UNIT-V

☆ STRUCTURES IN C ☆

* Introduction:-

— Storage of complex data item is an important task a programmer has to perform. Complex data items involve set of records having variety of information. This call for a data type that can manage this variety at ease. C provides a constructed data type known as structures. We can combine different data types of our choice using structures. These structures help to organize complex data in a more meaningful way. Thus, structure is constructed or user defined data type which can group together different data types.

* Structure in C:-

— It is a user defined data type. We have seen basic data types like int, float and char.

— We can store only one type of value in these variables. But in case we want to store a record containing name, age and income of person, we need to store it in three different variables viz. char, int and float. Also, if we have to store a list of persons having name, age and income then we needed three arrays of same types. It is going to be very difficult to manage the different arrays.

– As an example, let us see a program which can store a list of persons have name, age, and income and sort it age-wise. We need to declare 3 arrays char name[50][20], int age[50], float income[50] (the size is assumed to be 50.)

★ Program:- List of employees using arrays.

```
#include <stdio.h>
#include<conio.h>
void main()
 {
   char name[50][20], temp1 [20];
   int age[50], temp2;
   float income [50], temp3;
   int i, j, n;
   printf("\n Enter how many persons:\n");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
     printf("\n Enter name: \n");
     fflush(stdin);
     gets(name[i]);
     printf("\n Enter age: \n");
     scanf("%d", &age[i]);
     printf("\n Enter income: \n");
     scanf("%f", &income[i]);
   }
   for(i=0; i<n-1; i++)
   {
     for(j=i+1; j<n; j++)
      {
```

```
if(age[i] > age[j])
 {
  strcpy (temp1, name[i]);
  strcpy (name[i], name[j]);
  strcpy (name[j], temp1);
  temp2 = age[i];
  age[i] = age[j];
  age[j] = temp2;
  temp3 = income[i];
  income[i] = income [j];
  income [j] = i~~ncome~~[temp3;
 }
 }
}
printf ("The list is:\n");
 for (i=0; i<n; i++)
  {
   printf("%s \t %d \t %f \n", name[i], age[i], income[i]);
  }
 getch();
}
```

— Note that in above program while sorting the records in the list, we had to handle each array seperately as these records are stored using 3 different arrays The program also becomes lengthy. Therefore we are use structure type variables to store the records, which allows us to store different data types in one variable making manipulation of records easier.

* Defining Structure :-

- There is a difference between structure declaration and definition.
- The declaration tell ~~the~~ compiler about prototype of structure.
- Whereas definition creates the structure variable. The definition allocates space in memory for structure variable.

Structure is declared as,

```
struct <name>
{
    data-type member1;
    data-type member2;
          :
    data-type membern;
};
```

- Where member1, member2, ... member n etc. can be int, float, char, array, struct itself. Also note that the declaration ends with semicolon. The structure declared is called structure template.

for example,

```
struct person
{
    char name [20];
    int age;
    float income;
};
```

- The fields name, age and income are called structure members. Thus, we have declared a Variable type or template called as struct person which is capable of storing name, age and income of one person. To store data items, variables of type struct person are to be defined.

for example, struct person p, q, r;

This will reserve space for storing 3 records in p, q, r as follows:

	name	age	income
p			
q			
r			

- The structure declaration and variable definition can be done together as follows:

```
struct person
{
    char name[20];
    int age;
    float income;
}p, q, r;
```

<u>Storing Data in Structure Variables</u>

- once we have declared the structure type variable, We can store the value in these variables. We have to store the values in the individual fields. This is done using dot(.) operator.

<u>for example</u>,

```
strcpy (p.name, "abc");
p.age = 30;
p.income = 30000.00
```

Consider the following program which store a records in structure type variables and copies into another variable of same type.

