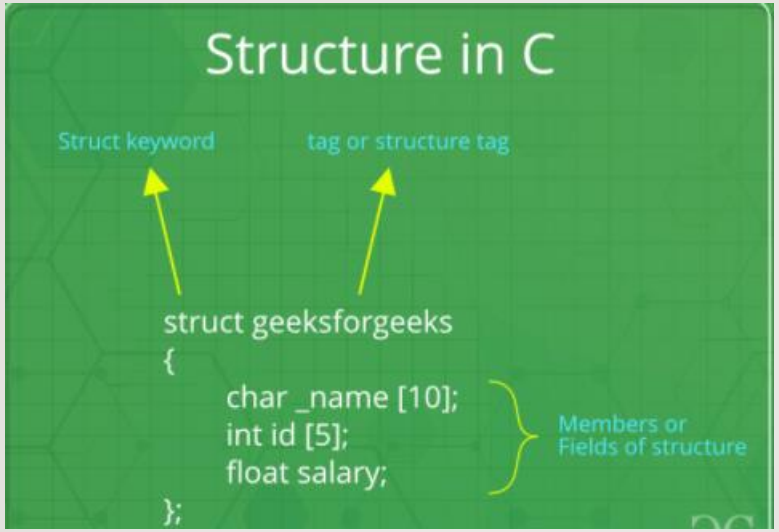
AMIT LEARNING

SESSION_6 : USER DEFINED DATA TYPES

CONTENTS

AMIT

- What is structure:
- A structure is a user defined data type in C .A structure creates a data type that can be used to group items of possibly different types into a single type.



➢ How to create a structure?

'struct' keyword is used to create a structure. Following is an example.

```
struct address
{
   char name[50];
   char street[100];
   char city[50];
   char state[20];
   int pin;
};
```

➢ How to declare a structure variable?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
// A variable declaration with structure declaration.
struct Point
{
   int x, y;
} p1;  // The variable p1 is declared with 'Point'


// A variable declaration like basic data types
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1;  // The variable p1 is declared like a normal variable
}
```

- ***How to initialize structure members?***
  Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
   int x = 0;  // COMPILER ERROR:  cannot initialize members here
   int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

- The reason for above error is simple, when a data type is declared, no memory is allocated for it. Memory is allocated only when variables are created.

- Structure members can be initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point
{
   int x, y;
};

int main()
{
   // A valid initialization. member x gets value 0 and y
   // gets value 1.  The order of declaration is followed.
   struct Point p1 = {0, 1};
}
```

How to access structure element?

- Structure members are accessed using dot (.) operator.

What is designated Initialization?

Designated Initialization allows structure members to be initialized in any order. This feature has been added in C99 standard.

Example:

```c
struct Point
{
    int x, y, z;
};

int main()
{
    // Examples of initialization using designated initialization
    struct Point p1 = {.y = 0, .z = 1, .x = 2};
    struct Point p2 = {.x = 20};

    printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
    printf ("x = %d", p2.x);
    return 0;
}
```

Output:

```
x = 2, y = 0, z = 1
x = 20
```

1 Struct

1.1 Why using struct

CODE

```c
#include<stdio.h>

struct Point
{
    int x, y;
};

int main()
{
    struct Point p1 = {0, 1};

    // Accessing members of point p1
    p1.x = 20;
    printf ("x = %d, y = %d", p1.x, p1.y);

    return 0;
}
```

OUTPUT

Output:

```
x = 20, y = 1
```

DESCRIPTION

- Structure members are accessed using dot (.) operator

CODE

```c
#include <stdio.h>
#include <stdlib.h>


   struct ahmed{
   char x;
   int y ;
   float z;
   }i,j,k;


int main()
{
   struct ahmed b;

   return 0;
}
```

DESCRIPTION

There are two ways to make an object from a struct

1 – Declare the struct above main and create the object inside the function like object b

2 – Or you can declare the objects inside the struct like objects I, j, k

1 Struct

1.2 Struct initialization -> How to access the members

CODE

```c
#include <stdio.h>
#include <stdlib.h>

    struct student{
    int id;
    char name[10];
    char age ;
    float grade;
    };

 int main()
 {
    struct student student_1 = {122,"mohamed",20,88.5};

    student_1.id    = 123;
    strcpy(student_1.name,"Ahmed");
    student_1.age   = 20;
    student_1.grade = 90;

    printf("ID    = %d\n",student_1.id);
    printf("Name  = %s\n",student_1.name);
    printf("Age   = %d\n",student_1.age);
    printf("Grade = %g\n",student_1.grade);
    return 0;
 }
```

OUTPUT

```
ID    = 123
Name  = Ahmed
Age   = 20
Grade = 90
```

DESCRIPTION

In this example we create a new data type called “struct student”

We create an object from this struct called “student_1”

In structs there are two ways to access members and write the data :

One : you can write the data of all members in the initialization line

Two : you can access member by member

Note : we must use strcpy function in strings because cannot assign name of array .

TASK

Implement a database for Egyptian league using struct "football_team"

Create an object from this struct and name it with your favorite club

the struct members are : int established_year

char rank_in_league

char captain_name [20]

And print all members

OUTPUT EX.

```
Established year : 1907
Rank in league   : 1
Captain name     : EL SHENAAWY
```

CODE

```c
#include <stdio.h>
#include <stdlib.h>

   struct football_team{
   int  established_year;
   char rank_in_league;
   char captain_name[20];
   };

int main()
{
   struct football_team al_ahly = {1907,1,"EL SHENAAWY"};;

   printf("Est.     : %d\n",al_ahly.established_year);
   printf("Rank     : %d\n",al_ahly.rank_in_league);
   printf("Captian  : %s\n",al_ahly.captain_name);

   return 0;
}
```

DESCRIPTION

1 -> Create a new data type using struct named with “football_team”.

