# Chapter 3

# Arrays, Pointers and Structures

**LEARNING OBJECTIVES**

- *Arrays*
- *Array initialization*
- *Passing array elements to function*
- *Two dimensional arrays*
- *Syntax for 3D array declaration*
- *Pointers*
- *Pointer to pointer*
- *Pointer to void (generic pointer)*
- *Array of pointers*
- *Pointer to function*
- *Dynamic memory management*
- *Memory allocation function*
- *Realloc*
- *Structures*
- *Nesting of structures*
- *Array of structures*
- *Structures & functions*
- *Union*
- *Declaration*
- *Bit fields*

## Arrays

In *C* we have the following derived data types:

- Arrays
- Pointers
- Structures
- Unions

Imagine a problem that requires to read, process, and print 10 integers. We can declare 10 variables, each with different name. Having 10 different names creates a problem; we need 10 read and 10 write statements each for different variable.

### Definition

An array is a collection of elements of same data type. Array is a sequenced collection. So, we can refer to the elements in array as 0th element, 1st element, and so on, until we get the last element. The array elements are individually addressed with their subscripts/indices along with array name. We place subscript value in square brackets ([ ]) followed by array name. This notation is called indexing.

There is a powerful programming construct, loop, that makes array processing easy. It does not matter if there are 1, 10, 100 or 1000 elements.

We can also use a variable name in subscript, as the value of variable changes; it refers different elements at different times.

### Syntax of Array Declaration

```
Data_type array_name_[size];
```

Here, data type says the type of elements in collection, array_name is the name given to collection of elements and size says the number of elements in array.

**Example:** int marks[6];

Here, 'int' specifies the type of variable, marks specifies name of variable. The number 6 tells the dimension/size. The '[ ]' tells the compiler that we are dealing with array.

Accessing array elements: All the array elements are numbered, starting from 0, thus marks [3] is not the third, but the fourth element.

**Example:** marks[2] – 3rd element

marks[0] – 1st element

We can use the variable as index.

Thus marks[i] – ith element. As the value of i changes, refers different elements in array.

### Summary about Arrays

- An array is a collection of similar elements.
- The first element in array is numbered 0, and the last element is one less than the total size of the array.
- An array is also known as subscripted variable.
- Before using an array, its type and dimension must be declared.
- How big an array is, its elements are always stored in contiguous memory locations.
- Individual elements accessed by index indicating relative position in collection.
- Index of an array must be an integer.

### Array Initialization

#### *Syntax*

```
Data_type array_name[size] = {values};
```

**Example:**

```
int  n[6]= {2,4,8,12,20,25};  //  Array  ini-
tialized with list of values
int num[10] = {2,4,12,20,35};
// remaining 5 elements are initialized with
0
// values
int  b[10]  =  {0};  //  Entire  array  elements
initialized with 0.
```

**Note:**

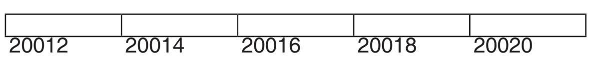
- Till the array elements are not given any specific value, they are supposed to contain garbage values.
- If the number of elements used for initialization is lesser than the size of array, then the remaining elements are initialized with zero.
- Where the array is initialized with all the elements, mentioning the dimension is optional.

## Array Elements in Memory

Consider the following declaration – int num[5].

What happens in memory when we make this declaration?

- 10 bytes get received in memory, 2 bytes each for 5 integers.
- Since array is not initialized, all five values in it would be garbage. This happens because the default storage class is auto. If it is declared as static, all the array elements would be initialized with 0.




**Note:** In *C*, the compiler does not check whether the subscript used for array exceeds the size of array.

Data entered with a subscript exceeding the array size will simply be placed in memory out size the array, and there will be no error/warning message to warn the programmer.

### *Passing array elements to function*

Array elements can be passed to a function by value or by reference.

**Example:** A program to pass an array by value:

```
void main( )
{
void display(int[  ]);// Declaration
int marks[ ] = {10,15,20,25,30};
display (marks);// function call
}
void display(int n[ ] )// function definition
{
int i;
for(i = 0 ; i < 5 ; i++)
printf("%d ", n[ i ] );
}
```

**Output:**

10 15 20 25 30

Here, we are passing the entire array by name. The formal parameter to receive is declared as an array, so it receives entire array elements.

To pass the individual elements of an array, we have to use index of element with array name.

**Example:** display (marks[ *i* ] ); sends only the ith element as parameter.

**Example:** A program to demonstrate call by reference:

```
void main( )
{
void display (int *);
int marks[ ] = {5, 10 15, 20, 25};
display(&marks[0]);
}
void display(int *p)
{
int i;
for(i = 0; i < 5; i++)
printf("%d ",*(p+i));
}
```

**Output:**

5 10 15 20 25

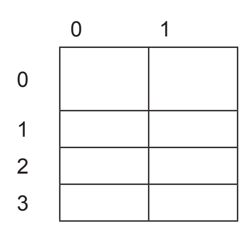
Here, we pass the address of very first element. Hence, the variable in which this address is collected (*p*) is declared as a pointer variable.

**Note:** Array elements are stored in contiguous memory location, by passing the address of the first element; entire array elements can be accessed.

## Two-dimensional Arrays

In *C* a two-dimensional array looks like an array of arrays, i.e., a two-dimensional array is the collection of one-dimensional arrays.

**Example:** `int x[4][2];`




By convention, first dimension says the number of rows in array and second dimension says the number of columns in each row.

In memory, whether it is one-dimensional or a two-dimensional array, the array elements are stored in one continuous chain.

The arrangement of array elements of a two-dimensional array in memory is shown below:

| *X*[0][0] | *x*[0][1] | *x*[1][0] | *x*[1][1] | *x*[2][0] | *X*[2][1] | *X*[3][0] | *X*[3][1] |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| 6000 | 6002 | 6004 | 6006 | 6008 | 6010 | 6012 | 6014 |

## Initialization

We can initialize two-dimensional array as one-dimensional array:

```
int a[4] [2] = {0,1,2,3,4,5,6,7}
```

The nested braces can be used to show the exact nature of array, i.e.,

```
int a[4][2] = {{0,1},{2,3}{4,5},{6,7}}
```

Here, we define each row as a one-dimensional array of two elements enclosed in braces.

