) > G ( infix [ i ] ) )
                  prefix [ j ++ ] = s [ top --] ;

            if ( F ( s[top] ) != G ( infix[i] ) )
                  s [++top ] = infix [ i ];
            else
                  top -- ;
        }

        while ( s[top] != '#' )
            prefix [ j ++ ] = s [ top -- ] ;

        prefix [ j ] = '\0' ;
}
```

```
/* Insert the function: infix_2_prefix (infix, prefix) */
void infix_2_prefix ( char infix[], char prefix [] ) ;
```

```
/* Stack precedence function: F */
int  F ( char symbol )
{
  switch ( symbol )
  {
    case '#' : return -1;
    case ')' : return  0;

    case '+' :
    case '-' : return  1;

    case '*' :
    case '/' : return  3;

    case '$' :
    case '^' : return  6;

    default  : return  8;
  }
}
```

```
/* Input precedence function: G */
int  G ( char symbol )
{
  switch ( symbol )
  {
    case ')' : return 9;

    case '+' :
    case '-' : return 2;

    case '*' :
    case '/' : return 4;

    case '$' :
    case '^' : return 5;

    default  : return 7;

    case '(' : return 0;
  }
}
```

	infix[i]	F Stack	G I/P
	#	-1	-
	)	0	9
L	+ -	1	2
L	* /	3	4
R	^ $	6	5
L	operands	8	7
L	(	-	0

Precedence table

```
void strrev ( char dst [ ], char str[ ] ) ;
```

```
void main ()
{
        char      infix[20],  prefix[20] ;
        char      rev_infix[20], rev_prefix[20] ;

        printf ("Enter infix expr: " );
        scanf ("%s" , infix );

        strrev (  rev_infix, infix );

        infix_2_prefix ( rev_infix, rev_prefix ) ;

        strrev (  prefix,  rev_prefix);

        printf ("Prefix: " );
        printf ("%s\n" , prefix );
}
```

How to evaluate a postfix expression?

Postfix: | 6 | 3 | 2 | - | 5 | * | + | 2 | ^ | 3 | + | | | |

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

Stack	op2	op1	result = op1 ⊕ op2
6			
6 3			
6 3 2			
6 ~~3~~ ~~2~~ 6 1	2	3	result = 3 - 2 = 1
6 1 5			
6 ~~1~~ ~~5~~ 6 5	5	1	result = 1 * 5 = 5
~~6~~ ~~5~~ 11	5	6	result = 6 + 5 = 11
11 2			
~~11~~ ~~2~~ 121	2	11	result = 11 ^ 2 = 121
121 3			
~~121~~ ~~3~~ 124	3	121	result = 121 + 3 = 124

Result = 124

Infix: (6 + (3 - 2) * 5) ^ 2 + 3

Postfix: $6\ \underbrace{3\ 2\ -}_{1}\ 5\ *\ +\ 2\ \hat{}\ 3\ +$ 3 - 2 = 1

$6\ \underbrace{1\ 5\ *}_{5}\ +\ 2\ \hat{}\ 3\ +$ 1 * 5 = 5

$\underbrace{6\ 5\ +}_{11}\ 2\ \hat{}\ 3\ +$ 6 + 5 = 11

$\underbrace{11\ 2\ \hat{}}_{121}\ 3\ +$ 11 ^ 2 = 121

$\underbrace{121\ 3\ +}_{124}$ 121 + 3 = 124

How to design an algorithm to evaluate a postfix expression?

Postfix: | 6 | 3 | 2 | - | 5 | * | + | 2 | ^ | 3 | + | | | |

Stack	op2	op1	result = op1 ⊕ op2
6			
6 3			
6 3 2			
6 ~~3~~ ~~2~~ 6 1	2	3	result = 3 - 2 = 1
6 1 5			
6 ~~1~~ ~~5~~ 6 5	5	1	result = 1 * 5 = 5
~~6~~ ~~5~~ 11	5	6	result = 6 + 5 = 11
11 2			
~~11~~ ~~2~~ 121	2	11	result = 11 ^ 2 = 121
121 3			
~~121~~ ~~3~~ 124	3	121	result = 121+ 3 = 124

Result = 124

```
Algorithm   compute ( operand1,  operator,  operand2 )
switch ( operator )
      case '+' :  return  operand1 + operand2
      case '-' :  return  operand1 - operand2
      case '*' :  return  operand1 * operand2
      case '/' :  return  operand1 / operand2
      case '^' :
      case '$' :  return  pow ( operand1,  operand2 )
end switch

Algorithm   evaluate ( postfix )
top  =  -1
∀ i = 0  to  postfix [ i ] != '\0'
   if ( postfix[i] is digit )
        stack [++top ] =  postfix [ i ] - '0';
   else
        operand2  =  stack [ top-- ]
        operand1  =  stack [ top-- ]
        stack[++top ] = compute (operand1,  postfix[i], operand2 )
return  stack [ top-- ]
```


How to write a C function to evaluate a postfix expression?

```c
#include      <stdio.h>
#include      <math.h>
double compute (double operand1, char operator, double operand2 )
{
     switch ( operator )
     {
        case '+' : return  operand1  +  operand2;
        case '-' : return  operand1  -  operand2;
        case '*' : return  operand1  *  operand2;
        case '*' : return  operand1  /  operand2;
        case '^' :
        case '$' : return  pow (  operand1,  operand2 );
     }
}

double evaluate (char     postfix [] )
{
    int          i,  top  = -1 ;
    double      stack[20],  operand1,  operand2;

    for ( i = 0;  postfix [ i ] != '\0' ;  i++ )  {
        if ( postfix[i]  >=  '0'  &&  postfix[i]  <=  '9' )
            stack [++ top ]  =  postfix [ i ]  - '0';
        else {
            operand2 =  stack [ top -- ];
            operand1 =  stack [ top -- ];
            stack [++top] = compute (operand1,  postfix[i], operand2 );
        }
    }
    return  stack [ top -- ];
}
```

```c
void  main ( )
{
      char       postfix[20] ;
      double    result;

      printf ("Enter postfix expr: " );
      scanf ("%s" , postfix );

      result  =  evaluate ( postfix );

      printf ("Result = %lf\n" , result );
}
```

Test Case 1:

Enter postfix expression: 6 3 2 - 5 * + 2 ^ 3 +

Result = 124.0

Test Case 2:

Enter postfix expression: 1 2 + 3 – 2 1 + 3 $ –

Result = -27.0

[No Title]

Dr. Padma Reddy A.M | Sai Vidya Institute of Technology | Bengaluru | download: nandipublications.com, saividya.ac.in