2 -> Make an object from this new data type called “al_ahly”.

3 -> Access the members and fill the data.

4 -> Print all members.

1 Struct

1.2 Struct initialization -> Struct assignation

CODE

```c
#include <stdio.h>
#include <stdlib.h>

struct student{
char  name[20];
int   id;
char  age;
float grade;
};

int main()
{
   struct student x = {"Mohamed",5664,20,95.5};
   struct student y;
   y = x;
   printf("Student Name  :%s\n",y.name);
   printf("Student ID    :%d\n",y.id);
   printf("Student Age   :%d\n",y.age);
   printf("Student Grade :%g\n",y.grade);
   return 0;
}
```

OUTPUT

```
Student Name  : Mohamed
Student ID    : 5664
Student Age   : 20
Student Grade : 95.5
```

DESCRIPTION

You can define an object from a struct like object x

Then make another object like y

y = x ; ⟶ This line makes member 1 in y = member 1 in x

member 2 in y = member 2 in x

member 3 in y = member 3 in x

member 4 in y = member 4 in x

- In real-world processor, does not read or write the memory byte by byte but actually, for performance reason, The processor only accesses the byte, half word, word, or double word at a time.

- In 32 bits processor word size is 4 bytes if the data address within the 32 bits then it perfectly fit in the memory alignment but if it crosses the boundary of 32 bits then the processor has to take some extra cycles to fetch the data from that unaligned memory.

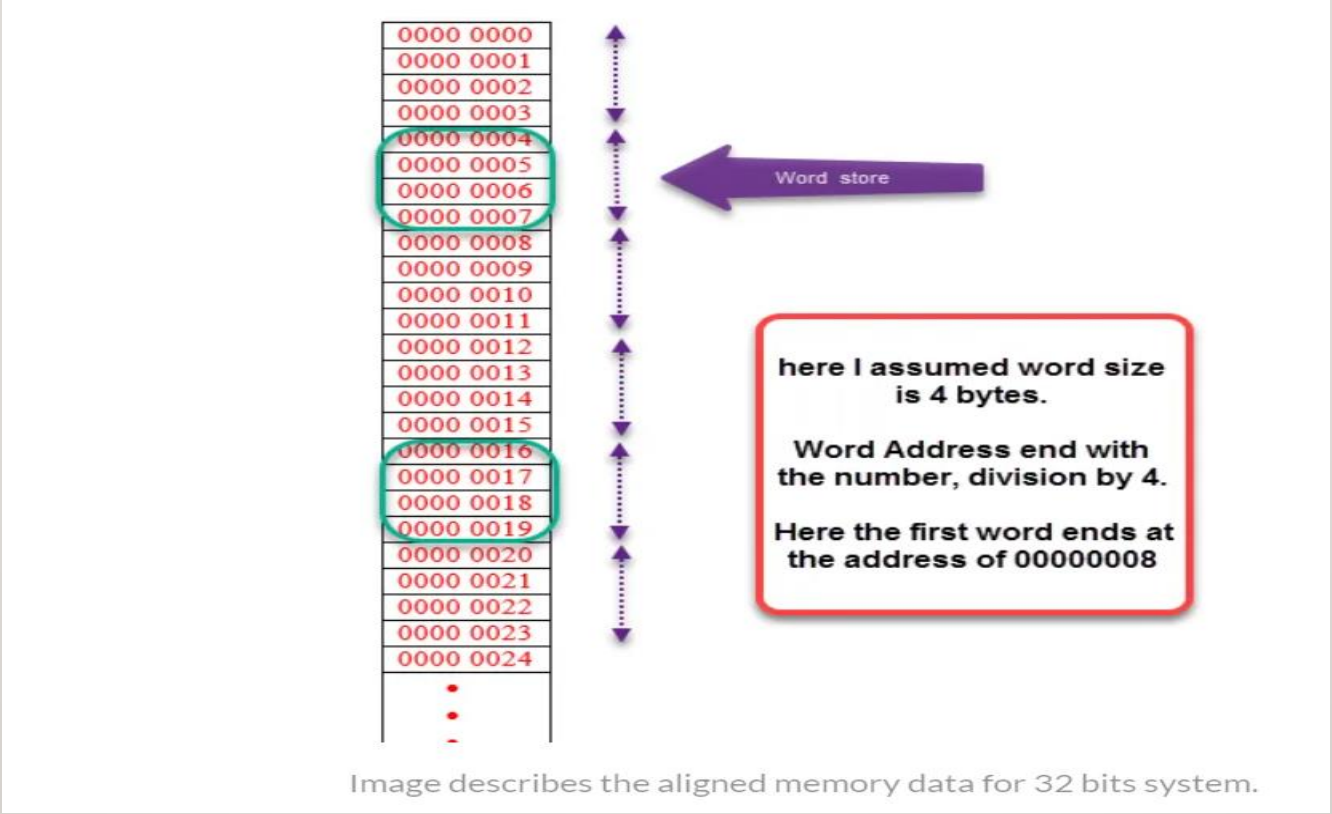
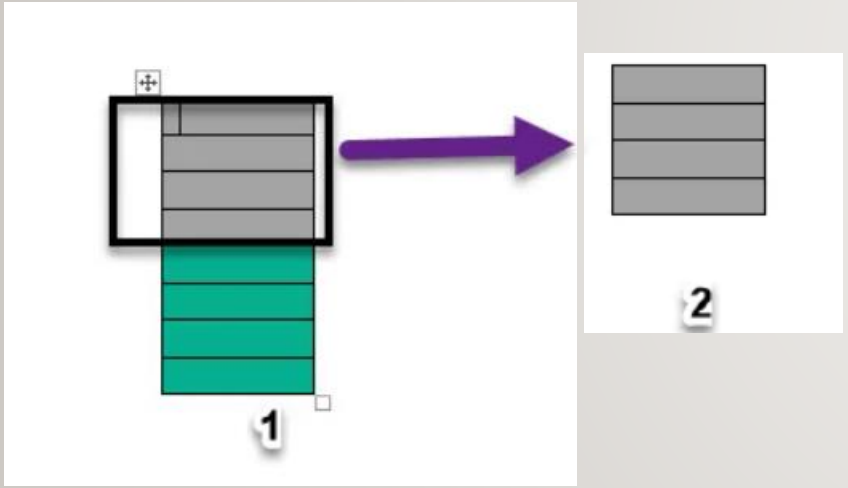


