C Programming Lecture 15

Tuesday, 25 June 2024 8:18 PM

int a[10];

4B

int *a;

3 2 5 7 8

10x int

Structures in C

Structure in C is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member.

Structures

```
struct structure_name   {
    data_type member1;
    data_type member2;
    .
    .
    data_type memeberN;
};
```

struct employee e1;

int x;

x

struct employee{
int id;
char name[100];
float salary;
}

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

int x;

sizeof (int)

4B

sizeof (struct employee)

struct employee x;

x.id = 37;

x.name = " Ajaan";

x.salary = 10000000.00;

Note: static members are not allowed in Structures and Unions in C

Size of Structures

Total size of the structure is not always the sum of the sizes of all data members.
Let's understand this:

```
struct Readout{
    char hour;
    int value;
```

64 bit, size of char = 1B
int = 4B

Size of Structures

Total size of the structure is not always the sum of the sizes of all data members.
Let's understand this:

```
struct Readout{
   char hour;
   int value;
   char seq;
};
```

64 bit, size of char = 1B
int = 4B

struct employee x;
x.id = 37;
x.name = " Ajaan";
x.salary = 10000000.00;

- So the size of Readout on a 4 byte-int machine would be 12 bytes and not 6 bytes, this is due to the padding added by the compiler to avoid alignment issues.
- Padding is only added when a structure member is followed by a member with a larger size or at the end of the structure.
- Different compilers might have different alignment constraints as C standards state that alignment of structure totally depends on the implementation.
- C language doesn't allow the compilers to reorder the struct members to reduce the amount of padding. In order to minimize the amount of padding, the struct members must be sorted in a descending order

Use `#pragma pack(1)` to stop the default padding.

1B

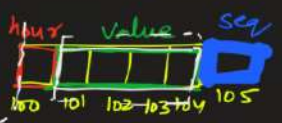
word-addressable

64-bit

1 block = 64-bits
8B

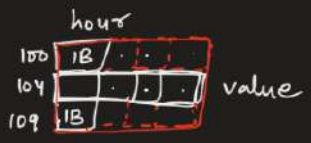
32-bit
1 block = 32-bits
= 4B

```
struct Readout {
100  char hour;  — 1B
101  int value;  — 4B
105  char seq;   — 1B
}
```

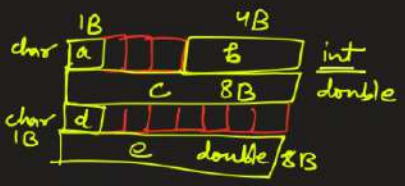


100 eg. 4B at a time

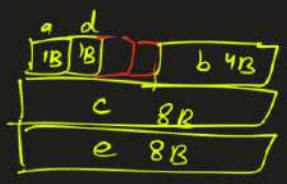
value




```
✓ char a;
✓ int b;
✓ double c;
  char d;
  double e;
```




```
char a
char d
int b
double c
double e
```



Declaring Structure Variables

```
struct employee  {
    int id;
    char name[50];
    float salary;
};
struct employee e1, e2;
```

[OR]

```
struct employee  {
    int id;
    char name[50];
    float salary;
}e1,e2;
```

Accessing Structure Members

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Using dot (.) operator

```
#include<stdio.h>
#include<stdlib.h>
struct employee {
    int age;
    char* name;
    float salary;
} e1;

void main() {
    e1.age = 30;
    e1.name = "Saurabh Jain";
    e1.salary = 30000.0;
    printf("%s with age %d has salary of INR %f", e1.name, e1.age, e1.salary);
}
```

Using struct pointer (->) operator

```
#include<stdio.h>
#include<stdlib.h>
struct employee{
    int age;
    char* name;
    float salary;
}e1;

void main(){
    struct employee*ptr= &e1;
    e1.age = 30;
    (*ptr).age = 10;
    e1.name = "Saurabh Jain";
    ptr->name = "Sakshi";
    e1.salary = 30000.0;
    printf("%s with age %d has salary of INR %f", e1.name, ptr->age, (*ptr).salary);
}
```

Data segment

```
    printf("%s with age %d has salary of INR %f", e1.name, ptr->age, (*ptr).salary);
}
```

```
struct Employee {
    int age;
    char * name;
    float salary;
} e1;
```

Data segment

e1

age	name
salary	

100

100

ptr *
struct Employee

```
struct Employee * ptr = &e1
```

```
e1.age = 30;
```

```
(*ptr).age = 10;
≡ ptr -> age = 10;
```

Typedef in C

The typedef is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

```
typedef <existing_name> <alias_name>
```

```
typedef unsigned int uint;

uint a, b;
```

Typedef and Structures

```
typedef struct student
// you can skip writing student here
{
   char name[20];
   int age;
} stu;

stu s1,s2;
```

