ARRAYS

- An Array is defined as, an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations.
- The data items of an array are of **same type** and each data items can be accessed using the same **name** but different **index** value.
- An array is a set of pairs, <*index, value* >, such that each index has a value associated with it. It can be called as ***corresponding*** or a ***mapping***

Ex: <index, value>

< 0 , 25 > list[0]=25
< 1 , 15 > list[1]=15
< 2 , 20 > list[2]=20
< 3 , 17 > list[3]=17
< 4 , 35 > list[4]=35

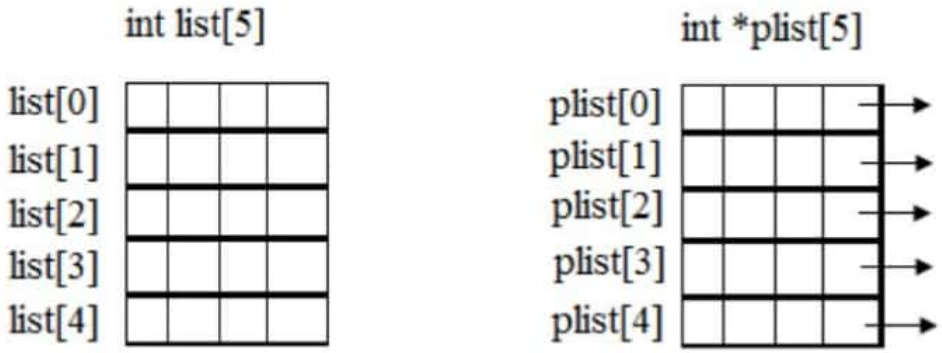
Here, ***list*** is the name of array. By using, list [0] to list [4] the data items in list can be accessed.

Array in C

Declaration: A one dimensional array in C is **declared** by adding brackets to the name of a variable.

Ex: **int list[5], *plist[5];**

- The array **list[5],** defines 5 integers and in C array start at index 0, so list[0], list[1], list[2], list[3], list[4] are the names of five array elements which contains an integer value.
- The array ***plist[5],** defines an array of 5 pointers to integers. Where, plist[0], plist[1], plist[2], plist[3], plist[4] are the five array elements which contains a pointer to an integer.



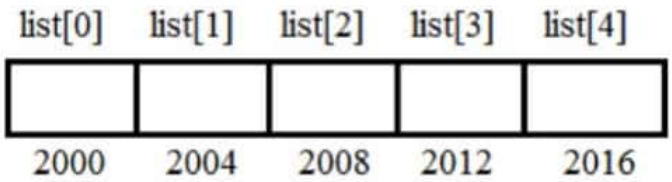
Implementation:

- When the complier encounters an array declaration, **list[5],** it allocates five consecutive memory locations. Each memory is enough large to hold a single integer.
- The address of first element of an array is called **Base Address.** Ex: For **list[5]** the address of **list[0]** is called the base address.
- If the memory address of **list[i]** need to compute by the compiler, then the size of the **int** would get by **sizeof (int)**, then memory address of list[i] is as follows:

$$\textbf{list[i]} = \alpha + \textbf{i} * \textbf{sizeof (int)}$$

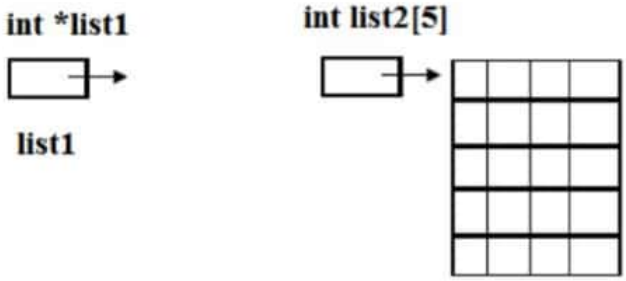
Where, α is base address.

list[3] = α+ 3 * sizeof(int)
= 2000 + 3 * 4
list[3] = 2012



Difference between **int *list1; & int list2[5];**

The variables **list1** and **list2** are both pointers to an **int,** but in **list2[5]** five memory locations are reserved for holding integers. **list2** is a pointer to **list2[0]** and **list2+i** is a pointer to **list2[i]**.



Note: In **C** the offset ***i*** do not multiply with the size of the type to get to the appropriate element of the array. Hence **(list2+i)** is equal **&list2[i]** and ***(list2+i)** is equal to **list2[i].**

How C treats an array when it is parameter to a function?