Image describes the aligned memory data for 32 bits system.

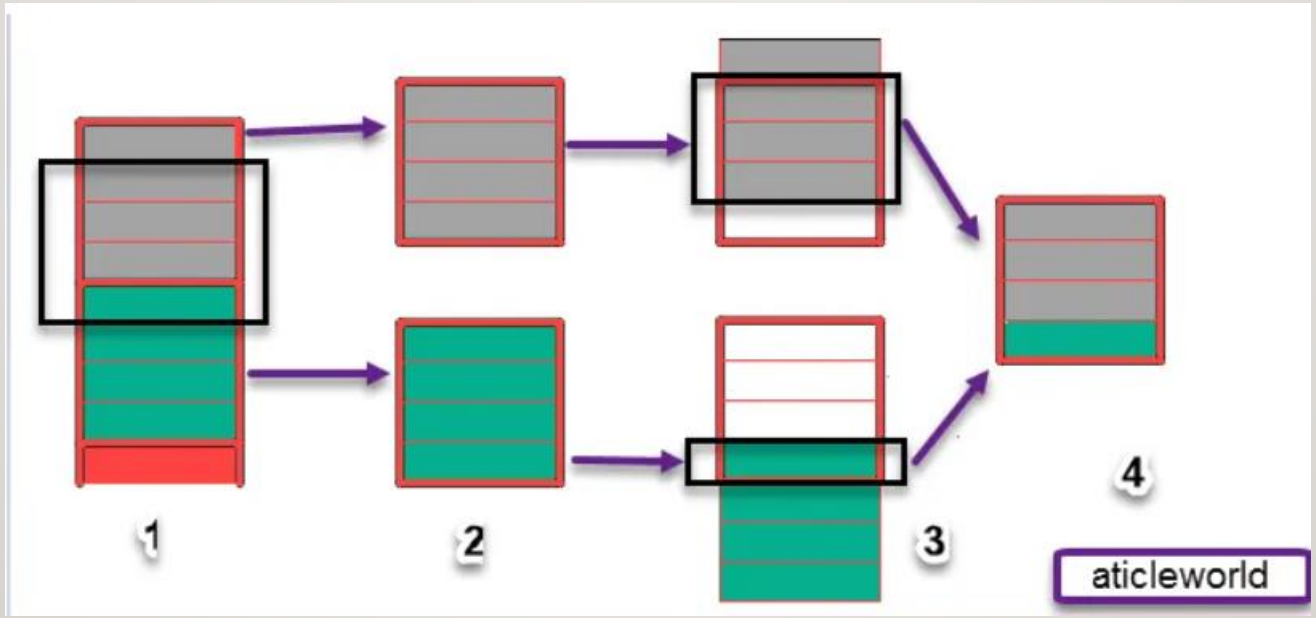
- When memory is aligned then the processor easily fetches the data from the memory. In images 1 and 2, you can see that the processor takes one cycle to Access the aligned data.

- **Note:** Alignment of data types mandated by the processor architecture, not by language.



- **Below image describes the steps, how is the processor access the unaligned memory**
- When the processor gets an unaligned memory then it takes the following steps to access the unaligned memory.

1.CPU selects the unaligned memory which represents through the black dark border.
2.CPU access the whole 4 bytes of above and below the black square border.
3.Shift one byte above and three bytes below in corresponding to the above and below chunks of memory.
4.Combined both chunks of data and get the actual bytes of data.



- The RISC processor throws the exception when he faced the unaligned memory but some MIPS have some special instruction to handle the unaligned memory situation, unaligned memory is not big issues for Intel x86 processor it easily handles the unaligned memory but sometimes it takes some extra ticks to fetch the unaligned memory.

- In the program, there are mainly two-property attached to the variable first is the value of the variable and second its address. In the case of the Intel X86 architecture address of the variable in the multiple of 1, 2, 4 or 8, in other words, we can say that the address of variable should be multiple of the power of 2.

- Generally, the compiler handles the scenario of alignment and it aligned the variable in their boundary. We don't need to worry about the alignment, in the 32 bits X86 architecture alignment of data type generally similar to their length.

- **In below table, I have described the alignment of some primitive data type which frequently used in the program**

1 Struct

1.3 Struct size

Data Type	32-bit (bytes)	64-bit (bytes)
char	1	1
short	2	2
int	4	4
float	4	4
double	8	8
pointer	4	8

- When you create the **structure** or **union** then compiler inserts some extra bytes between the members of structure or union for the alignment. These extra unused bytes are called padding bytes and this technique is called structure padding in C.

