Ch. 9. Structures

Arrays & Strings ⇒ Similar data (int, float, char)

Structures can hold ⇒ dissimilar data

Syntax for creating Structures

A C structure can be created as follows:

```
typedef struct employee {
    int code;
    float salary;
    char name[10];
};
```

⇒ This declares a new user defined data-type.

`};` → Semi-colon is important

We can use this user defined data type as follows

```
Struct employee e1;
Strcpy(e1.name, "Harry");
e1.code = 100;
e1.salary = 71.22;
```

⇒ Creating a structure variable

So a structure in C is a collection of variable of different types under a single name

Quick Quiz: Write a program to store the details of 2 employees from user defined data. Use the structure declared above.

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

```
struct employee {
    int code;
    float salary;
    char name[10];
};

int main() {
struct employee e1, e2;
printf("Enter the value for code
                of e1: ");
scanf("%d", &e1.code);
printf("Enter the value for salary of
                e1: ");
scanf("%f", &e1.salary);
printf("Enter the value for name of e1: ");
scanf("%s", e1.name);

printf("Enter the value for code of e2: ");
scanf("%d", &e2.code);
printf("Enter the value of salary of e2: ");
scanf("%f", &e2.salary);
printf("Enter the value for name of e2: ");
scanf("%s", e2.name);

return 0;

}
```

Output:

Enter the value for code of e1: 3
" salary : 21.2
" name : harry

Enter the value for code of e2: 4
" salary : 22.2
" name : Shubham

Q: Why use structures?
A We can create the data types for the employee structure seperately but when the number of properties in a structure increases, it becomes difficult for us to create data variables without structures. In a nutshell:

(a) structures keep the data organized
(b) structures make data management easy for the programmer

Array of Structures

Just like an array of integers, an array of floats & an array of characters, we can create an array of structures

Struct employee facebook[100]; An array of structures.

We can access the data using:
facebook[0].code = 100;
facebook[1].code = 101;
.... & so on

Initializing Structures

Structures can also be initialized as follows:

Struct employee harry = {100, 71.22, "Harry"};

Struct employee shubh = {0}; ⇒ All elements set to 0

Structures in Memory

Structures are stored in contiguous memory locations for the structure e1 of type struct employee, memory layout looks like this:

100	71.22	"Harry"
78810	78814	78818

Address →

In an array of structures, these employee instances are stored adjacent to each other

Pointer to Structures

A pointer to structure can be created as follows:

```
Struct employee *ptr;
ptr = &e1;
```

Now we can print structure elements using:

~~printf("%d", *(ptr).code);~~

printf("%d", (*ptr).code);

Arrow Operator

Instead of writing (*ptr).code, we can use arrow operator to access structure properties as follows.

(*ptr).code as ptr → code

Here: → is known as the arrow operator

Code:

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

struct employee{
    int code;
    float salary;
    char name[20];
};

int main(){
    struct employee e1;
    struct employee *ptr;

    ptr = &e1;
// (*ptr).code = 101;/*or you can also write etc */
    ptr -> code = 101;
    printf("%d", e1.code);
```

```
return 0;
}
```

Output
101

Passing Structure to a function

A structure can be passed to a function just like any other data type.

```
Void show(struct employee e);
```
function prototype

Typedef Keyword

We can use the typedef keyword to create an alias name for data types in C. typedef is more commonly used with structures.

```
struct complex {
    float real;  → struct complex c1, c2;
    float img;   for defining complex nos
};
```

```
typedef struct complex {
    float real;
    float img;
} complex ng;
```

⇒ Complex ng. C_1, C_2;
for defining complex ng.

A structure is a user defined data type that can be used to group elements of diff. types into a single type

For e.g. Engine: DDis. 190 Engine
Fuel type: Petrol
Fuel tank capacity: 37
Seating capacity: 5
City Mileage: 19.74 kmpl

For e.g.
```
struct {
    char *engine;
    char *fuel_type;
    int fuel_tank_capacity;
    int seating_cap;
    float city_mileage;
} car1, car2;
```

For e.g.2
```
struct {
    char *engine;
} car1, car2;
```
} This structure is in the global scope.