**Note:** If the array is completely initialized with supplied values, then we can omit the size of first dimension of an array (the left most dimension).

- For accessing elements of multi-dimensional arrays, we must use multiple subscripts with array name.
- Generally, we use nested loops to work with multi-dimensional array.

## Multidimensional Arrays

C allows array of two or more dimensions and maximum numbers of dimensions a C program can have depends on the compiler, we are using. Generally, an array having one dimension is called 1D array; array having two dimensions is called 2D array and so on.

**Syntax:**
type array-name[*d*1] [*d*2] [*d*3] [*d*4]…[*dn*];
where dn is the size of last dimension.

**Example:**
int table[5][5][20];
float arr[5][6][5][6][5];
In our example array "table" is a 3D. (A 3D array is an array of array of array)

## Declaration and Initialization of 3D array

A 3D array can be assumed as an array of arrays; it is an array of 2D arrays and as we know 2D array itself is an array of 1D arrays. A diagram can help you to understand this.

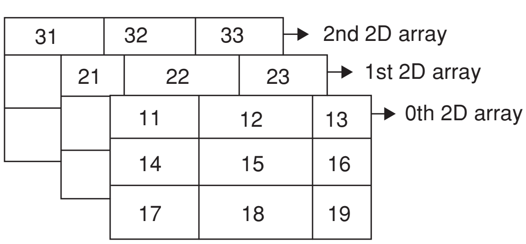



**Figure 1** 3D array conceptual view

**Example:**

```
void main( )
{
int i, j, k;
int arr [3] [3] [3] =
{
{11, 12, 13},
{14, 15, 16},
{17, 18, 19}
},
{21, 22, 23},
{24, 25, 26},
{27, 28, 29}
},
{31, 32, 33},
{34, 35, 36},
{37, 38, 39}
}
};
printf("3D Array Elements \n");
for (i = 0; i<3; i++)
{
for(j =0; j <3; j++)
{
for (k= 0; k<3; k++)
{
printf ("% d\t", arr[i][j][k]);
}
printf ("\n");
}
printf ("\n);
}
}
```

**Output:** 3D Array Elements

| | | |
|---|---|---|
| 11 | 12 | 13 |
| 14 | 15 | 16 |
| 17 | 18 | 19 |
| | | |
| 21 | 22 | 23 |
| 24 | 25 | 26 |
| 27 | 28 | 29 |
| | | |
| 31 | 32 | 33 |
| 34 | 35 | 36 |
| 37 | 38 | 39 |

## Syntax for 3D Array Declaration

data–type array–name [table] [row] [column];
To store values in any 3D array, first point to table number, row number and lastly to column number.

## Pointers

Pointer is a variable which contains address of another variable. *C*'s clever use of pointers makes it the excellent language.

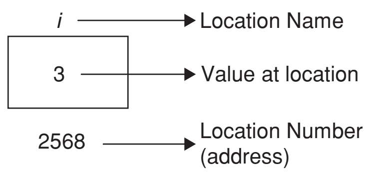
Consider the declaration:

```
int i = 3;
```

The declaration tells the *C* compiler to:

- Reserve space in memory to hold in integer value.
- Associate the name *i* with this memory location.
- Store the value 3 at this location.

Memory map is:




Computer may choose different location at different times for same variable. The important point is the address is a number.

The expression '&i' gives the address of variable '*i*'.

p = &i;

Assigns the address of '*i*' to variable '*p*'.

The variable '*p*' is declared as:

int **p*;

* tells the compiler that variable '*p*' is an address variable.

Memory map of i, **p* is –

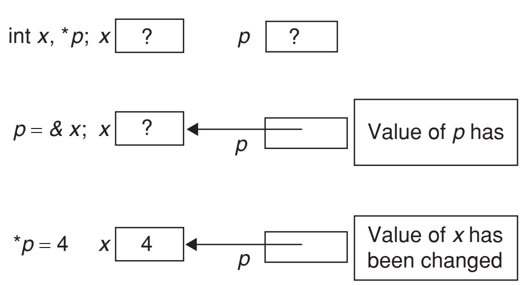
| **p* | *i* |
|---|---|
| 2568 | 3 |
| 2720 | 2568 |

Now, pointer '*p*' is referring to the variable '*i*'.

The variable '*i*' can be accessed in two ways:

- By using the name of variable.
- By using the pointer variable referring to location '*i*'.

The operator '*' can also be used along with pointer variable in expressions. The operator '*' acts as indirection operator.




Usage of '*p*' refers to value of '*p*', where as '**p*' refers to value at the address stored in '*p*', i.e., value of '*i*'.

**Example:**
```
int *p;
float *x;
char *ch ;
```

Here, *p*, *x* and ch are pointer variables, i.e., variables capable of holding address. Since addresses are always whole numbers, pointers would always contain whole numbers.

The declaration float **x* does not mean that *x* contains floating value, *x* will contain address of floating point variable. Similarly, 'ch' contains address of char value.

### Pointer to Pointer

We know, pointer is a variable that contains address of another variable. Now this variable address might be stored in another pointer. Thus, we now have a pointer that contains address of another pointer, known as pointer to pointer.

**Example:**

```
void main()
{
int i = 3, *p, **q;
p = &i;
q =&p;
printf("\n Address of i = %u", &i);
printf("\n Address of i = %u", p);
printf("\n Address of i = %u", *q);
printf("\n Address of p= %u", &p);
printf(\n Address of p= %u, q);
printf("\n Address of q = %u", &q)
printf('\n value of i= %d",i);
printf('\n value of i= %d",*(&i));
printf('\n value of i= %d",*p);
printf('\n value of i= %d",**q);
}
```

If the memory map is

| ***q* | **p* | *i* |
|---|---|---|
| 2010 | 2000 | 3 |
| 2050 | 2010 | 2000 |

Then the output is:

Address of *i* = 2000
Address of *i* = 2000
Address of *i* = 2000
Address of *p* = 2010
Address of *p* = 2010
Address of *q* = 2050
Value of *i* = 3
Value of *i* = 3
Value of *i* = 3
Value of *i* = 3

**Note:** We can extend pointer to a pointer to pointer. In principal, there is no limit on how far we can go on extending this definition.

