

Advanced Data types

Data Structure



Objectives

- Data structures introduction
- Define & initialize a structure
- Access elements
- Typedef keyword
- Nested structure
- Assignment Statements
- Structure Padding
- Structures and functions
- Use arrays of structures
- Pointers to structures

Section 1

DATA STRUCTURES INTRODUCTION

Why use structure?

In C, there are cases where we need to store multiple attributes of an entity. It is not necessary that an entity has all the information of one type only. It can have different attributes of different data types. For example, an entity Student may have its **name (string)**, **roll number (int)**, **marks (float)**. To store such type of information regarding an entity student, we have the following approaches:

- +) Construct individual arrays for storing names, roll numbers, and marks.
- +) Use a special data structure to store the collection of different data types.

The first approach in detail:

```
#include <stdio.h>

int main()
{
    char names[2][10],dummy; // 2-dimensioanal character array names is used to store the names of the students
    int roll_numbers[2],i;
    float marks[2];
    for (i=0; i<3; i++)
    {
        printf("Enter the name, roll number, and marks of the student %d",i+1);
        scanf("%s %d %f",&names[i],&roll_numbers[i],&marks[i]);
        scanf("%c",&dummy); // enter will be stored into dummy character at each iteration
    }
    printf("Printing the Student details ...\n");
    for (i=0; i<3; i++)
    {
        printf("%s %d %f\n",names[i],roll_numbers[i],marks[i]);
    }

    return 0;
}
```

Structures approach - 1

Output:

```
Enter the name, roll number, and marks of the student 1Arun 90 91
Enter the name, roll number, and marks of the student 2Varun 91 56
Enter the name, roll number, and marks of the student 3Sham 89 69

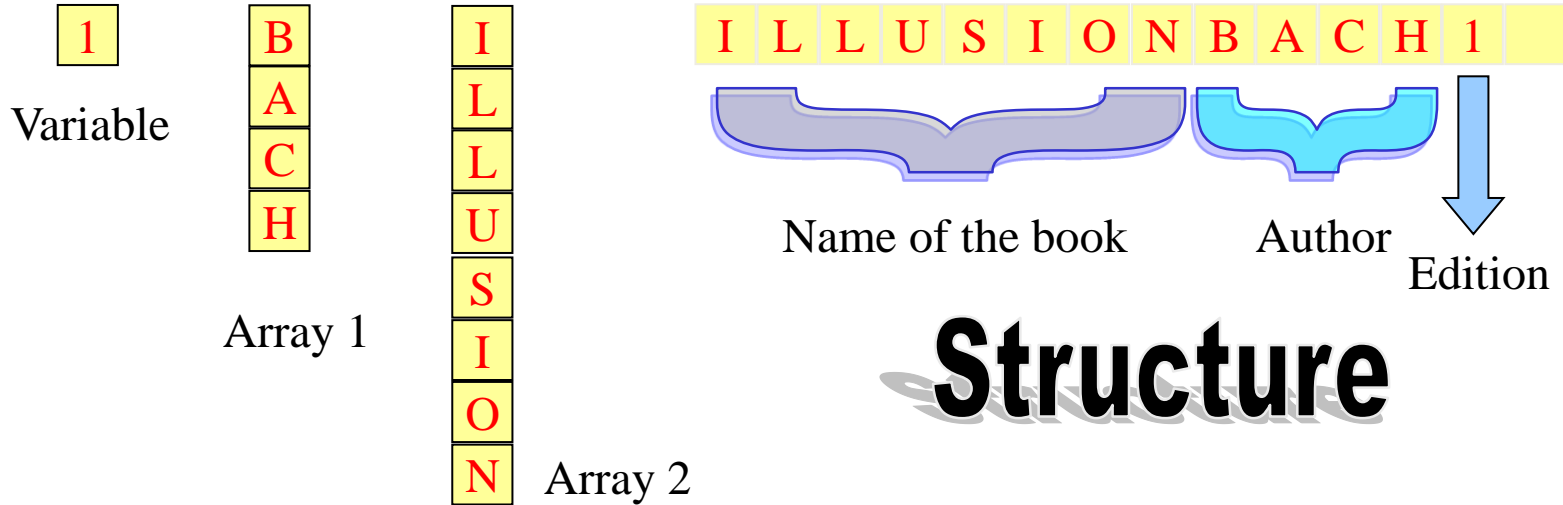
Printing the Student details...
Arun 90 91.000000
Varun 91 56.000000
Sham 89 69.000000
```



The above program may fulfill our requirement of storing the information of an entity student. However, the program is very complex, and the complexity increase with the amount of the input. The elements of each of the array are stored contiguously, but all the arrays may not be stored contiguously in the memory. C provides you with an additional and simpler approach where you can use a special **data structure**, i.e., structure, in which, you can group all the information of different data type regarding an entity.