- Padding increases the performance of the processor at the penalty of memory. In structure or union data members aligned as per the size of the highest bytes member to prevent the penalty of performance

Example 1:

```
typedef struct
{
    char A;
    int B;
    char C;
} InfoData;
```

Memory layout of structure InfoData

- In the above structure, an integer is the largest byte size member. So to prevent the penalty compiler inserts some extra padding bytes to improve the performance of the CPU. So the size of the InfoData will be 12 bytes due to the padding bytes inserted by the compiler for the data alignment.

- **Note:** In the case of structure and union we can save the wastage of memory to rearrange the structure members in the order of largest size to smallest.

```
typedef struct
{
    int A;
    char B;
    char C;
} Element;
```

Memory layout of Element after the rearranging of his members

➢ How to avoid Structure Padding in C?

- If you want you can avoid the structure padding in C using the pragma pack (#pragma pack(1)) or attribute (__attribute__((__packed__))).
- The #pragma effect on the hole program it changes the behavior of the compiler and its not only for the memory padding in struct.Let us see an example code,
- rearrange the structure members in the order of largest size to smallest.

```c
#pragma pack(push, 2)
typedef struct
{
    double A; // 8-byte
    char B; // 1-byte
    char C;   // 1-byte

} InfoData;
#pragma pack(pop)


/* main function */
int main(int argc, char *argv[])
{
    printf("\n Size of Structure = %d\n\n\n\n",sizeof(InfoData));

    return 0;
}
```

```
Size of Structure = 10
```

```c
//Using the default packing of compiler
typedef struct
{
    double A; // 8-byte
    char B; // 1-byte
    char C;   // 1-byte

} InfoData;


/* main function */
int main(int argc, char *argv[])
{
    printf("\n Size of Structure = %d\n\n\n\n",sizeof(InfoData));

    return 0;
}
```

```
Size of Structure = 16
```

```c
#include <stdio.h>
#include <stdlib.h>

   struct example{
   char x;
   int  y;
   char z;
   }example_1;

int main()
{
   printf("%d byte\n",sizeof(example_1));
   return 0;
}
```

OUTPUT

12 Bytes

DESCRIPTION

How the complier calculate the size of struct ?

Why size of example_1 = 12 bytes not 6 bytes ?

Because when the complier calculate the struct size the first thing the compiler do is searching for the biggest data type which is int in our case and int size = 4 bytes, The compiler reserve 4 bytes and allocate x in the first byte, The 3 remainder bytes is not enough for the next element which is int, So the complier reserve another 4 bytes and allocate y in all of them, Then reserve another 4 bytes and allocate z in the first byte. Now there are 6 bytes we reserved in RAM and did not use them, this problem is called PADDING

CODE

```c
#include <stdio.h>
#include <stdlib.h>

   struct example{
   int  y;
   char x;
   char z;
   char v;
   char b;
   }example_1;

int main()
{
   printf("%d byte\n",sizeof(example_1));
   return 0;
}
```

OUTPUT

```
8 Bytes
```

DESCRIPTION

First thing the compiler searching for the biggest data type which is int,

The compiler reserve 4 bytes and allocate y in all of them, Then reserve another 4 bytes and allocate x in the first byte, z in the second byte, v in third, and b in the last byte.

Now we reserved 8 bytes and use all of them.

But what if there was another variable after b

Let us see in the next example

1 Struct

1.3 Struct size -> Example (2)

CODE

```c
#include <stdio.h>
#include <stdlib.h>

   struct example{
   int  y;
   char x;
   char z;
   char v;
   char b;
   char j;
   }example_1;

int main()
{
   printf("%d byte\n",sizeof(example_1));
   return 0;
}
```

OUTPUT

```
12 Bytes
```

DESCRIPTION

First thing the compiler searching for the biggest data type which is int, The compiler reserve 4 bytes and allocate y in all of them, Then reserve another 4 bytes and allocate x in the first byte, z in the second byte, v in third, and b in the last byte. Now there is another char member called j, the compiler will reserve another 4 bytes and allocate j in the first one.

Why reserve 4 bytes not 1 byte -> because the compiler reserve with the biggest data type in struct which is int in this example

Note : We noted that we can reduce the padding problem by rearranging the members in struct.

Last Thing you should to remembre this The main concept is not about the largest var in the struct it's about the processor how he will access the data it depends which type of accesses he uses byte, half word, word, or double word.

CODE

```c
#include <stdio.h>
#include <stdlib.h>
#pragma pack(1)

    struct example{
    int  y;
    char x;
    char z;
    char v;
    char b;
    char j;
    }example_1;

int main()
{
    printf("%d byte\n",sizeof(example_1));
    return 0;
}
```

OUTPUT

```
9 Bytes
```

DESCRIPTION

There is another way we can solve padding problem called packing

We can use #pragma

However it starts with # but it is **not preprocessor directive**

#pragma pack(1) force the compiler to reserve space in RAM by 1 byte

And ignore the biggest data type inside the struct

So in this code size of example_1 is 9 bytes because the compiler

reserve 1byte then 1byte then 1 byte then 1byte for y

Then 1byte for x then 1byte for z then 1byte for v then 1byte for b then

1 byte for j .

So the total bytes are 9 bytes

CODE

```c
#include <stdio.h>
#include <stdlib.h>
#pragma pack(1)

    struct example{
    char x;
    int  y;
    char z;
    char j;
    }example_1;



int main()
{
    printf("%d byte\n",sizeof(example_1));
    return 0;
}
```

OUTPUT

```
7 Bytes
```

DESCRIPTION

In this code size of example_1 is 7 bytes because the compiler reserve 1byte for x, then 1byte then 1byte then 1 byte then 1byte for y

Then 1byte for z then 1byte for j.