Program :- To store and display data in structure variable.

```
#include<stdio.h>
#include<conio.h>
 Void main()
 {
     struct person
      {
          char name[20];
          int age;
          float income;
      };
      struct person p,q;
       strcpy(p.name,"abc");
       p.age=25;
       p.income= 5000.00
        q=p;
       printf("%s\t %d \t %f \n",q.name, q.age, q.income);
    }
```

Output of above program will be

```
abc  25  5000.00
```

~~THEIS, eoe~~

* ARRAY of Structures :-

An array of records is used to store number of records, for example, a list of persons having name, age and income. In the previous section, we had defined a structure of the same. Now, we can have a variable declaration as struct person p[100]. It will reserve 100 locations as shown below to store list of 100 persons.

	name	age	income
p[0]			
p[1]			
p[2]			
⋮			
p[99]			

To access or store data into these locations, we can use dot operator.
for example, p[0].name is the name field of first record, p[0].age is the age field of first record, etc.
To read data into this array, we can use for loop. Similarly, to process data sequentially and display, a for loop can be used. Any record or field can also be randomly accessed.

* Program:- To store a list of persons and display the names of persons whose age is above 40.

```
void main()
 {
    struct person // structure declaration.
    {
      char name[20];
         int age;
    };

     int i, n;
     struct person p[100];   // Array of structure.
     printf("Enter number of persons \n");
     scanf ("%d", &n);
     for (i=0; i<n; i++)        // Read n Records
     {
       printf("Enter name: \n");
       gets(p[i].name);
       printf ("Enter age: \n");
       scanf("%d", &p[i].age);
     }
     printf("persons above 40 years are: \n");
     for (i=0; i<n; i++)    // Display records
     {
       if(p[i].age>40)  // if age>40
       printf("%s \t %d \n, p[i].name,
                                 p[i].age);
     }
```

* Program:- To prepare list of persons having name, age and salary and sort the list age wise.

soln ⟹

```
 Void main()
   {
```

```
struct person
{
    char name[20];          // structure definition
    int age;
    float sal;
};
   int i,j,n;
struct person p[100], temp;   // temp and array of
                              // structure.
structure
printf("Enter how many persons\n");
scanf("%d", &n);
for(i=0; i<n; i++)            // Read n records.
{
  printf("Enter name: \n");
  gets(p[i].name);
  printf("Enter age:\n");
   scanf("%d", &p[i].age);
   printf("Enter salary \n");
   scanf("%f", &p[i].sal~~ary~~ \n");
}
 for(i=0; i<n; i++)           //sort the records age wise
 {
      for(~~j=0;~~ j=i+1; j<n; j++)
      {
           if(p[i].age > p[j].age)
           {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
             }
         }
     }
   printf("sorted listed is \n");        //Display records
   for (i=0; i<n; i++)
   {
     printf("%s \t %d \t %f \n", p[i].name,
                                 p[i].age,
                                 p[i].sal);
   }
```

Program:-

* To read a list of students having roll number, name and marks in 3 subjects. Find total and percentage. Sort the list in descending order of percentage.

```
void main()
{
    struct student            //define structure.
    {
      int rollno;
      char name[20];
      int m1, m2, m3, total;
      float per;
    };

    struct student s[100], temp;  //temp and array
                                  // of structure.
     int i, j, n;
     printf("\n Enter number of students: \n");
     scanf("%d", &n);
     for(i=0; i<n; i++)
      {
         printf("Enter rollno: \n")
   scanf(~~printf(~~ "%d %d %d", &s[i].m1, &s[i].m2,
         &s[i].m3);

    s[i].total = s[i].m1 + s[i].m2 + s[i].m3
    s[i].per = s[i].total / 3.0;
    }
    for (i=0; i<n-1; i++)
    {
     for (j=i+1; j<n; j++)
      {
         if(s[i].per < s[j].per)
          {
              temp = s[i];
              s[i] = s[j];
              s[j] = temp;
          }
      }
 }
```

```
printf("sorted list is:\n");
for(i=0;i<n;i++)
{
    printf("%d \t %s \t %d \t %d \t %f \n",
            s[i].rollno, s[i].name, s[i].m1, s[i].m2,
            s[i].m3, s[i].total, s[i].per);
}
```

* Initializing Structure Variables:-

Just like initialization of other variables can be done at the time of ~~deleration~~ declaration [for example, int a = 10], structure variables can also be initialized

For example, struct person p={"abc", 20, 2000};
or struct person p[]={"abc", 20, 1000, "pqr", 30, 3000, "xyz", 40, 4000};

* Rules for Initializing Structure Variables:-

1. structure members cannot be initialized inside structure declaration or template.
2. The structure can be partially initialized like struct person p={"abc", 20};
3. The order of the values inside braces should be same as order of definition.
4. Default initial values will be 0 for int and float members and '\0' for char type member.