```
int main() {
car1.engine = "DDis 190 Engine";
car2.engine = "1.2L Kappa Dual VTVT";
printf("%s\n", car1.engine);
printf("%s", car2.engine);
return 0;
}
```

O/p: DDis 190 Engine
1.2L Kappa Dual VTVT

```
struct {
char *name;
int age;
int salary;
} emp1, emp2;

int manager ()
{
struct {
char *name;
int age;
int salary;
} manages;
...
}
```

Seperate Declaration

```
struct employee {
char *name;
int age;
int salary;
};

int manager()
{
struct employee manager;
...
}

int main()
{
struct employee emp1;
struct employee emp2;
...
}
```

Structure tag

Structure tag is used to identify a particular kind of a structure

Syntax: typedef existing_data_type new_data_type

typedef gives freedom to the user by allowing them to create their own types.

For e.g
```
#include <stdio.h>
typedef int INTEGER;
int main() {
INTEGER var = 100;
printf("%d", var); return 0; }
```

o/p = 100

Typedef in structures,

```
typedef struct car{
    char engine[50];
    char fuel_type[10];
    int fuel_tank_cap;
    int seating_cap;
    float city_mileage;
}car;

int main(){
    car c1;
}
```

car becomes new data type

INITIALIZING AND ACCESSING THE STRUCTURE MEMBERS

For e.g.

```
struct abc{
    int p;
    int q;
};

int main(){
    struct abc x={23,34};
}
```

We can access the members of structure using dot(.) operator.

DESIGNATED INITIALIZATION

Designated initialization allows structure members to be initialized in any order.

```
struct abc {        Don't forgot to used dot operator
    int x;
    int y;
    int z;
};

int main()
{
    struct abc a = {.y=20, .x=10, .z=30};
    printf("%d %d %d", a.x, a.y, a.z);
    return 0;
}
```

o/p = ~~10, 20~~ 10 20 30

ARRAY OF STRUCTURE

Instead of declaring multiple variables, we can also declare an array of structures in which each ~~array~~ element of the array will represent a structure variable.

```
struct car {
    int fuel_tank;
    int seating_cap;
    float city_mileage;
};

main() {
    struct car c[2];
}
```

POINTER TO STRUCTURE

```
struct abc {
    int x;
    int y;
};

int main() {
    struct abc a = {0, 1};
    struct abc *ptr = &a;

    printf("%d %d", ptr->x, ptr->y);
    return 0;
}
```

ptr is a pointer to some variable of type struct abc.

ptr->x is equivalent to (*ptr).x or (*&a).x or a.x

o/p = 0 1

STRUCTURE PADDING IN C

When an object of some structure type is declared then some contiguous block of memory would be allocated to structure members.

How memory is allocated to structure members?

For eg:

```
struct abc {
    char a;
    char b;
    char c;
} y, var;
```

Calculating the size of structures

Let size of characters be 1 byte & size of int be 4 bytes.

For e.g.

```
struct abcd
{
    char a;   // 1 byte
    char b;   // 1 byte
    int c;    // 4 bytes
} var;
```

Total : 6 bytes but the size of structures is not 6 bytes

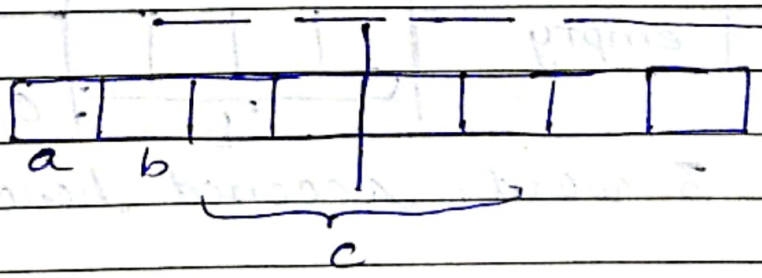
Structure Padding

Processor doesn't read one byte at a time from memory.
It reads 1 word at a time.

If we have a 32 bit processor then it means that it can access 4 bytes at a time which means word size is 4 bytes.

If we have a 64 bit processor then it means it can access 8 bytes at a time which means word size is 8 bytes

IN 32 BIT ARCHITECTURE

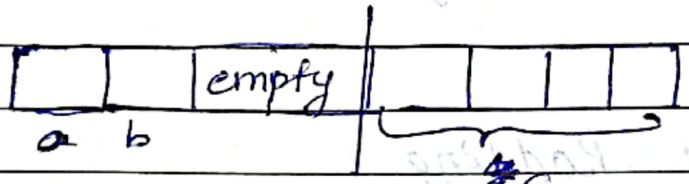


In 1 CPU cycle char a, char b & two bytes of int c can be accessed. There is not problem with char a & char b but....

Whenever we want the value stored in variable c, 2 cycles are req^d to access the contents of variable c. In 1st cycle, 1st two bytes can be accessed & in 2nd cycle, last 2 bytes. It's an unnecessary wastage of cycles.

We can save the no. of cycles using the concept called padding:

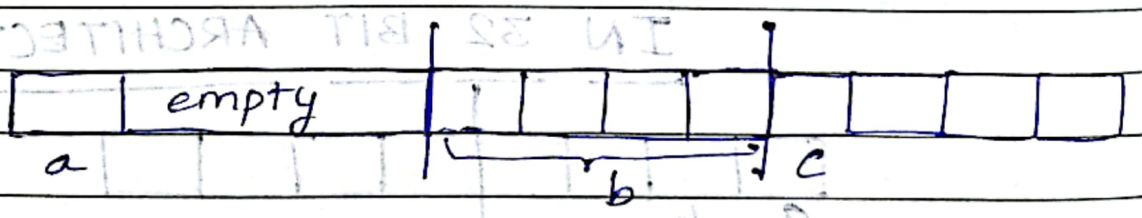
In 32 bit architecture



Total = 1 + 1 + 2 + 4 = 8

What happens if we change the order of members?

For e.g. struct abc {
char a;
int b;
char c;
};
Total = 12 =



3 words accessed hence 12 bytes

STRUCTURES PACKING

Because of structure padding, size of structure becomes more than the size of actual structure. Due to this some memory would get wasted.

We can avoid the wastage of memory by simply writing #pragma pack(1)