So the total bytes are 7 bytes

Note : #pragma save spaces in RAM but it reduce the execution time of the program

CODE

```c
#include <stdio.h>
#include <stdlib.h>
#pragma pack(2)

    struct example{
    char x;
    int  y;
    char z;
    char j;
    }example_1;



int main()
{
    printf("%d byte\n",sizeof(example_1));
    return 0;
}
```

OUTPUT

```
8 Bytes
```

DESCRIPTION

#pragma pack(2) force the compiler to reserve space in RAM by 2 bytes

So in this code size of example_1 is 8 bytes because the compiler reserve 2bytes and allocate x in the first byte, then 2bytes then 2 bytes for y, then 2bytes for z and j.

So the total bytes are 8 bytes, and there is 1byte padding.

1 Struct

1.5 Bit field in struct -> Example (6)

CODE

```c
#include <stdio.h>
#include <stdlib.h>

    struct values{
    char x:3;
    char y:2;
    char z:1;
    char k:2;
    }v1;

int main()
{
    printf("%d",sizeof(v1));
    return 0;
}
```

OUTPUT

```
1 Bytes
```

DESCRIPTION

What is bit field inside struct ?

C language allows us to **determine a specific number of bits**

for a member inside the struct

There is no #pragma here so the compiler will searching for the biggest

data type which is char.

So the compiler will reserve 1 byte in RAM.

Instead of reserving 1 byte (8 bits) for x, in bit field, x will take only 3 bits

2 bits for y, 1 for z, and 2 bits for k.

3+2+1+2 = 8 bits (1byte)

1 Struct

1.5 Bit field in struct -> Example (7)

CODE

```c
#include <stdio.h>
#include <stdlib.h>

    struct values{
    int x:5;
    int y:11;
    int z:1;
    int k:15;
    }v1;

int main()
{
    printf("%d",sizeof(v1));
    return 0;
}
```

OUTPUT

```
4 Bytes
```

DESCRIPTION

In this example the biggest data type is int so the compiler will reserve

4 bytes (32bits) allocate 5 bits for x, 11 for y, 1 for z, and 15 bit for k

The total bits are 32 bits (4bytes)

CODE

```c
#include <stdio.h>
#include <stdlib.h>

    struct values{
    int x:5;
    int y:11;
    int z:1;
    int k:16;
    }v1;

int main()
{
    printf("%d",sizeof(v1));
    return 0;
}
```

OUTPUT

```
8 Bytes
```

DESCRIPTION

In this example the biggest data type is int so the compiler will reserve

4 bytes (32bits) allocate 5 bits for x, 11 for y, 1 for z.

Now there are 15 bits available, but we need 16 bits for k

So there are not enough space for k

The compiler reserve another 4bytes and allocate k in 16 bitts of them.

1 Struct

1.5 Bit field in struct -> Example (9)

CODE

```c
// A simple representation of the date
struct date {
    unsigned int d;
    unsigned int m;
    unsigned int y;
};

int main()
{
    printf("Size of date is %lu bytes\n",
           sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

OUTPUT

```
Size of date is 12 bytes
Date is 31/12/2014
```

DESCRIPTION

- The above representation of 'date' takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, the value of m is from 1 to 12, we can optimize the space using bit fields.

1 Struct

1.5 Bit field in struct -> Example (10)

CODE

```
// Space optimized representation of the date
struct date {
    // d has value between 0 and 31, so 5 bits
    // are sufficient
    int d : 5;

    // m has value between 0 and 15, so 4 bits
    // are sufficient
    int m : 4;

    int y;
};

int main()
{
    printf("Size of date is %lu bytes\n",
           sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
    return 0;
}
```

OUTPUT

```
Size of date is 8 bytes
Date is -1/-4/2014
```

DESCRIPTION

- A special unnamed bit field of size 0 is used to force alignment on next boundary.
- d=1111 is MSB IS (-) so output -1
- M=1000 is MSB (-) so output -4

CODE

```c
#include <stdio.h>
#include <stdlib.h>

   struct values{
   int  x:32;
   char y:8;
   char z:8;
   char k:8;
   char l:8;
   char f:1;
   }v1;

int main()
{
   printf("%d bytes",sizeof(v1));
   return 0;
}
```

OUTPUT

```
12 Bytes
```

DESCRIPTION

In this example the biggest data type is int so the compiler will reserve 4 bytes (32bits) allocate all of them for x, then reserve another 4bytes(32bits) and allocate 8 bits for y, 8 for z, 8 for k, and 8bits for l.

Now there is no space and still f need 1bit.

The complier will reserve 4 bytes (32 bits) and allocate f in just 1 bit.

Following are some interesting facts about bit fields in C

- A special unnamed bit field of size 0 is used to force alignment on next boundary
- We cannot have pointers to bit field members as they may not start at a byte boundary.
- It is implementation defined to assign an out-of-range value to a bit field member.
- Array of bit fields is not allowed. fails in the compilation.
- Use bit fields in C to figure out a way whether a machine is little-endian or big-endian.

CODE

```c
#include <stdio.h>
struct test {
    unsigned int x : 5;
    unsigned int y : 5;
    unsigned int z;
};
int main()
{
    struct test t;

    // Uncommenting the following line will make
    // the program compile and run
    printf("Address of t.x is %p", &t.x);

    // The below line works fine as z is not a
    // bit field member
    printf("Address of t.z is %p", &t.z);
    return 0;
}
```

OUTPUT