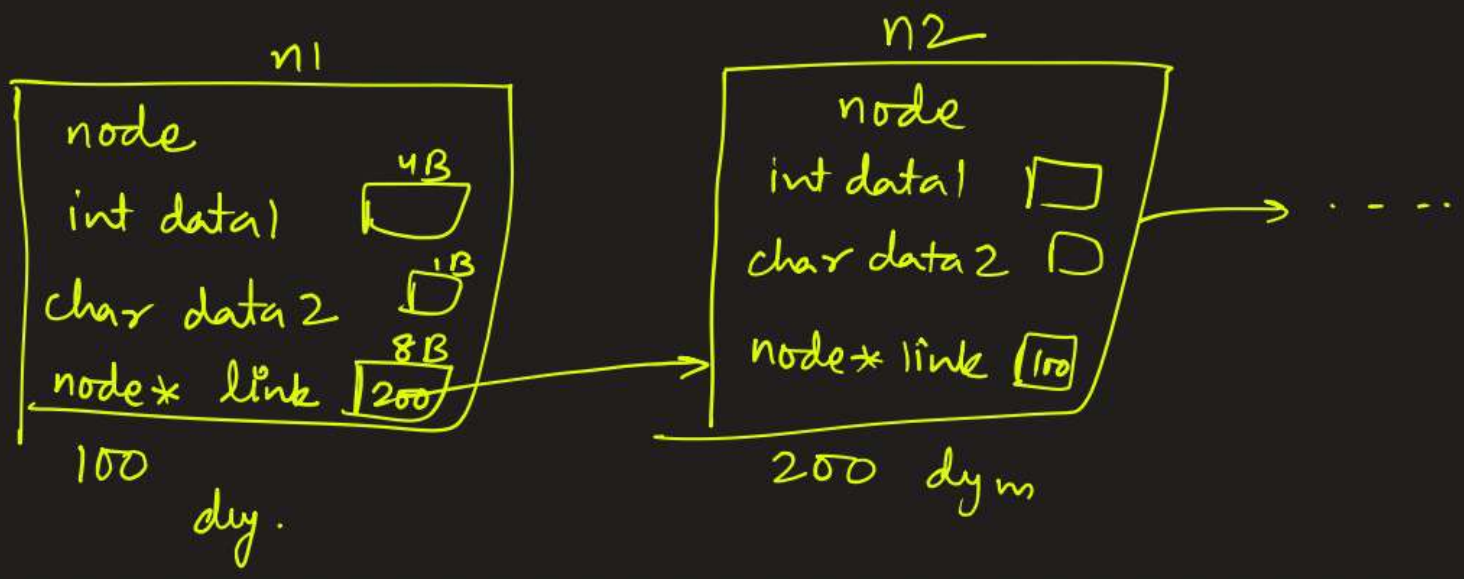
Typedef and Pointers

```
typedef int* ptr;
ptr p1, p2 ;
```

```
struct Employee

typedef struct Employee emp;

emp e1, e2;

emp* e1 = &e2;

emp* e1 = (emp*)malloc(sizeof(emp));

e1->a = —
e1->b = —
```

Self Referential Structures

```
struct node {
    int data1;
    char data2;
    struct node* link;
};
```

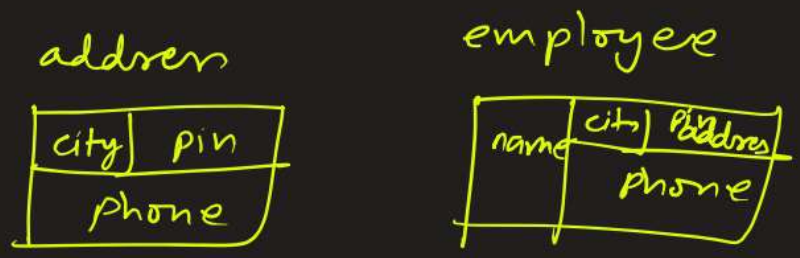


Nested Structures

```c
struct address
{
    char city[20];
    int pin;
    char phone[14];
};
struct employee
{
    char name[20];
    struct address add;
};
```

[OR]

```c
struct employee
{
    char name[20];

   struct address
   {
       char city[20];
       int pin;
       char phone[14];
   } add;
};
```




```c
struct
    employee e1;
    e1.name = "Shubham";
    e1.add.pin = 300100;
```

Passing & Returning Structures

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct{
    intage;
    char* name;
    floatsalary;
} Emp;
void fun1(Emp e){
    e.age = 40;
    e.name = "Rohan";
    printf("[Inside fun1] %s with age %d has salary of INR %f \n", e.name, e.age, e.salary);
}
void fun2(Emp *e){
    e->age = 25;
    e->name = "Arnab";
    printf("[Inside fun2] %s with age %d has salary of INR %f \n", e->name, e->age, e->salary);
}
void main(){
    Emp e1;
    e1.name = "Saurabh Jain";
    e1.salary = 30000.0;
    e1.age = 32;
    fun1(e1); // Call by value
    printf("%s with age %d has salary of INR %f \n", e1.name, e1.age, e1.salary);
    fun2(&e1); // Call by reference
    printf("%s with age %d has salary of INR %f \n", e1.name, e1.age, e1.salary);
}
```

Union in C

- A **union** is like a **struct** in which all members are allocated at the same address so that the union occupies only as much space as its largest member.
- It can only hold value for one member at a time.
  For example, if we want to store address,

```
union Address {
   const char* name;     // "Sameer"
   int number;           // 45
   const char* street;   // "Tilak Nagar"
};
```

Size of Unions

The size of a union is taken according the size of largest member in it.
Let's understand this:

```
union Readout{
   char hour;     1B
   int value;     4B
   char seq;      1B
};
```

4B = max(1, 4, 1)

So the size of Readout on a 4 byte-int machine would be 4 bytes and not 6 bytes, this is due to using same memory for all members.

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct{
    char* name;
    int age;
    struct address{
        char* street;
        int pincode;
        char* state;
    }add;
    char* phone;
    union class10marks{
        float percentage;
        float cpi;
        char grade;
    }c10;
}stu;
void main(){
    stu s1;
    s1.name = "Shivam";
    s1.age = 21;
    s1.add.street = "Road 1, BKC, Bombay";
    s1.add.pincode = 500074;
    s1.add.state = "Maharashtra";
    s1.phone = "9000000000";
    s1.c10.percentage = 87.9;
    printf("%f", s1.c10.cpi);
}
```

```
union {
 char a, b;
  int c;
};
```