- All parameters of a C functions must be declared within the function. As various parameters are passed to functions, the name of an array can be passed as parameter.
- The range of a one-dimensional array is defined only in the main function since new storage for an array is not allocated within a function.
- If the size of a one dimensional array is needed, it must be passed into function as a argument or accessed as a global variable.

Example: Array Program

```
#define MAX_SIZE  100
float sum(float [], int);
float input[MAX_SIZE],  answer;
void main(void)
{
        int i;
        for( i=0; i<MAX_SIZE;  i++)
        input[i]= i;
        answer = sum(input,  MAX_SIZE);
        printf("\n The sum is: %f \n",answer);
}

float sum(float list[], int n)
{
        int i;
        float tempsum = 0;
        for(i=0;i<n; i++)
                tempsum = tempsum + list[i];
        return tempsum;
}
```

When **sum** is invoked, **input=&input[0]** is copied into a temporary location and associated with the formal parameter ***list***

A function that prints out both the address of the *i*th element of the array and the value found at that address can written as shown in below program.

```
void print1 (int *ptr, int rows)
{
        int i;
        printf(" Address contents \n");
        for(i=0;i<rows; i++)
                printf("% 8u %5d \n", ptr+i, *(prt+i));
        printf("\n");
}
```

Output:

Address	**Content**
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4

STRUCTURES

In C, a way to group data that permits the data to vary in type. This mechanism is called the **structure**, for short **struct**.

A structure (a record) is a collection of data items, where each item is identified as to its type and name.

Syntax: struct

```
{       data_type member 1;
        data_type member 2;
        .....
        .....
        data_type member n;
} variable_name;
```

Ex:

```
struct {
        char name[10];
        int age;
        float salary;
} Person;
```

The above example creates a **structure** and variable name is **Person** and that has three fields:

name = a name that is a character array
age = an integer value representing the age of the person
salary = a float value representing the salary of the individual

Assign values to fields

To assign values to the fields, use . (dot) as the structure member operator. This operator is used to select a particular member of the structure

Ex:

```
strcpy(Person.name,"james");
Person.age = 10;
Person.salary = 35000;
```

Type-Defined Structure

The structure definition associated with keyword **typedef** is called Type-Defined Structure.

Syntax 1: typedef struct

```
{
        data_type member 1;
        data_type member 2;
        .....
        .....
        data_type member n;
}Type_name;
```

Where,

- **typedef** is the keyword used at the beginning of the definition and by using typedef user defined data type can be obtained.
- **struct** is the keyword which tells structure is defined to the complier
- The members are declare with their data_type
- **Type_name** is not a variable, it is user defined data_type.

Syntax 2: struct struct_name

```
{
        data_type member 1;
        data_type member 2;
        .....
        .....
        data_type member n;
};
typedef struct struct_name Type_name;
```

Ex:

```
typedef struct{
            char name[10];
            int age;
            float salary;
        }humanBeing;
```

In above example, **humanBeing** is the name of the type and it is a user defined data type.

Declarations of structure variables:

```
humanBeing person1, person2;
```

This statement declares the variable **person1** and **person2** are of type **humanBeing.**

Structure Operation

The various operations can be performed on structures and structure members.

1. Structure Equality Check:

Here, the equality or inequality check of two structure variable of same type or dissimilar type is not allowed

```
typedef struct{
            char name[10];
            int age;
            float salary;
        }humanBeing;
humanBeing person1, person2;
```

if (person1 = = person2) is invalid.

The **valid function** is shown below

```
#define FALSE  0
#define TRUE  1
if (humansEqual(person1,person2))
        printf("The  two human  beings are the same\n");
else
        printf("The  two human  beings are not the same\n");
```

```
int humansEqual(humanBeing  person1, humanBeing  person2)
    { /* return TRUE  if person1 and person2 are the same human  being otherwise
                return FALSE  */
        if (strcmp(person1.name, person2.name))
                return FALSE;
        if (person1.age != person2.age)
                return FALSE;
        if (person1.salary != person2.salary)
                return FALSE;
        return TRUE;
     }
```

Program: Function to check equality of structures

2. Assignment operation on Structure variables:

person1 = person2

The above statement means that the value of every field of the structure of person 2 is assigned as the value of the corresponding field of person 1, but this is invalid statement.