```
prog.c: In function 'main':
prog.c:14:1: error: cannot take address of bit-field 'x'
 printf("Address of t.x is %p", &t.x);
```

DESCRIPTION

- We cannot have address to bit field members as they may not start at a byte boundary.

1 Struct

1.6 Bit field in struct

CODE

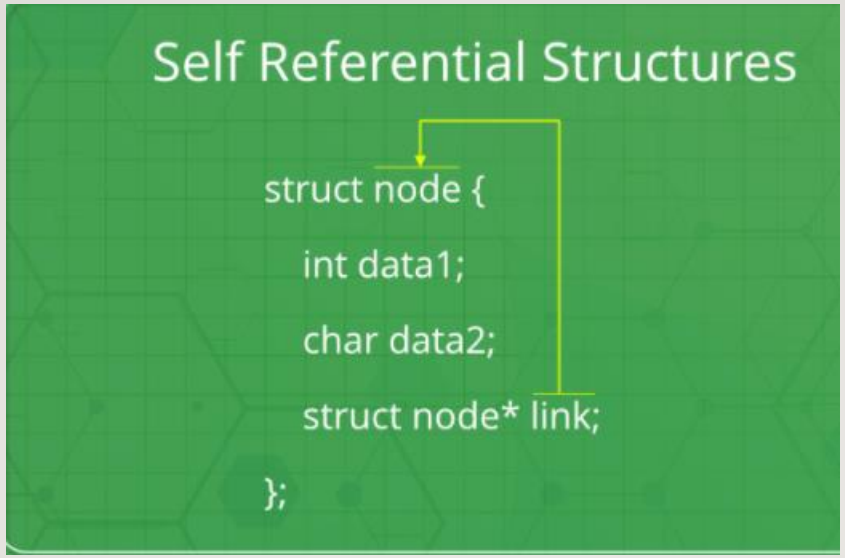
```
struct test {
    unsigned int x[10] : 5;
};

int main()
{
}
```

OUTPUT

```
prog.c:3:1: error: bit-field 'x' has invalid type
 unsigned int x[10]: 5;
 ^
```

DESCRIPTION

- Array of bit fields is not allowed.

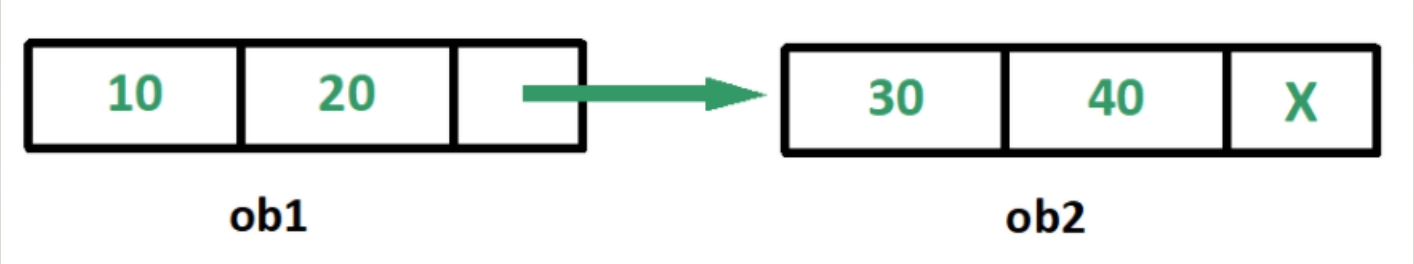


- **Types of Self Referential Structures**

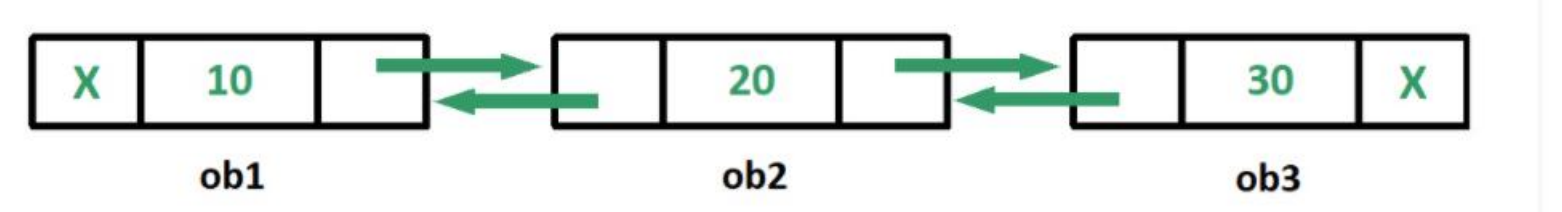
1.Self Referential Structure with Single Link.

2.Self Referential Structure with Multiple Links.

➢ Self Referential Structure with Single Link:



➢ Self Referential Structure with Multiple Links:



1 Struct

1.6 Nested struct (Self Referential Structures)

CODE

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

int main()
{
    struct node ob1; // Node1

    // Initialization
    ob1.link = NULL;
    ob1.data1 = 10;
    ob1.data2 = 20;

    struct node ob2; // Node2

    // Initialization
    ob2.link = NULL;
    ob2.data1 = 30;
    ob2.data2 = 40;

    // Linking ob1 and ob2
    ob1.link = &ob2;

    // Accessing data members of  ob2 using ob1
    printf("%d", ob1.link->data1);
    printf("\n%d", ob1.link->data2);
    return 0;
}
```

OUTPUT

```
30
40
```

DESCRIPTION

1.Self Referential Structure with Single Link

1 Struct

1.6 Nested struct (Self Referential Structures)

CODE

```
struct node {
    int data;
    struct node* prev_link;
    struct node* next_link;
};

int main()
{
    struct node ob1; // Node1

    // Initialization
    ob1.prev_link = NULL;
    ob1.next_link = NULL;
    ob1.data = 10;

    struct node ob2; // Node2

    // Initialization
    ob2.prev_link = NULL;
    ob2.next_link = NULL;
    ob2.data = 20;
```

```
// Accessing  data of ob1, ob2 and ob3 by ob1
printf("%d\t", ob1.data);
printf("%d\t", ob1.next_link->data);
printf("%d\n", ob1.next_link->next_link->data);

// Accessing data of ob1, ob2 and ob3 by ob2
printf("%d\t", ob2.prev_link->data);
printf("%d\t", ob2.data);
printf("%d\n", ob2.next_link->data);
```