### Pointers for Inter-function Communication

We know that functions can be called by value and called by reference.

- If the actual parameter should not change in called function, pass the parameter-by value.

- If the value of actual parameter should get changed in called function, then use pass-by reference.
- If the function has to return more than one value, return these values indirectly by using call-by-reference.

**Example:** The following program demonstrates how to return multiple values.

```
void main( )
{
void areaperi(int, int *, int *);
int r;
float a,p;
printf("\n Enter radius of a circle");
scanf("%d", &r);
areaperi(r, &a, &p);
printf("Area = %f", a);
printf("\n Perimeter = %f", p);
}
void areaperi(int x, int *p, int *q)
{
*p = 3.14*x*x;
*q = 2 * 3.14*x;
}
```

**Output:**
Enter radius of circle 5
Area = 78:500000
Perimeter = 31.400000

**Compatibility:** Pointers have a type associated with them. They are not just pointer types, but rather are pointers to a specific type. The size of all pointers is same, which is equal to size of int. Every pointer holds the address of one memory location in computer, but size of variable that the pointer references can be different.

## Pointer to Void (Generic Pointer)

A pointer to void is a generic type; this can point to any type. Its limitation is that the pointed data cannot be referenced directly. Since void pointer has no object type, so its length is undetermined; it cannot be dereference unless it is cast.

**Example:** The following example demonstrates generic pointer.

```
void main ( )
{
int a = 10;
float x = 5.7;
void *p;
p = &a;
printf("\n value of a = %d", *((int*)p));
p= &x;
printf ("\n value of x = % f", *((float *)p));
}
```

**Output:**
value of $a$ = 10
value of $x$ = 5.700000

## Operations can be Performed on Pointers

1. Addition of a number to a pointer.

**Example:**
```
int i = 4, *j, *k;
  j =&i;
  j = j +1;
  k = j +5;
```

2. Subtraction of a number from a pointer.

**Example:**
```
int i = 4, * j, * k;
  j = &i; j = j -1;
  k = j - 3;
```

3. Subtraction of one pointer from another. One pointer variable can be subtracted from another (provided both variables point to same array elements). The resulting value indicates the number of bytes (elements) separating (the corresponding array elements).

**Example:**
```
void main ( )
{
int a[ ] = {5,10,15,20,25} ,*i, *j;
i = &a[0];
j = &a[4];
printf("%d, %d", j-i,*j-*i);
}
```

**Output:** 4, 20
The expression *j-i* prints 4 but not 8. because j and i pointing to integers that are 4 integers apart.

4. Comparison of two pointer variables. Pointer variables can be compared provided both pointing to the same data type.

**Notes:** Do not attempt the following operations on pointers:
1. Addition of two pointers.
2. Multiplication of a pointer with a number or another pointer.
3. Division of a pointer with a number or another pointer.

### *Important points about pointer arithmetic*

- A pointer when incremented always points to an immediately next location.
- A pointer when decremented always points to an element precedes the current element.

**Notice the difference with:**
```
(*p)++
```
Here, the expression would have been evaluated as the value pointed by *p* increased by one. The value of *p* would not be modified if we write

$$\boxed{*p{+}{+} = *q{+}{+};}$$

Because ++ has a higher precedence than *, both *p* and *q* are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to **p* is *q before both *p* and *q* are increased. And then both are increased, it would be equivalent to

```
*p = *q
++p;
++q;
```

### *Implementation of arrays in C*

Array name is the pointer to the first element in array. The following discussion explains how pointers are used for implementing arrays in *C*.

```
int n[ ] = {10,20,30,40,50};
```

| *n* | | | | | |
|---|---|---|---|---|---|
| | 10 | 20 | 30 | 40 | 50 |
| | 5512 | 5514 | 5516 | 5518 | 5520 |

- We know that mentioning the array name gets the base address.

  `int *p = n;`

  Now 'p' points to $0^{th}$ element of array '*n*'.
- 0th element can be accessed as *array_ name.

  `int  x = *n;`

  `stores n[0] into 'x'.`
- we can say that *array _ name and *(array _ name+0) are same. This indicates the following are same.

  ```
  num[i]
  *(num + i)
  *(i+num)
  ```

  (num is an array; i is an index)

## Array of Pointers

The way there can be an array of ints or array of floats, similarly there can be an array of pointers. An array of pointers is the collection of addresses.

These arrays of pointers may point to isolated elements or an array element.

**Example 1:** Array of pointers pointing to isolated elements:

```
int i = 5, j=10, k =15;
int *ap[3];
ap[0] = &i; ap[1] = &j; ap[2] = &k;
```

**Example 2:** Array of pointers pointing to elements of an array:

```
int a[ ] = {0,20,45,50,70};
int *p[5], i ;
for(i = 0; i <5 ; i++ )
p[i] = &a[i] ;
```

**Example 3:** Array of pointers pointing to elements of different arrays;

```
int a[ ] = {5,10,20,25};
int b[ ] = {0,100,200,300,400};
int c[ ] = {50,150,250,350,450};
int *p[3];
p[0] = a; p[1] = b; p[2]=c;
```

**Example 4:** Array of pointers pointing to 0th element of each row of a two-dimensional array

```
int a[3][2] = {{1,2} {3,4}, {5,6}};
int *p[3];
p[0] = a[0]; p[1] =a[1];p[2] = a[2];
```

## Pointer to Function

Function is a set of instructions stored in memory, so the function also contains the base address. This address can hold by using a pointer called pointer to function.

**Syntax:**

```
return_type  (*function_pointer)(parameter –
list );
```

**Example:** `int (*fp)(float, char, char);`

**Example:**

```
// pointer to functions
# include <iostream>
Using name space std;
int addition(int a, int b)
{
return (a + b);
}
int subtraction(int a, int b)
{
return (a – b) ;
}
int operation (int x, int y, int (*funtocall)
(int, int))
{
int g;
g = (*functocall)(x, y);
return (g);
}
int main( )
{
int m, n;
int (*minus)(int, int) = substraction;
m = operation(7, 5, addition);
n = operation(20, m, minus);
cout < < n;
return 0;
}
```

In the example, minus is a pointer to a function that has two parameters of type int. It is immediately assigned to point to the function subtraction, all in a single line.