* program:- To store a list of items having items number, item name, rate, search an item in the list if item number is entered.

```
void main()
{
    struct item
    {
        int item_no;
    char name[20];
    float rate;
};

struct item lst[] = {10,"Rin",15.50,11,"Lux",
                        10.50,12, "surf", 50.60};

int s,i, flag=1;
printf("Enter item number: \n");
scanf("%d", &s);
for (i=0; i<n; i++)
{
    if(lst[i].item_no==s)
    {
        flag=0;
        printf("%s \t %f \n", lst[i].name,
           lst[i].rate);
        break;
    }
}
if(flag==1)
printf("Not available");
}
```

———— x o x ————

* STRUCTURE AND POINTERS:-

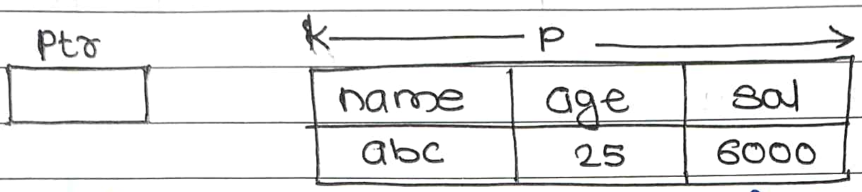
- Just like we can have pointer variable of int, float or character type, we can also have pointer variable of structure type. For example, if we have structure declared as,

```
struct person
{
    char name[20];
    int age;
    float sal;
}
```

Now, let us have two variables as:

struct person p = {"abc", 25, 6000};
struct person *ptr;

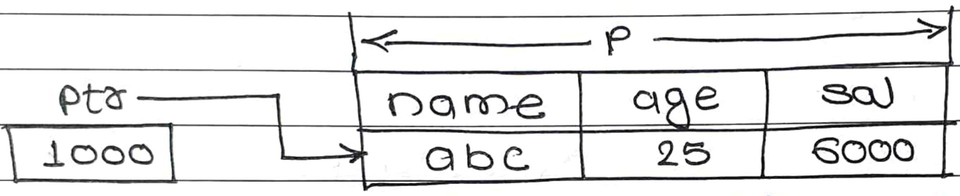
two locations in the memory will be reserved as:



p is simple structure variable and ptr is structure type pointer variable.
Let us store address of p in ptr.

ptr = &p;

This will store address of p say 1000 in ptr.



To access data stored in p, dot operator can be used.

p.name Will give "abc"
p.age Will give 25
p. sal Will give 6000.

Since ptr is a pointer to p, we can access the data through ptr also. An arrow operator (->) is used to access the fields through the pointer type structure variable.

ptr -> name will be same as p.name i.e. "abc".
ptr -> age will be same as p.age i.e. 25.
ptr -> sal will be same as p.sal i.e. 6000.

To access entire structure variable through ptr we can use * operator, i.e. *ptr is same as p.

We can also access the fields through ~~point~~ ptr using . operator as (*ptr).name will be same as p.name i.e. "abc".

(*ptr)•age will be same as ~~page~~ p.age i.e. 25.
(*ptr)• sal will be same as p. sal i.e. 6000.

Now let us write a simple program to illustrate this.

<u>Program:-</u> Use of structure type pointer.

```
void main()
 {
     struct person
     {
        char name[20];
        int age;
        float sal;
     }

   struct person p={"abc", 25, 6000}
   struct person *ptr, temp;
    ptr = &p;
   printf(%s \t %d \t %f \n", ptr->name, Ptr->age,
              ptr -> sal);
   temp = * ptr;
   printf(%s \t %d \t %f \n", temp.name, temp.age,
              temp.sal);

 }
```

<u>Explanation:-</u>

— ptr is pointer to p. hence, output of first printf will be, abc 25 6000

— temp= *ptr will store a value (entire record in p) pointed by ptr in temp. Hence, the output of second printf will also be same.