```
#pragma pack(1)
struct abc {
    char a;
    int b;
    char c;
} var;
```

Total: 6 bytes

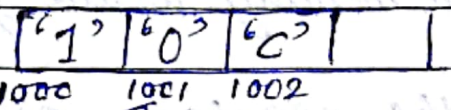
#pragma is a special purpose directive used to turn on or off certain features

Q Consider the foll. 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;
}
```

a) 0, c b) 0, a+2 c) '0', 'a+2' d) '0', 'c'



q [1000] q contains the address of the whole structure

q was pointer to the whole structure. now it is pointer to a character

Q The following C declaration

```
struct node
{
    int i;
    float j;
};
struct node *s[10];
```

define s to be:

a) An array, each element of which is a pointer to a structure of type node

b) A structure of 2 fields, each field being a pointer to an array of 10 elements

c) A structure of 3 fields: an integer, a float & an array of 10 elements

d) An array, each element of which is a structure of type node

Ans: As sq. brackets have more precedence than * it's an array.

INTRODUCTION TO UNIONS

Union is a user defined data type but unlike structures, union members share same memory location.

For e.g.

```
union abc {
    int a;
    char b;
} y;
```

a's address = 6295616
b's " = "

```
struct abcd
    int a;
    char b;
} y;
```

a's address = 6295616
b's " = 6295628

Memory Allocation

struct abc

union abc

In union, members will share same memory location. If we make changes in one member then it will be reflected to other members as well.

For e.g.-

```
union abc {
    int a;
    char b;
} var;

int main() {
    var.a = 65;
    printf("a = %d\n", var.a);
    ||  ("b = %d\n", var.b);

    return 0;
}
```

o/p = a = 65
b = A

∵ ASCII of A = 65

SIZE OF UNION

Size of the union is taken according to the size of the largest member of the union.

E.g.

```
union abcd {
    int a;
    char b;
    double c;
    float d;
};
```

size of int = 4 bytes
size of char = 1 byte
" " double = 8 bytes
" " float = 4 bytes

∴ Total = 8

ACCESSING MEMBERS USING POINTERS

We can access the members of union through pointers by using the arrow operator (->).

Date
Page

9

```
struct {
    short s[5];
    union {
        float y;
        long z;
    } u;
} t;
```

Assume that the objects of the type short, float & long occupy 2 bytes, 4 bytes & 8 bytes resp. The memory req. for variable t, ignoring alignment considerations, is

a) 22 bytes c) 18 bytes
b) 14 bytes d) 10 bytes

for short s[5] $\Rightarrow 5 \times 2 = 10$ bytes

for union: Memory allocated to union is equal to the memory needed for the largest member of it.