**Example:** Program to demonstrate function pointer

```
int add(int, int);
int sub(int, int);
void main( )
{
Int (*fp) (int, int);
fp = add;
```

```
printf("\n 4+5=%d", fp(4,5));
fp = sub;
printf ("\n 4 - 5 = %d", fp(4,5));
}
int add(int x, int y)
{
return x + y;
}
int sub(int x,int y)
{
return x - y;
}
```

**Output:** 4 + 5 = 9
4 – 5 = –1

***Pointer to structure*** The main usage of pointer to structure is we can pass structure as parameter to function as call by reference.

The other usage is to create linked lists and other dynamic data structures which depend on dynamic allocation.

Consider the declaration

struct employee

```
{
char name[20];
Int age;
float salary;
};
```

struct employee * *p*;

Variable of structures can be accessed using '.' Operator (or) → operator that is

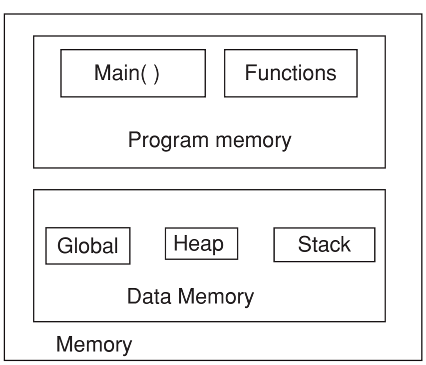
(*$p$).age = 20 ; (or) $p$ → age = 20;

(*$p$).salary = 40, 231.0; (or) $p$ → salary = 40,231.0;

## Dynamic Memory Management

We can allocate the memory to objects in two ways—static and dynamic allocation. Static memory allocation requires declaration and definition of memory fully specified in the source program. The number of bytes required cannot be changed during run time. Dynamic memory allocation uses predefined functions to allocate and de-allocate memory for data dynamically during the execution of program.

We can refer to dynamically allocated memory only through pointers. Conceptual view of memory:




### Memory Allocation Function

- Static memory allocation uses stack memory for variables.
- Dynamic memory management allocates memory from heap.

The following are the four memory management functions available in alloc.h and stdlib.h.

1. **Malloc (Block memory allocation):** Malloc function allocates block of memory that contained the number of bytes specified in parenthesis. It returns 'void' pointer to the first byte of allocated memory. The allocated memory is not initialized. If the memory allocation is not successful then it return NULL pointer.
   **Declaration**
   void *malloc (size_t size);
   The type size_t is defined as unsigned int in several header files including `stdio.h.`

   **Syntax:** pointer = (type*) malloc(size );
2. **Calloc (contiguous memory allocation):** Calloc is primarily used to allocate memory for arrays. It initializes the allocated memory with null characters.

   **Declaration:** void *calloc (size_t ele_count, size_t ele_size);

   **Syntax:** ptr = (type*)calloc(ele-count,ele-size);
3. **Realloc (reallocation of memory):** The realloc function is highly inefficient. When given a pointer to a previously allocated block of memory, realloc changes the size of block by deleting or extending the memory at the end of block. If the memory cannot be extended, then realloc allocates completely new block, copies the contents from existing memory location to new location, and deletes the old location.

   **Declaration:** void *realloc (void *ptr, size_t new_ size);

   **Syntax:** ptr = (type*)realloc(ptr, new_ size);
4. **Free (Releasing memory):** When the memory allocated by malloc, calloc or realloc is no longer needed, they can be freed using the function free( ).

   **Declaration:** void free(void *ptr);

   **Syntax:** free(ptr);

Free function de-allocates complete memory referenced by the pointer. Part of the memory block cannot be de-allocated.

## Structures

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. When we require using a collection of different data items of different data types, we can use a structure.

- Structure is a method of packing data of different types.
- A structure is a convenient method of handling a group of related data items of different data types.

**Syntax for declaration**

```
struct sturct_name
{
Data_type_1 var1;
Data_type_2 var2;
:
Data_type_n varn;
};
```

**Example:**

```
struct lib - books
{
char title [20];
char author[15];
int pages;
float price;
};
```

The keyword struct declares a structure to hold the details of four fields namely title, author, pages and price, these are members of the structures.

We can declare structure variables using the tag name anywhere in the program.

**Example:** struct lib – books book1, book2, book3;

- Declares book1, book2, book3 as variables of type struct lib _ books, each declaration has four elements of the structure lib_ books.

  Memory map of book1:

| Book1 | | |
|---|---|---|
| | Title | 20 bytes |
| | Author | 15 bytes |
| | Pages | 2 bytes |
| | Price | 4 bytes |

- Memory will not be allocated to the structure until it is instantiated. i.e., till the declaration of a variable to structure.
- To access the members of a structure variable, *C* provides the member of (.) operator.

**Example:** To access author of book 1 – book1. author

**Syntax:** `structure_var.member_name;`

- The structures can also be initialized as any other variable of C.

**Example:** struct lib-books book4={“Let us *C*”, “yashwanth”, 450, 200.95};

**Note:** The values must provide in the same order as they appear in structure declaration.

- One structure variable can be assigned to another structure variable.
- Structure variables cannot be compared.

**Example:**

```
# include <stdio.h>
void main( )
{
Struct s1{
int id_no;
char name[20];
char address[20];
char combination[3];
   int age;
   } newstudent;
printf (“ Enter student Information”);
printf (“Enter student id _ no”);
scanf (“%d”, &newstudent.id_no):
printf (“ Enter the name of the student”);
scanf (“%s”, & newstudent.name);
printf (“ Enter the address  of the student”);
scanf (“%s”, &newstudent.address);
printf(“Enter   the   combination   of   the
student”)’;
scanf(“%s”, &newstudent.combination”);
printf (“ Enter the age of student);
scanf (“%d “, &newstudent.age”);
printf (“ student information”);
printf  (“ student id_no = %d”, newstudent.
id - no);
printf(“student  name  =  %s”,  newstudent.
name);
printf(“student address = %s“, newstudent.
address);
printf (“students combination = %s”, newstu-
dent. combination);
printf(“Age  of  student  =  %d”,  newstudent.
age);
}
```

## Nesting of Structures

The structures can be nested in two ways:

- Placing the structure variable as a member in another structure declaration.
- Declaration of the entire structure in another structure.