* **Array of structure and pointer**

A pointer ~~structure~~ variable can be used to store address of an array of records. We can extends this idea for array of structure also. If we have single pointer variable.

struct person *p;

This declaration will reserve only two byte of memory for p at the time of complication. This variable can be allocated memory block, to store an array of records using malloc as:

p=(struct person*) malloc(n* sizeof(struct person));

The function malloc will allocate memory block for storing n records of the size of struct person, i.e., 26*n bytes and the address of first record will be address of 3rd record an so on. In general, p+i will be address of i+1 record. We can use -> operator to access the locations pointed by p, p+1, p+2 etc.

i.e. (p+i) -> name will access name in i+1th record.
(p+i) -> age will access age in i+1th record.
(p+i) -> sal will access sal in i+1th record.

— Let us write a program to illustrate these concepts.

* Program :- To store a list of n persons and sort it on the basis of age.

```
void main()
{
    struct person
    {
        char name[20];
        int age;
        float sal;
    };

    struct person *p.temp;
    int i, j, n;
    clrscr();
    printf("Enter number of persons \n");
    scanf("%d",&n);
    p=(struct person*) malloc (n* sizeof(struct
                                     person));
    for(i=0; i<n; i++)
    {
        printf("Enter name:\n");
        gets((p+i) -> name);
        printf("Enter age:\n");
        scanf("%d", &(p+i)->age);
        printf("Enter salary: \n");
        scanf("%d", &(p+i) -> sal);
    }
    for(i=0; i<n-1; i++)
    {
        for(j=~~0~~ i+1; j<n; i++)
        {
            if((p+i)->age > (p+j) ->age)
            {
                temp= *(p+i);
                *(p+i) = *(p+j);
                *(p+j) = temp;
            }
        }
    }
    printf("sorted list is:\n");
    for(i=0; i<n; i++)
    {
      printf("%s\t%d \t %f \n", (p+i)->name, (p+i)->age,
                                  (p+i)->sal);
    }
  getch();
}
```

Explanation:-

— p=(struct person *) malloc(n* sizeof(struct person)); will allocate memory for storing n records, where n will be known at run time. Thus, depending on value of n, memory is allocated to the pointer variable ptr. If we have 10 records to store, memory for 10 records (260 bytes) will be allocated.

— The first for loop runs for n times accepting every time name, age and salary and it is stored in locations whose addresses are p, p+1, p+2, - - -,.

— The second nested for will sort the records age wise using selection sort. Here *(p+i) refers to the record pointed by p+i which is equivalent to p[i]. *(p+j) refers to the records pointed by p+j which is equivalent to p[j].

— The third loop for loop displays all the sorted records.

* Nested Structure

- We can have another structure variable as a part of structure. Consider that we want to store name and date of birth of a person. Date of birth ~~can of a~~ can contain 3 fields day, month, and year. Hence, we declare it as another structure as shown on next page.

```
struct dob
{
    int dd,mm,yy;
 }
  struct person
   {
       char name[20];
       struct dob d;
   };
```

Thus, we have name and d as two fields in struct person where d is a structure type variable nested inside struct person. If we define.

struct person p;

p will be allocated memory as:

p

Name	d		
	dd	mm	yy

If we want to acess name we write p.name. If date of birth is to be accessed, we can use p.d.dd, p.d.mm, and p.d.yy.

Let us write a simple program which stores name and date of birth of n persons and display list of persons whose date of bir th is in the month of June.

* Program:- To store name and date of birth and display list of persons whose birthday is in June.

```
void main()
 {
      struct dob
       {
         int dd, mm, yy;
       };
struct person
{
   char name[20];
   struct dob d;
};

Struct person p[100];
int i, j, n;
printf("Enter number of persons \n");
scanf("%d", &n);
for(i=0; i<n; i++)
 {
   printf("Enter name: \n");
   gets(p[i].name);
   printf("Enter date of birth in dd:mm:yy format:\n");
   scanf("%d : %d : %d", &p[i].d.dd, &p[i].d.mm,
                          &p[i].d.yy);
 }

printf("List of persons whose date of birth in
          June is: \n");
for (i=0; i<n; i++)
 {
   if(p[i].d.mm == 6)
   printf("%s \t %d:%d:%d \n", p[i].name, p[i].d.dd,
            p[i].d.mm, p[i].d.yy);
 }

getch();
}
```

* Passing structures to Functions:-

– A function can be passed parameter of type int, float, char etc. We can also pass structure type data to a function and function can return a structure.
We can pass
1. structure members indivisually,
2. Structure variables by value.
3. Structure variables by address.
4. An array of structure.