Valid Statements is given below:

```
strcpy(person1.name,  person2.name);
person1.age  = person2.age;
person1.salary = person2.salary;
```

Structure within a structure:

There is possibility to embed a structure within a structure. There are 2 ways to embed structure.

1. The structures are defined separately and a variable of structure type is declared inside the definition of another structure. The accessing of the variable of a structure type that are nested inside another structure in the same way as accessing other member of that structure

Example: The following example shows two structures, where both the structure are defined separately.

```
typedef struct {
                int month;
                int day;
                int year;
        }date;

typedef struct {
                char name[10];
                int age;
                float salary;
                date dob;
        } humanBeing;
humanBeing person1;
```

A person born on February 11, 1944, would have the values for the date struct set as:

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

2. The complete definition of a structure is placed inside the definition of another structure.

Example:

```
typedef struct {
                char name[10];
                int age;
                float salary;
                struct {
                        int month;
                        int day;
                        int year;
                    } date;
        } humanBeing;
```

SELF-REFERENTIAL STRUCTURES

A self-referential structure is one in which one or more of its components is a pointer to itself. Self-referential structures usually require dynamic storage management routines (malloc and free) to explicitly obtain and release memory.

Consider as an example:

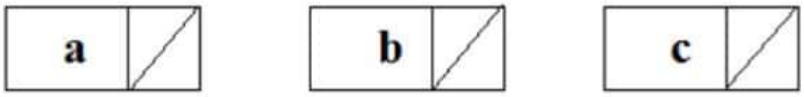
```
typedef struct {
        char data;
        struct list *link ;
} list;
```

Each instance of the structure **list** will have two components **data** and **link**.

- **Data:** is a single character,
- **Link:** link is a pointer to a list structure. The value of link is either the address in memory of an instance of list or the null pointer.

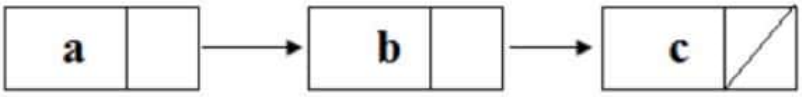
Consider these statements, which create three structures and assign values to their respective fields:

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.1ink = item3.link = NULL;
```



Structures item1, item2 and item3 each contain the data item **a, b,** and **c** respectively, and the null pointer. These structures can be attached together by replacing the **null link** field in item 2 with one that points to item 3 and by replacing the null link field in item 1 with one that points to item 2.

```
item1.link = &item2;
item2.1ink = &item3;
```



Unions:

A union is similar to a structure, it is collection of data similar data type or dissimilar.

Syntax:

```
union{
        data_type member 1;
        data_type member 2;
        ..........
        ..........
        data_type member n;
    }variable_name;
```

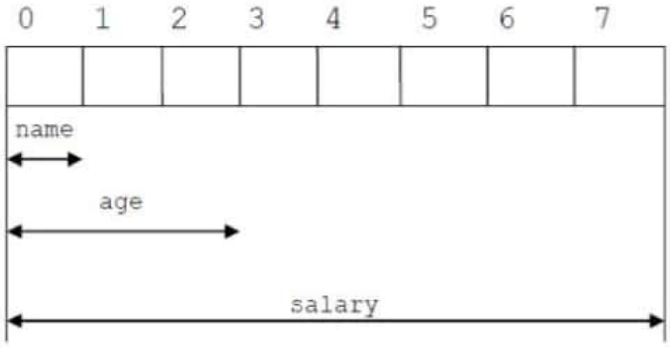
Example:

```
union{
        int children;
        int beard;
    } u;
```

Union Declaration:

A union declaration is similar to a structure, but the fields of a union must share their memory space. This means that only one field of the union is "active" at any given time.

```
union{
        char name;
        int age;
        float salary;
    }u;
```



The major difference between a union and a structure is that unlike structure members which are stored in separate memory locations, all the members of union must share the same memory space. This means that only one field of the union is "active" at any given time.

Example:

```
#include <stdio.h>
union job {
                char name[32];
                float salary;
                int worker_no;
              }u;

int main( ){
                printf("Enter name:\n");
                scanf("%s", &u.name);
                printf("Enter salary: \n");
                scanf("%f", &u.salary);
                printf("Displaying\n Name :%s\n",u.name);
                printf("Salary: %.1f",u.salary);
                return 0;
        }
```

Output:

Enter name: Albert
Enter salary: 45678.90

Displaying
Name: f%gupad (Garbage Value)
Salary: 45678.90