**Example:**

```
struct date
{
int day;
int month;
int year;
};
struct student
{
int id_no;
char name[20];
char address [20];
int age;
structure date doa;
} oldstudent, newstudent;
```

The structure ‘student’ contains another structure date as one of its members.

To access the day of date of admission (doa) of old student – oldstudent.doa.day.

**Example:**

```
struct outer
{
```

```
int o1;
float o2;
struct inner
{
int i1;
float i2;
};
} out1, out2;
```

The innermost members in a nested structure can be accessed by chaining all the concerned structure variables, from outermost to innermost; accessing i1 for out1-out1. inner.i1;

## Array of Structures

It is possible to define an array of structures. For example, if we are maintaining information of all the students in the college and if 100 students are studying in the college, we need to use an array than single variables.

**Example:**

```
structure information
{
int id _ no;
char name[20];
char address[20];
char combination[3];
int age;
}
student[100];
```

**Example:**

```
# include <stdio.h>
{
struct info
{
int  id _ no;
char name[20];
char address[20];
char combination[3];
int age;
}
struct info std[100];
int, i ,n;
printf (" Enter the number of students");
scanf ("%d", &n);
scanf("Enter id_no, name, address, combina-
tion and age");
for (i = 0; i<n; i ++)
scanf("  %d  %s  %s  %s  %d",  &std[i].id_no,
std[i].name, std[i].address,
std[i]. combination,&std [i].age);
printf("student information");
for (i = 0 ; i < n; i ++)
printf("%d  %s  %s  %  s  %  d",  std[i].id_no,
std[i].name, std[i].address, std[i]. combi-
nation, std[i]. age);
```

## Structures and Functions

- An entire structure can be passed as a parameter like any other variable.
- A function can also return a structure variable.

**Example:**

```
# include <stdio.h>
struct employee
{
int emp_id;
char name[25];
char department[10];
float salary;
};
void main( )
{
static struct employee emp1 = {
12, "shyam", "computer", 7500.00};
/* sending entire employee structure */
display(emp1);
}
/* function to pass entire structure vari-
able */
display(empf)
struct employee empf
{
printf (" %d %s % s %f", empf.empid, empf.
name, empf.department, empf.salary);
}
```

# Union

Union, like structure contains members whose individual data types may differ from one another. The members that compose union all share the same storage area within the computer's memory whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory.

## Declaration

```
union item
{
int m;
float p;
char c;
}Code;
```

This declares a variable code of type union item.

The union contains three members each with a different data type. However, we can use only one of them at a time. The compiler allocates a piece of storage that is large enough to access a union member; we can use the same syntax that we use to access structure members, i.e.,

```
Code.m
Code.p
Code.c
```

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored.

**Example:**

```
union marks
{
float perc;
char grade;
}
main( )
{
union marks student1;
student1.perc = 98.5;
printf("marks are %f address is %16ℓu", stu-
dent1.perc, &student1. perc);
student1. grade = 'c';
printf("grade is %c address is %16ℓu", stu-
dent1. grade, &student1. grade);
}
```

**Example:**

```
# include <stdio.h>
void main ( )
{
Union u_example
{
float decval;
int p_num;
double my_value;
}U1;
U1.my_value = 125.5;
U1.pnum = 10;
U1.decval = 1000.5f;
printf("decval = %f pnum = %d my_value = % lf
", U1. decval, U1.pnum, U1.my_value);
printf(" U1 size = %d decval size =%d,
pnum size = %d my-value size = % d",
sizeof (U1), sizeof (U1.decval), sizeof
(U1.pnum), sizeof (U1.my_value));
       }
```

## Bit Fields

When a program variable '$x$' is declared as int, then '$x$' takes the values from ($-2^{15}$) to ($2^{15}-1$), if $x$ in the program takes only two values, 1 and 0, which requires only one bit, then the remaining 15 bits are waste.

In order to not to have this wastage, we can use bit fields with the several variables with the small enough maximal values, which can pack into a single memory location

**Example:**

```
struct student
{
Int gender : 1 ; // gender takes only 0,1
values
Int marriage : 2 ; // marriage takes 4(0, 1,
2, 3) values
Int marks : 7 ; // marks takes values from
0 - 127
}
```

# Exercises

## Practice Problems I

***Directions for questions 1 to 15:*** Select the correct alternative from the given choices.

**1.** Output of the following C program is

```
intF(int x, int *py, int **pz)
{
int y, z;
** pz+= 1;
z = *pz;
*py+= 2;
y = *py;
x+ = 3;
return x+y+z;
}
void main( )
{
int c, *b, **a ;
c = 4;
b = &c;
a = &b;
printf( "%d", F(c, b, a));
}
```

(A) 30 (B) 22
(C) 20 (D) Error

**2.**
```
main( )
{
    char  *ptr;
    ptr = "Hello World";
     printf("%c\n",*&*ptr);
}
```

Output of the above program is
(A) Garbage value
(B) Error
(C) H
(D) Hello world

**3.**
```
#include <stdio.h>
main( )
{
register a =10;
char b[ ] = "Hi";
printf("%s %d ", b, a);
}
```

Output is
(A) Hi 10 (B) Error
(C) Hi (D) Hi garbage value

4.
```
main ( )
{
    int fun( ) ;
    (*fun) ( ) ;
}
int fun( )
{ printf("Hello") ;
}
```
(A) Hello (B) Error
(C) No output (D) H

5. Let $B$ be a two-dimensional array declared as
$B$ : array[1...10] [1...15] of integer;

Assuming that each integer takes one memory location the array is stored in row major order and the first element of the array is stored at location 100, what is the address of the element $B[i]$ $[j]$?
(A) $15i + 10j + 84$ (B) $15i + j - 16$
(C) $15i + j$ (D) $15i + j + 84$