1. Passing Structure Members Indivisually:

* program – passing structure member to function.

```
Void cal_bonus(float);
Void main()
{
    struct person;
    {
      char name[20];
      int age;
      float sal;
    };

   struct person p={"aaa",20,2000}
   cal_bonus(p.sal);
}

Void cal_bonus(float salary)
 {
    float bonus;
    bonus = salary * 0.5;
    printf("Bonus is "%f", bonus);

 }
```

In above program, we have passed salary

of the person and calculated bonus in the function. The parameter is a float, hence the function argument is also float.

2. **Passing structure by value:-**

consider following program.

```
Void disp(struct person);
Void main()
{
    struct person p={"abc", 20, 2000};
     disp(p);
 }

 Void disp(struct person p)
  {
    printf("%s \t %d \t %f \n", p.name, p.age,
                         p.sal);
  }
```

Output:- abc 20 2000

Here, we are passing the variable p to the function and all the fields will be available to the function which can be displayed.

Typedef:

Whenever one writes function that requires passing of structure, we have to repeatedly write the word struct and name of structure. In order to avoid this, we can create a short-cut or alias for this using keyword typedef as:

```
typedef struct person
 {
     char name[20];
     int age;
     float sal;
 } PER;
```

}

This means PER is an alias (another name) for struct person.

5. <u>Passing structure by address</u>:-

Read the given program and find what will be its output.

<u>Program</u>:- Passing structure by value.

```
typedef struct person
 {
        char names[20];
        int age;
        float sal;
  }PER;

   void disp(PER);
   void modify(PER);
   void main()
  {
     PER p={"abc", 20, 20000};
     disp(p);
     modify(p);
     disp(p);
  }

 Void disp(PER p)
 {
  printf("%s \t %d \t %f \n", p.name, p.age, p.sal);
 }

 Void modify(PER p)
  {
      p.age = p.age+1;
      p.sal = 1.1 * p.sal;
  }
```

<u>Output</u>:-

```
            abc   20   20000
            abc   20   20000
```

Explanation:

- There are two functions disp and modify to which structure variable is passed by value.
- When disp is called for the first time, main will pass the contents of p in main to p in disp and the records get displayed.
- When modify is called, main will pass the contents of p in main to p in modify. The function modify will change age field of p to 21 and sal field to 22000; but these changes will be made in local variable p of modify and not of main. Hence, when control is back in main, its p is unchanged.
- When disp is called second time, main will pass the contents of structure variable passed to it? The answer is, we have to pass it by address as shown in following program.

Program:- passing structure by address.

```
typedef struct person
 {
     char name2[0];
     int age;
     float sal;
 }PER;

 void disp (PER);
 void modify (PER*);
 void main ()
  {
```

```
  PER p = {"abc", 20, 20000};
   disp(p);
   modify (&p);
   disp (p);
}

void disp(PER p)
{
   printf("%s \t %d \t %f \n", p.name, P.age, p.sal);
}

void modify (PER *ptr)
{
     ptr -> age -> ptr -> age+1;
     ptr -> Sal = 1.1 * ptr -> sal;
}
```

Output:

```
          abc    20    20000
          abc    21    22000
```

4. Passing Structure Type Array to Function

As we know if we have to pass an array to a function, we have to pass address of first element in the array to the function. Function argument should be pointer type variable. In this case, it should be pointer to structure. Let us consider a program to read a list of persons and sort it on the basis of age. We will write seperate function for reading, sorting and displaying the records.