OUTPUT

```
10    20    30
10    20    30
```

DESCRIPTION

- **Self Referential Structure with Multiple Links**

AMIT

1 Struct

1.6 Nested struct (Self Referential Structures)

CODE

```c
#include <stdio.h>
#include <stdlib.h>

struct test{
  struct n{
  int a;
  int b;
  int c;
  }n_1;
  struct m{
  int a;
  int b;
  int c;
  }m_1;
}t;

int main()
{
    t.n_1.a = 10;
    printf("%d",t.n_1.a);
    return 0;
}
```

OUTPUT

```
10
```

DESCRIPTION

Nested struct means struct inside a struct

t.n_1.a = 10; ⟶ this line mean access struct t

then access struct n_1

then access member a

and make it = 10

1 Struct

1.6 Nested struct (Self Referential Structures)

CODE

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

struct student_college_detail
{
    int college_id;
    char college_name[50];
};

struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct student_college_detail clg_data;
}stu_data;

int main()
{
    struct student_detail stu_data = {1, "Raju", 90.5, 71145,
                                      "Anna University"};
    printf(" Id is: %d \n", stu_data.id);
    printf(" Name is: %s \n", stu_data.name);
    printf(" Percentage is: %f \n\n", stu_data.percentage);

    printf(" College Id is: %d \n",
                    stu_data.clg_data.college_id);
    printf(" College Name is: %s \n",
                    stu_data.clg_data.college_name);
    return 0;
}
```

OUTPUT

```
Id is: 1
Name is: Raju
Percentage is: 90.500000
College Id is: 71145
College Name is: Anna University
```

DESCRIPTION

- Struct inside struct.

1 Struct

1.6 Nested struct and using in declare struct pointer variable and struct variable

CODE

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

struct student_college_detail
{
    int college_id;
    char college_name[50];
};

struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct student_college_detail clg_data;
}stu_data, *stu_data_ptr;

int main()
{
  struct student_detail stu_data = {1, "Raju", 90.5, 71145,
                                    "Anna University"};
    stu_data_ptr = &stu_data;

    printf(" Id is: %d \n", stu_data_ptr->id);
    printf(" Name is: %s \n", stu_data_ptr->name);
    printf(" Percentage is: %f \n\n",
                        stu_data_ptr->percentage);

    printf(" College Id is: %d \n",
                        stu_data_ptr->clg_data.college_id);
    printf(" College Name is: %s \n",
                       stu_data_ptr->clg_data.college_name);

    return 0;
}
```

OUTPUT

```
Id is: 1
Name is: Raju
Percentage is: 90.500000
College Id is: 71145
College Name is: Anna University
```

DESCRIPTION

- Nested struct using struct pointer and struct variable.

CODE

```c
#include <stdio.h>
#include <stdlib.h>

    struct data{
    int y[3] ;
    }a,b,c;

int main()
{
    a.y[0]=20;
    a.y[1]=30;
    a.y[2]=40;
    int i;
    for(i=0;i<3;i++)
    printf("%d\n",a.y[i]);
    return 0;
}
```

OUTPUT

```
20
30
40
```

DESCRIPTION

In this example there is a struct consists of array

From this struct we made 3 objects a, b and c.

First we have to access the object a for example

Inside a there is array of 3 elements

We can access any element of them.

CODE

```c
#include <stdio.h>
#include <stdlib.h>

   struct data{
   int a,b,c;
   }arr[10];

int main()
{
   arr[0].a = 20;
   arr[0].b = 30;
   arr[0].c = 40;

   printf("%d\n",arr[0].a);
   printf("%d\n",arr[0].b);
   printf("%d\n",arr[0].c);

   return 0;
}
```

OUTPUT

```
20
30
40
```

DESCRIPTION

In this example there is a struct consists of 3 members a, b and c

From this struct we made array of 10 elements.

First we have to access the array element arr[0] for example

Inside arr[0] there is a struct of 3 members.

We can access any member of them.

CODE

```c
#include <stdio.h>
#include <stdlib.h>

   struct student{
   int   id;
   char  age;
   };

int main()
{
   struct student student_1 = {55,22};
   struct student *p;
   p = &student_1;
   (*p).id = 66;
    p->age = 20;

   printf("Student ID  : %d\n",student_1.id);
   printf("Student Age : %d\n",student_1.age);

   return 0;
}
```

OUTPUT

```
Student ID  : 66
Student Age : 20
```

DESCRIPTION

Like any other data type, we can make a pointer points to an object from a struct.

And we can access any member through this pointer.

The use of pointers with struct is very important specially with functions

CODE

```c
#include <stdio.h>
#include <stdlib.h>

struct employee{
char age;
int salary;
char week;
}emp1;

void display(struct employee*);

int main()
{
emp1.salary=2000;
display(&emp1);
return 0;
}

void display(struct employee*ptr)
{
printf("%d",ptr->salary);
}
```

OUTPUT

```
2000
```

DESCRIPTION

In this example we declare a struct called struct employee and make an object emp1 from this struct.

We pass emp1 address to the function display.