6. Consider the following $C$ program which is supposed to compute the transpose of a given 4 × 4 matrix $M$. Note that, there is a $Y$ in the program which indicates some missing statements. Choose the correct option to replace $Y$ in the program.

```
# include <stdio.h>
int M[4][4] = { 8, 10, 9, 16, 12, 13, 11,
15, 14, 7, 6, 3, 4, 2, 1, 5 };
main( )
{
   int i, j, temp;
    for (i = 0; i<4; ++i)
     {
             Y
     }
     for (i=0; i<4; ++i)
     for (j=0; j<4; ++j)
      printf("%d", M[i] [j]);
}
```
(A)
```
for (j=0; j<4; ++j)
    {
       M[j] [i] = temp;
        temp = M[j][i];
         M[j][i] = M[i][j];
        }
```
(B)
```
for (j=0; j<4; ++j)
    {
       temp = M[j][i];
       M[i][j] = M[j][i];
        M[j][i] =  temp;
        }
```
(C)
```
for (j=i; j<4; ++j)
    {
       temp = M[i][j];
       M[i][j] = M[j][i];
        M[j][i] =  temp;
        }
```
(D)
```
for (j=i; j<4; ++j)
        {
       M[i][j] = temp;
        temp = M[j][i];
       M[j][i] = M[i][j] ;
        }
```

7. Consider the C program shown below:

```
# include <stdio.h>
# define print(a) printf("%d", a)
int a;
void z(int n)
{
  n += a;
print (n);
}
void x(int *p)
{
   int a = *p+2;
   z(a) ;
*p = a;
   print(a);
}
main(void)
  {
     a = 6;
     x(&a);
     print(a);
  }
```
The output of this program is
(A) 14 8 6 (B) 16 6 6
(C) 8 6 6 (D) 22 11 12

8. Consider the program below:

```
# include <stdio.h>
int fun(int n, int *p)
{
    int x,y;
    if (n<=1)
     {
         *p = 1;
          return 1;
    }
x = fun(n-1, p);
y = x +p;
*p = x;
return y;
}
int main( )
    {
        int a =15;
    printf( "%d\n", fun(5, &a));
    return 0;
    }
```
The output value is
(A) 14 (B) 15
(C) 8 (D) 95

**9.** Consider the following C program segment

```
char p[20] ;
int i;
char *s = "string" ;
int l = strlen(s);
for (i=0; i<l; i++)
p[i] = s[l - i] ;
printf("%s", p) ;
```

The output of the program is

(A) string
(B) gnirt
(C) gnirts
(D) No output is printed

**10.**
```
# include <stdio.h>
main( )
{
    struct AA
      {
            int A = 5;
             char name[ ] = "ANU";
      };
    struct  AA  *p  =  malloc(sizeof(struct
    AA));
                printf("%d",p->A);
                printf("%s",p->name);
}
```

Output of the program is

(A) 5 ANU
(B) Runtime error
(C) Compiler error
(D) Linker error

**11.** The declaration

```
union u_tag {
int ival;
float fval;
char sval;
} u;
```

denotes *u* is a variable of type *u*_tag and

(A) *u* can have a value of int, float and char
(B) *u* can represent either integer value, float value or character value at a time
(C) *u* can have a value of float but not integer
(D) None of the above

**12.** If the following program is run from command line as myprog 1 2 3, what would be the output?

```
main (int argc, char *argv[ ])
{
int i;
i = argv [1] + argv [2] - argv [3];
printf ("%d", i);
}
```

(A) 123 (B) 6
(C) 0 (D) Error

**13.** The following C program is run from the command line as
myprog one two;
what will be the output?

```
main (int argc, char *argv [ ])
{
printf ("%c",**++argv);
}
```

(A) m (B) o
(C) myprog (D) one

**14.** The following program

```
change(int *);
main( ) {
int a = 4;
change(a);
printf ("%d", a);
}
change(a)
int a;
{
printf("%d", a);
}
```

Outputs

(A) 44 (B) 55
(C) 34 (D) 22

**15.** What is the output of the following program:

```
main( )
{
    const int x = 10;
    int *ptrx;
    ptrx = &x;
    *ptrx = 20;
    printf ("%d", x);
}
```

(A) 5 (B) 10
(C) Error (D) 20

## Practice Problems 2

***Directions for questions 1 to 11:*** Select the correct alternative from the given choices.

1. The following program segment

```
int *i;
*i = 10;
```

(A) Results in run time error
(B) Is a dangling reference
(C) Results in compilation error
(D) Assigns 10 to *i*

2. A $m \times n$ matrix is stored in column major form. The expression which accesses the $(ij)$th entry of the same matrix is
(A) $n \times (j - 1) + i$
(B) $m \times (j - 1) + i$
(C) $n \times (m - 1) + ij$
(D) $m \times (n - 1) + j$

3. int $* S[a]$ is 1D array of integers, which of the following refers to the third element in the array?
(A) $*(S + 2)$ (B) $*(S + 3)$
(C) $S + 2$ (D) $S + 3$

4. If an array is declared as char a[10][12]; what is referred to by *a*[5]?
(A) Pointer to 3rd Row
(B) Pointer to 4th Row
(C) Pointer to 5th Row
(D) Pointer to 6th Row

5. The following code is run from the command line as myprog 1 2 3. What would be the output?

```
main(int argc, char *argv[ ])
{
    int i, j = 0;
    for (i = 1; i < argc; i++)
    j = j + atoi (argv [i]);
    printf ("%d", j);
}
```

(A) 123 (B) 6
(C) Error (D) "123"

6. What will be the following C program output?

```
main (int argc, char *argv[ ], char *env
[ ]) {
int i;
for(i = 1; i < argc; i++)
printf ("%s", env[i]);
}
```

(A) List of all arguments
(B) List of all path parameters
(C) Error
(D) List of environment variables

7. The declaration

```
enum colors {
    red,
    blue,
    yellow = 1,
    green
};
```

assigns the value 1 to
(A) Red and Yellow
(B) Blue
(C) Red and blue
(D) Blue and yellow