☆ Program to sort list of persons using seperate functions for ~~red~~ read, sort & display.

```
typedef struct person
{
    char name[20];
    int age;
    float sal;
}PER;

void read_recs(PER*);
void disp_recs(PER*);
void sort-recs(PER*);
int n;

void main()
{
    PERSON p[100];
    int i;
    printf("How many persons? \n");
    scanf("%d", &n);
    read-recs(p);
    ~~read~~ sort_recs(p);
    disp_recs(p);
}

void read_recs(PER *ptr)
{
    int i;
    for(i=0; i<n; i++)
    {
        printf("Enter name \n");
        gets(ptr+i) -> name);
        printf("Enter age \n");
        scanf("%d", &(ptr+i)->age);
        printf("Enter salary \n");
        scanf("%f", &(ptr+i)-> sal);
    }
}
```

```
void sort_recs(PER *ptr)
 {
   int i, j;
   PER temp;
  for(i=0; i<n; i++)
  {
  for(j=i+1; j<n; j++)
   {
    if((ptr+i)->age > (ptr+j)->age)
    {

       temp *(ptr+i);
       *(ptr+i) = *(ptr+j);
      *(ptr+j) = temp;
    }
   }
  }
}


void disp_recs (PER *ptr)
 {
    int i, j;
   for(i=0; i<n; i++)
   {
     printf("%s \t %d \t %f \n", (ptr+i)->name,
               (ptr+i)-> age, (ptr+i)-> sal);
   }
 }
```

Explanation :-

→ The main function has array P[100]. Its address p is passed to each function.

→ The function accepts the address in pointer variable ptr. Thus, ptr has address of first record, ptr+1 is address of second record. In general, i+1 th record can be accessed through ptr+i.

* Returning a structure from Function :-

Indivisual structure members or entire ~~function~~ structure can be returned back via a return statement at the access point in the calling function. If indivisual structure member is returned, return type of the function will be same as type of structure member. If structure is returned, the return type of the function is same same as that of structure type.
Consider following example of addition of two rational numbers represented as struct type.

Program :- Addition of two rational numbers

```
typedef struct rational
{
      int num, den;
} RAT;

RAT add (RAT, RAT)

Void main ( )
 {
   RAT r1 = {2, 3}, r2 = {4,5}, r3;
   r3 = add (r1 , r2);
   printf ("%d | %d", r3.num, r3.den);
 }
  RAT add( RAT r1 | RAT r2)
   {
      RAT r3;
      r3.num = r1.num r2.den + r1.den r2.num;
      r3.den = r1.den r2.den;
      return (r3);
   }
```

Explanation :- The num and den are numerator and denominator of rational numbers.
- The function add accepts two ~~structure~~ type (RAT) variable r1, r2; calculates numerator and denominator of resultant rational number r3 and returns it.

* Pointers in C:-

→ Pointers is a very powerful tool which allows programmer lot of flexibility and help in improving efficiency of the program. This concept is important because almost all data structure are based on pointers.

* Basic Concepts of pointers

– First thing that should be clear to you is, We are writting a program which is going to be executed by a machine which has memory and each location in memory has an address. When we declare a variable say int a; a location in the memory is reserved to store an integer number.

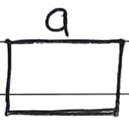


fig:- Memory allocation for a.

We have the location name as 'a'. The computer which is a digital device gives number to the location which is called as address of that location. It is just like we give some nice name to our home and Muncipal Corporation identifies it with House Number or Survey No. etc. Thus, the memory variable will have associated with it an unique address as shown in following figure.

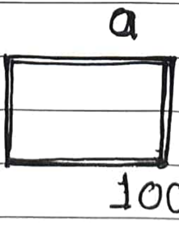


fig:- Address of a.

What if I want to know the address of that variable? It is possible to access the address of a variable using & operator. The operator & (ampersand) gives address of the corresponding variable. Thus, in above case, &a will be 10000. The address will be solely decided by the computer and not by the programmer. Let us write a program to understand this concept.

* Program:- Program to illustrate address of (&) operator.

```
void main()
{
    int i=4, j=8;
    printf("value of i = %d\n", i);
    printf("Address of i = %u\n", &i);
    printf("value of j = %d\n", j);
    printf("Address of j = %u\n", &j);
}
```

Explanation:-

- In above program we have two variables declared as i and j. The situation in the memory will be shown in following figure.

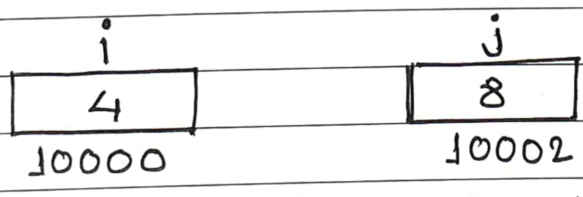


fig:- Memory allocation and address of i and j.