The function display receive the address in pointer to struct employee

Now we can access any member in emp1 through the pointer ptr

Note : You have to write the function proto type below the struct

2 Union

2.1 Union size

CODE

```c
#include <stdio.h>
#include <stdlib.h>

union v{
unsigned char x;
unsigned int  y;
}v1;
int main()
{
v1.x=1;
v1.y=256;
printf("Size = %d bytes\n",sizeof(v1));
printf(" x    = %d\n",v1.x);
printf(" y    = %d\n",v1.y);
}
```

OUTPUT

```
Size = 4 bytes
 x    = 0
 y    = 256
```

DESCRIPTION

In this example we create a union v and make an object v1

Size of v1 = 4 bytes why ?

Because the size of the union = the size of the biggest data type inside it

The compiler reserve just 4 bytes in RAM for v1

V1.x = 1 -> makes x =1

V1.y = 256 -> makes y = 256 and the value of x is gone

Because v1 has only 4 bytes and all members sharing this 4 bytes.

2 Union

2.2 Why using union

CODE

```
#include <stdio.h>
#include <stdlib.h>
struct data_base{
   int price;
   struct books{
    char name[12];
    int pages;
    int sales;
   }book;
   struct shirts{
    char color[12];
    int size;
    int model;
   }shirt;
   struct computers{
    char brand[12];
    int ram;
    int hdd;
   }computer;
}souq;
int main()
{
   printf("%d Byte",sizeof(souq));
   return 0;
}
```

OUTPUT

```
64 Byte
```

DESCRIPTION

In this example the size of souq = 64 bytes

However in any database you can not access all these members at the same time.

You can access one member at the time.

So why reserving space in RAM for all members, we need a space for the biggest member only.

And here is why union is important

2 Union

2.2 Why using union

CODE

```c
#include <stdio.h>
#include <stdlib.h>
union data_base{
   int price;
   struct books{
    char name[12];
    int pages;
    int sales;
   }book;
   struct shirts{
    char color[12];
    int size;
    int model;
   }shirt;
   struct computers{
    char brand[12];
    int ram;
    int hdd;
   }computer;
}souq;
int main()
{
   printf("%d Byte",sizeof(souq));
   return 0;
}
```

OUTPUT

```
20 Byte
```

DESCRIPTION

In this example when we used union instead of struct we save a lot of space in RAM.

3 Enum

3.1 Enum initialization

CODE

```c
#include <stdio.h>
#include <stdlib.h>

enum status{released,pressed};

int main()
{
    int button , led;
    if(button == pressed)
    led = 1;
    return 0;
}
```

DESCRIPTION

Enum give the words a numbers

Here in this example the word released is initialized automatically with 0

The word pressed is initialized automatically with 1

Enum is very important specially with flags

Enum also makes your code more readable

3 Enum

3.2 Enum size

CODE

```c
#include <stdio.h>
#include <stdlib.h>

enum week{saturday,
          sunday,
          monday,
          tusday=22,
          wednsday,
          thursday,
          friday}w_1;

int main()
{
   printf("%d bytes\n",sizeof(w_1));

   return 0;
}
```

OUTPUT

```
4 bytes
```

DESCRIPTION

Size of Enum = size of int

In this case -> size of Enum = 4 Bytes

3 Enum

3.3 Different between Enum and #define

CODE

```c
#include <stdio.h>
#include <stdlib.h>

enum week{saturday,
          sunday,
          monday,
          tusday=22,
          wednsday,
          thursday,
          friday}w_1;

int main()
{
    printf("%d\n",saturday);
    printf("%d\n",sunday);
    printf("%d\n",monday);
    printf("%d\n",tusday);
    printf("%d\n",wednsday);
    printf("%d\n",thursday);
    printf("%d\n",friday);
    return 0;
}
```

OUTPUT

```
0
1
2
22
23
24
25
```

DESCRIPTION

The preprocessor is responsible of # define

The compiler is responsible of Enum

So what is the difference ?

The preprocessor is just make a text replacement, but the compiler is more advanced like this example :

when the compiler found Tuesday = 22

It automatically increment the next element by 1 to become 23.

4 Typedef

4.1 With primitive data types

CODE

```c
#include <stdio.h>
#include <stdlib.h>
typedef unsigned  char      uint8;
typedef signed    char      sint8;
typedef unsigned short int  uint16;
typedef signed   short int  sint16;
typedef unsigned long  int  uint32;
typedef signed   long int   sint32;
typedef float               float32;
typedef double              float64;

int main()
{
   uint32 x = 20;
   printf("%d",x);
   return 0;
}
```

OUTPUT

```
20
```

DESCRIPTION

Typedef is used for renaming

Instead of typing unsigned short int

Now you can just type uint16

4 Typedef

4.2 With user defined data types

CODE

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct student{
        int id;
        char age ;
        float grade;
        }s;

int main()
{
    s ahmed;
    ahmed.age = 20;
    printf("%d",ahmed.age);
    return 0;
}
```

OUTPUT

```
20
```

DESCRIPTION

Using typedef here makes s a new data type **not object**

So, by using typedef you can type

s ahmed directly without typing struct student ahmed

4 Typedef

4.3 Create a new data type

CODE

```c
#include <stdio.h>
#include <stdlib.h>
typedef int* ptr;

int main()
{
   int   x , y , z;
   ptr   i , j , k;

   i=&x;
   j=&y;
   k=&z;
   *i = 5;
   *j = 6;
   *k = 7;
   printf("%d\n",x);
   printf("%d\n",y);
   printf("%d\n",z);

   return 0;
}
```

OUTPUT

```
5
6
7
```

DESCRIPTION

In this example we use typedef to renaming int*

With ptr

Now I and j and k become pointers to integers

And we can access x, y, and z through them

THANK YOU!

AMIT LEARNING

[NEXT SESSION IS -> SESSION_7 : ALGORITHMS]