8. What would be the output of the following program?

```
sum = 0;
for (i = -10; i < 0; i++)
    sum = sum + abs(i);
printf ("%d", sum);
```

(A) 100 (B) −505
(C) 55 (D) −55

9. An integer occupies 2 bytes of memory, float occupies 4 bytes and character occupies 1 byte. A structure is defined as:

```
struct tab {
    char a;
    int b;
    float c;
} table [10];
```

Then the total memory requirement (in bytes) is
(A) 14 (B) 70
(C) 40 (D) 100

10. What are the values of u1 and u2?

```
int u1, u2;
int x = 2;
int *ptr;
u1 = 2*(x + 10);
ptr = &x;
u2 = 2*(*ptr + 10);
```

(A) $u1 = 8, u2 = 16$
(B) $u1 = 23, u2 = 24$
(C) $u1 = 24, u2 = 24$
(D) None of the above

11. What is the output?

```
func(a, b)
int a, b;
{
return (a = (a = = b));
}
main ()
{
int process(), func();
printf("The value of process is %d", pro-
cess (func,3,6));
}
process (pf, val1, val2)
int (*pf) ();
```

```
int val1, val2;
{
return ((*pf) (val1, val2));
}
```

(A) The value of process is 0
(B) The value of process is 3
(C) The value of process is 6
(D) Logical error

## Previous Years' Questions

**1.** Consider the following program in C language:

```
# include < stdio. h>
main ()
{
int i;
int *pi = &i;
scanf ("%d", pi);
printf("%d\n", i + 5);
}
```

Which one of the following statement is TRUE? **[2014]**

(A) Compilation fails
(B) Execution results in a run-time error
(C) On execution, the value printed is 5 more than the address of variable *i*.
(D) On execution, the value printed is 5 more than the integer value entered.

**2.** Consider the following C function in which size is the number of elements in the array *E*:

```
int MyX (int *E, unsigned int size)
{
int Y = 0;
int Z;
int i, j, k;
for (i = 0; i < size; i++)
Y = Y + E[i];
for (i = 0; i < size; i++)
for (j = 1; j < size; j++)
{
Z = 0;
for (k = i; k < = j; k++)
Z = Z + E[k];
if (Z > Y)
Y = Z;
}
return Y;
}
```

The value returned by the function My X is the **[2014]**

(A) maximum possible sum of elements in any sub - array of array *E*.
(B) maximum element in any sub-array of array *E*.
(C) sum of the maximum elements in all possible sub-arrays of array *E*.
(D) the sum of all the elements in the array *E*.

**3.** The output of the following C program is ______ **[2015]**

```
void f1 (int a, int b)  {
        int c;
        c=a; a=b; b=c;
}
void f2(int *a, int *b)  {
        int c;
        c=*a; *a=*b; *b=c;
}
int main ( ) {
        int a=4, b=5, c=6;
        f1 (a, b);
        f2 (&b, &c);
        printf("%d", c-a-b);
}
```

**4.** What is the output of the following C code? Assume that the address of *x* is 2000 (in decimal) and an integer requires four bytes of memory. **[2015]**

```
int main ( ) {
  unsigned int x[4] [3] =
  { {1, 2, 3}, {4, 5, 6}, {7, 8, 9},
  {10, 11, 12}};
  printf ("%u, %u, %u", x + 3, *(x +
  3), *(x + 2) + 3);
}
```

(A) 2036, 2036, 2036
(B) 2012, 4, 2204
(C) 2036, 10, 10
(D) 2012, 4, 6

**5.** Consider the following function written in the C programming language. **[2015]**

```
void foo(char *a {
        if ( *a && *a != ' '){
                foo(a + 1);
                putchar(*a);
        }
}
```

The output of the above function on input "ABCD EFGH" is

(A) ABCD EFGH
(B) ABCD
(C) HGFE DCBA
(D) DCBA

**6.** Consider the following C program segment. **[2015]**

```
#include <stdio.h>
int main()
{
    char s1[7] = "1234", *p;
```

```
    p = s1 + 2;
    *p = '0';
    printf("%s", s1);
}
```

What will be printed by the program?

(A) 12 (B) 120400

(C) 1204 (D) 1034

**7.** Consider the following C program **[2015]**

```
#include<stdio.h>
int main ( )
{
static int a[ ] = {10, 20, 30, 40, 50};
static int *p[ ] = {a, a+3, a+4, a+1, a+2};
int **ptr = p;
ptr++;
printf("%d%d", ptr-p, **ptr);
}
```

**8.** Consider the following C program. **[2016]**

```
void f (int, short);
void main( )
{
int i = 100;
short s = 12;
short *p = &s;
_____; // call to f( )
}
```

Which one of the following expressions, when placed in the blank above, will **NOT** result in a type checking error?

(A) $f(s,*s)$ (B) $i = f(i,s)$

(C) $f(i,*s)$ (D) $f(i,*p)$

**9.** Consider the following *C* program. **[2016]**

```
# include<stdio.h>
void mystery (int *ptra, int *ptrb) {
int *temp;
temp = ptrb;
ptrb = ptra;
ptra = temp;

}
int main ( ) {
int a = 2016, b = 0, c = 4, d = 42;
mystery (&a, &b);
if (a < c)
    mystery(&c, &a);
mystery (&a, &d);
printf("%d\n", a)
}
```

The output of the program is _____.