Here we are assuming that the address of i and j are 10000 and 10002 respectively. The output of the program will be as follows.

```
Value of i = 4
Address of i = 10000
value of j = 8
Address of j = 10002
```

∴ Note that while displaying address %u is used because the address will never be negative number. It is displayed as unsigned integer.

When C program is executed the RAM consist of operating system, the program and the data involved in the program in seperate areas. The data area is nothing but the space reserved for the variables in the program. The space in RAM is measured in bytes. Each cell in RAM is of 1 byte. The address is given to each byte in RAM.

Generally, following is space allocation for each variable type.

1 byte for char
2 bytes for int
4 bytes for float and longtime
8 bytes for double.

Hence, when we declare variables as below the memory allocation will be as shown in figure on next page.

```
int a, b;
float x, y;
char ch;
```

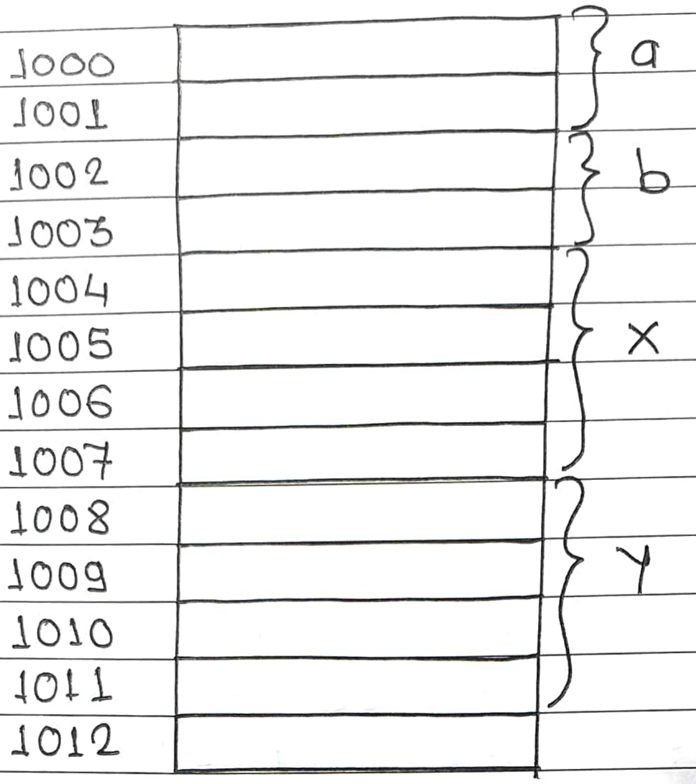


fig:- Memory allocation for variables.

Just like we have & operator giving address of a variable, there is another operator called operator which gives value stored at a particular address. It is called indirection operator. e.g. *(&a) will give value stored at address of a, which is nothing but value stored at a only.

Let us consider one more program to illustrate the concept.

★ program:- Use of & and * operator.

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

* How to use pointers?

There are a few important operations, which we will do with the help of pointers very frequently.
(a) We define a pointer variable,
(b) assign the address of variable to a pointer and
(c) finally access the value at the address available in the pointer variable.
This is done by using unary operator * that returns the value of the variable located at the address of specified by its operand.
The following example makes use of these operations

```
#include <stdio.h>
int main() {

int var=20;  /* Actual variable declaration */
int *ip;     /* pointer variable declaration */

 ip = &var  /* store address of var in pointer variable */

printf("Address of var variable: %x \n", &var);
 /* Address stored in pointer variable */
printf("Address stored in ip variable: %x \n", ip);
 /* access the value using the pointer */
printf("value of *ip variable: %d \n", *ip);

return 0;
}
```

When the above code is compiled and executed, it produces the following result-

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer-

Sr. No.	Concept & Description
1.	Pointer Arithmetic There are four arithmatic operators that can be used in pointers: ++, --, +, -
2.	Array of Pointers You can define arrays to hold a number of pointers.
3.	Pointer to Pointer C allows you to have pointer on a pointer and so on.
4.	Passing Pointers to Functions in C passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.

* Pointer Declaration and Initialization:-

There are three- basic types of variables in C viz. int, float, char. If a variable is of type int, it will store only integer type data. It is possible to store address of a variable in memory. C provides a pointer type variable which is capable of storing address of another variable is declared as,

```
int *p;
```

Thus, a pointer variable is declared like any other variable with * preceding the variable name.