$\therefore$ size = 8 bytes

$\therefore$ Total = 10 + 8 = 18 bytes

```
struct store
{
    double price;      // 8 bytes
    char* title;       // 8 --11--
    char* author;      // 8
    int num_pages;     // 4
    int color;         // 4
    int size;          // 4
    char* design;      // 8
};
```

Total: 44 bytes

pragma pack(1):

```
struct store {
    double price;      // 8 bytes
union {
  struct {
    char* title;       // 8 bytes
    char * author;     // 8 bytes
    int num_pages;     // 4 bytes
     } book;

  struct {
    int color;         // 4 bytes
    int size;          // 4 bytes
    char * design;     // 8 bytes
     } shirt;
  } item;
};

int main() {
  struct store s;
s.item.book.title = "The Alchemist";
s.item.book.author = "Paulo Coelho";
s.item.book.author =
s.item.book.num_pages = 197;
printf("%s\n", s.item.book.title);
printf("%ld", sizeof(s));
return 0;
}
```

o/p
```
The Alchemist
28
```

```
typedef union
{ int a;
  char b;
  double c;
} data;

int main()
{
  data arr[10];
  arr[0].a = 10;
  arr[1].b = 'a';
  arr[2].c = 10.178;
  // and so on
  return 0;
}
```

Size = 8 bytes

Size = 80 bytes

```
typedef struct
{ int a;
  char b;
  double c;
} data;

int main()
{
  data arr[10];
  arr[0].a = 10;
  arr[1].b = 'a';
  arr[2].c = 10.178;
  return 0;
}
```

Size = 13 bytes

Size = 130 bytes

Practice-set 9

9) Create a 2-D vector using structures in C

```
#include <stdio.h>

struct vector{
    int x;
    int y;
};

int main(){
struct vector v1, v2;
 v1.x = 34;
 v1.y = 54;
printf("X dem is %d & Y dem is %d\n",
                        v1.x, v1.y);

 v2.x = 3345;
 v2.y = 534;
printf("X dem is %d & Y dem is %d\n",
                        v2.x, v2.y);

 return 0;

};
```

Output:

X dem is 34 & Y dem is 54

Q2) Write a function sumVectors which returns the sum of 2 vectors passed to it. The vectors must be 2 dimensional

```
#include<stdio.h>

struct vectors{
int x;
int y;
};

struct vectors sumVectors(struct vectors v1,
                          struct vectors v2){
struct vectors result;
result.x = v1.x + v2.x;
result.y = v1.y + v2.y;
return result;
}

int main(){
struct vectors v1, v2, sum;
v1.x = 4;
v1.y = 9;
printf("X dem is %d & Y dem is %d\n",
                    v1.x, v1.y);

v2.x = 5;
v2.y = 4;
printf("X dem is %d & Y dem is %d\n",
                    v2.x, v2.y);
```

```
sum = sumVector(v1, v2);
printf("x dem of result is %d & y dem
        is %d \n", sum.x, sum.y);
return 0;
}
```

Output

x dem is 4 & y dem is 9
11 5 4

x dem of result is 9 & y dem is 13

Q3) Twenty integers are to be stored in memory. What will you prefer - Array or Structure?

⇒ Array

Q4) Write a program with a structure representing a complex number

Q5) Create an array of 5 complex numbers created in problem 5 & display them with the help of a display function. The values must be taken as an input from the user

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

typedef struct complex{
    int real;
    int complex;
} comp;
```

```
void display(comp c){
printf("The value of real part is %d\n",
                                c.real);
printf("The value of Imaginary part is %d
                                c.complex);
}

int main(){
comp cnums[5];
for(int i=0;i<5;i++){
printf("Enter the real value of %d
                        num\n",i+1);
scanf("%d",&cnums[i].real);

printf("Enter the complex value for
                        %d num\n",i+1);
scanf("%d",&cnums[i].complex);
}

for(int i=0;i<5;i++){
    display(cnums[i]);
}

return 0;
}
```

Output:

```
Enter the real value for 1 num
1
Enter the complex value for 2 num
2
```

~~The~~ The rest it will go on for 5 numbers

The value of real part is I
complex
" Imaginary 2
.
.
.
}

Similarly for 5 nos

Q.6 Write a structure capable of storing a date. Write a function to compare these dates

```
#include<stdio.h>

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

void display(date d){
printf("The date is: %d/%d/%d\n",
            d.date, d.month, d.year);
}
int main(){
```

```
int dateCmp(date d1, date d2){
if(d1.year > d2.year){
    return 1;
}
if(d1.year < d2.year){
    return -1;
}
if(d1.month > d2.month){
    return 1;
}
if(d1.month < d2.month){
    return -1;
}
if(d1.date > d2.date){
    return 1;
}
if(d1.date < d2.date){
    return -1;
}
return 0;
}
```

```
int main(){
    date d1={2,11,21};
    date d2={5,4,22};

    display(d1);
    display(d2);

    return 0;
}
int a= datecmp(d1,d2);
printf("Date Comparison function returns:
        %d", a);
```

Output:

The date is: 2/11/21
— 11. —— : 5/4/22

Date comparison returns: -1