**10.** The following function computes the maximum value contained in an integer array $p$ [ ] of size $n$ ($n >= 1$). **[2016]**

```
int max (int *p, int n) {
    int a = 0, b = n – 1;
    while (_____) {
        if (p [a] < = p [b]) {a = a+1;}
        else             { b = b – 1;}
    }
    return p[a];
}
```

The missing loop condition is

(A) $a\ != n$

(B) $b\ != 0$

(C) $b > (a + 1)$

(D) $b\ != a$

**11.** The value printed by the following program is ___. **[2016]**

```
void f (int* p, int m) {
m = m +5;
*p = *p + m;
return;
}
void main () {
int i = 5, j = 10;
f(&i, j);
print f ("%d", i +j);
}
```

**12.** Consider the following program: **[2016]**

```
int f (int *p, int n)
{   if (n < = 1) return 0;
else return max (f (p +1, n – 1), p [0] – p [1] );
}
int main ()
{
int a[ ] = {3,5,2,6,4};
printf ("%d", f(a,5));
}
```

Note: *max (x,y) returns the maximum of x and y.*

The value printed by this program is ______

**13.** Consider the following C code:

```
# include <stdio.h>
int *assignval (int *x, int val) {
     *x = val;
     return x;
}
void main ( )  {
     int *x = malloc (sizeof (int));
     if (NULL == x) return;
      x = assignval (x, 0);
      if (x)  {
           x  =  (int  *)  malloc
           (sizeof (int));
           if (NULL == x) return;
           x = assignval (x, 10);
      }
     printf("%d\n", *x);
     free (x);
}
```

The code suffers from which one of the following problems: **[2017]**

(A) compiler error as the return of malloc is not typecast appropriately
(B) compiler error because the comparison should be made as x == NULL and not as shown
(C) compiles successfully but execution may result in dangling pointer
(D) compiles successfully but execution may result in memory leak

**14.** Consider the following C program.

```
# include <<stdio.h>
# include <<string.h>
void printlength (char *s, char *t)
{
   unsigned int c = 0;
   int  len  =  ((strlen(s)  -  strlen
   (t))  > c)  ? strlen  (s)  : strlen
   (t);
   printf ("%d\n", len);
}
void main ( ) {
   char *x = "abc";
   char *y = "defgh";
   printlength (x, y);
}
```

Recall that strlen is defined in string.h as returning a value of type size_t, which is an unsigned int. the output of the program is _________. **[2017]**

**15.** Given the following binary number in 32-bit (single precision) IEEE-754 format:

00111110011011010000000000000000

The decimal value closest to this floating-point number is **[2017]**

(A) $1.45 \times 10^{1}$ (B) $1.45 \times 10^{-1}$
(C) $2.27 \times 10^{-1}$ (D) $2.27 \times 10^{1}$

**16.** Match the following:

| | |
|---|---|
| (P) static char var; | (i) Sequence of memory locations to store addresses |
| (Q) m = malloc (10); m = NULL; | (ii) A variable located in data section of memory |
| (R) char *ptr [10]; | (iii) Request to allocate a CPU register to store data |
| (S) register int var1; | (iv) A lost memory which cannot be freed |

**[2017]**

(A) P → (ii), Q → (iv), R → (i), S → (iii)
(B) P → (ii), Q → (i), R → (iv), S → (iii)
(C) P → (ii), Q → (iv), R → (iii), S → (i)
(D) P → (iii), Q → (iv), R → (i), S → (ii)

**17.** Consider the following function implemented in C:

```
void printxy (int x, int y) {
     int *ptr;
     x = 0;
     ptr = &x;
     y = *ptr;
     *ptr = 1;
     printf ("%d, %d" x, y);
}
```

The output of invoking printxy (1, 1) is **[2017]**

(A) 0, 0 (B) 0, 1
(C) 1, 0 (D) 1, 1

**18.** Consider the following snippet of a C program. Assume that swap (&x, &y) exchanges the contents of x and y.

```
int main ()  {
      int array[] = {3, 5, 1, 4, 6, 2};
      int done = 0;
      int i;
while (done == 0)  {
   done = 1;
   for (i=0; i <=4; i++)  {
      if (array[i] < array[i+1]) {
         swap(&array[i], &array[i + 1]) ;
         done = 0;
      }
   }
   for (i=5; i >=1; i--)  {
        if (array[i] > array[i-1]) {
             swap(&array[i], &array[i-1]);
             done = 0;
        }
   }
}
```

```
    printf{"%d", array[3]);
}
```

The output of the program is ________. **[2017]**

**19.** Consider the following C Program.

```
#include<stdio.h>
#include<string,h>
int main ()  {
             char* c = "GATECSIT2017";
             char* p = c;
             printf{"%d",
             (int) strlen(c+2[p]-6[p]-1)) ;
             return 0;
}
```

The output of the program is __________. **[2017]**

**20.** Consider the following C program.

```
#include<stdio.h>
struct Ournode {
    char x, y, z;
} ;
Int main () {
    struct Ournode p = {'1', '0', 'a'+2};
    struct Ournode *q = &p;
    printf ("%c, %c", *( (char*)q+1),
                        * ( (char*)q+2) );
    return 0;
}
```

The output of this program is: **[2018]**

(A) `0, c`
(B) `0, a+2`
(C) `'0', 'a+2'`
(D) `'0', 'c'`

**21.** Consider the following C program:

```
#include<stdio.h>
void fun1 (char *s1, char * s2) {
char *tmp;
tmp = s1;
s1 = s2
s2 = tmp;
}
void fun2 (char **s1, char **s2) {
char *tmp;
tmp = *s1;
*s1 = *s2;
*s2 = tmp;
}
int main () {
char *str1 = "Hi", *str2 = "Bye";
fun1 (str1, str2);
printf ("%s %s ", str1, str2);
fun2 (&str1, &str2);
printf ("%s %s", str1, str2);
return 0;
}
```

The output of the program above is: **[2018]**

(A) Hi Bye Bye Hi (B) Hi Bye Hi Bye
(C) Bye Hi Hi Bye (D) Bye Hi Bye Hi

## Answer Keys

## Exercises

### Practice Problems 1

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1.** B | **2.** C | **3.** A | **4.** A | **5.** D | **6.** C | **7.** A | **8.** C | **9.** D | **10.** C |
| **11.** B | **12.** D | **13.** B | **14.** A | **15.** D | | | | | |

### Practice Problems 2

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1.** B | **2.** B | **3.** A | **4.** D | **5.** B | **6.** D | **7.** D | **8.** C | **9.** B | **10.** C |
| **11.** A | | | | | | | | | |

### Previous Years' Questions

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1.** D | **2.** A | **3.** –5 | **4.** A | **5.** D | **6.** C | **7.** 140 | **8.** D | **9.** 2016 | **10.** D |
| **11.** 30 | **12.** 3 | **13.** D | **14.** 3 | **15.** C | **16.** A | **17.** C | **18.** 3 | **19.** 2 | **20.** A |
| **21.** A | | | | | | | | | |