It means we are declaring a variable p which is a pointer type variable and it is capable of storing address of any integer variable. We can use this variable to access the value stored in another location. Suppose we have 2 variables declared as follows:

```
int a=10;
int *p;
```

Two locations will be reserved in the memory as shown in figure.

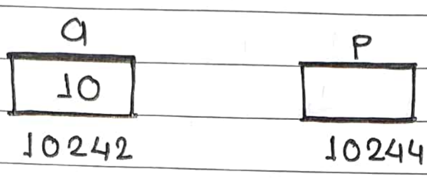
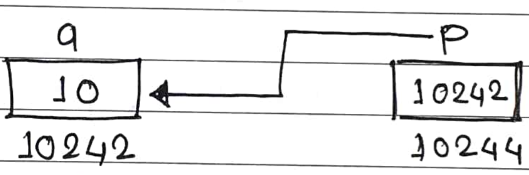


figure :- Memory allocation for a and p

Now, if I write a statement as

```
p=&a;
```

address of a (i.e., &a) will be assigned to p



It means p has the address of a i.e. p is pointing to a as shown and *p will gives value stored at a. Thus, if we store address of a variable in a pointer type variable we can access the value stored in that variable through the pointer type variable.

Let us consider a program:-

* program:- To illustrate pointer variable & operator.

```
Void main()
{
    int *p;
    int a=10;
    p=&a;
   printf("value of a =%d \n", a);
   printf(" Address of a = %u \n", &a);
   printf("value of p =%u \n", p);
   printf(" value of a= %d \n", *p);
}
```

Output of the program will be

```
Value of a = 10
Address of a = 10242
Value of p = 10242
Value of a = 10
```

* **Note:-** We can have float and char type pointers also. The float type pointer can store only address of float type variable. It cannot store address of int or char variable. Similarly, char type pointer can store only address of char type variable. Following program has all the three pointers.

* **Program:-** To illustrate pointer variable & operator.

```
Void main()
{
    int *pi, a=10;
    float *pf, x= 1.2345;
    char *pc, c='*';
    pi = &a;
    pf = &x;
    pc = &c;
    printf("%d \n", *pi);
    printf("%f \n", *pf);
    printf("%c \n", *pc);

}
```

Output:-

```
10
1.2345
*
```

* Program:- To illustrate operations with pointer variable.

```
void main()
{
   int a=10, b=20, *p, *q, c;
   p=&a;
   q=&b;
   c = *p + *q;
   printf("sum is %d \n", c);
}
```

Output:- 30.

* Program:- To illustrate operations with pointer variable.

```
void main()
{
   int a=10, *p;
   p=&a;
   *p=*p+100;
   printf("sum ~~of~~ is: %d", a);
}
```

output:- 110.

* Program:- To illustrate operations with pointer variable.

```
void main()
{
   int ~~a=20~~ a=10, b=20; c, *p;
 p ~~p~~=&c;
   *p= a+b;
   printf("sum is: %d \n", c);
}
```

output:- 30.

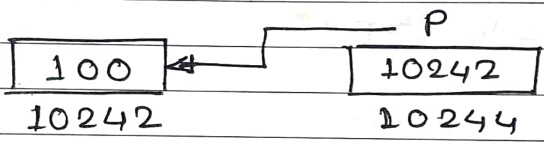
* **program:-** To illustrate operations with ~~integer~~ pointer variable.

```
Void main()
 {
   int *p;
   *p = 100;
   *p = *p * 10;
   printf(%d \n", *p);
 }
```

output:- 1000

Explanation-

– *p = 1000 will store the constant number 100 in a temporary location and p will point to this as shown in following figure.



– The main statement *p = *p * 10 will be store product of location pointed by p and 10 into a location pointed by p itself.

* **Pointer to a pointer:-**

* **program:-** To illustrate double pointer.

```
 void main()
 {
   int a=4, *b, **c;
   b = &a;
   c = &b;
  printf (" ~~%u \n", Address~~ %u \n" &a);
  printf (" % u \n", b);
  printf(" %u \n", c);
  printf(" %u <n
  printf(" % d \n", a);
 Printf ("%d ~~*b~~ \n", *b);
 printf (" %d ",\n, **c);
 }
```