Structures: Figure Sample

- A structure consists of a number of data items, which need not be of the same data type, grouped together
- The structure could hold as many of these items as desired



Structures: Syntax

- 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.
- The **struct** keyword is used to define the **structure**. Let's see the syntax to define the structure in C:

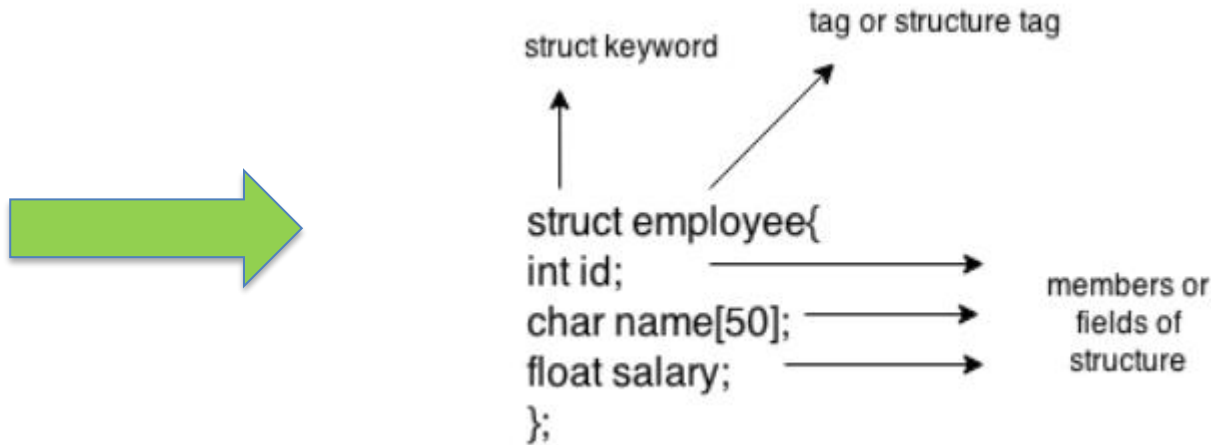
```
struct structure_name  
{  
    data_type member1;  
    data_type member2;  
    .  
    .  
    data_type memberN;  
};
```

Example

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

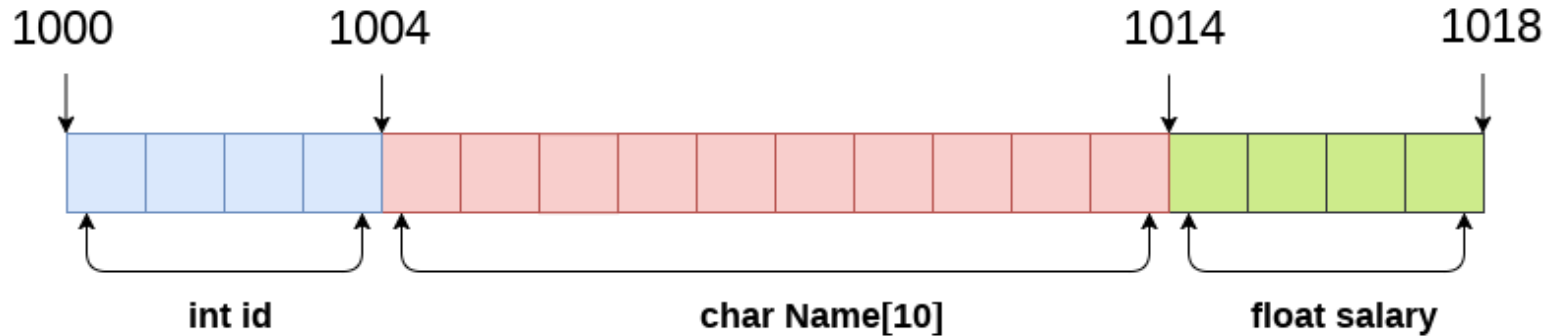

Structures: Syntax - 1

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:



Structures: Memory allocation

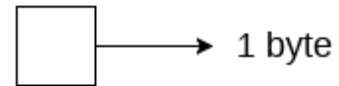
The following image shows the memory allocation of the structure employee that is defined in the above example:



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

`sizeof (emp) = 4 + 10 + 4 = 18 bytes`

where;
`sizeof (int) = 4 byte`
`sizeof (char) = 1 byte`
`sizeof (float) = 4 byte`



Section 2

DEFINE & INITIALIZE A STRUCTURE

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

- By **struct** keyword within **main()** function
- By declaring a variable at the time of defining the structure.

Defining structures - 1

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
#include <stdio.h>
struct employee
{
    int id;
    char name[50];
    float salary;
};
int main()
{
    struct employee e1, e2;
    return 0;
}
```

The variables e1 and e2 can be used to access the values stored in the structure.

Defining structures - 2

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
#include <stdio.h>
struct employee
{
    int id;
    char name[50];
    float salary;
} e1, e2;

int main()
{
    return 0;
}
```

If number of variables are not fixed, use the **1st** approach. It provides you the flexibility to declare the structure variable many times.

If no one of variables are fixed, use **2nd** approach. It saves your code to declare a variable in main() function.

Initializing Structures

- Like variables and arrays, structure variables can be initialized at the point of declaration

```
struct employee
{
    int no;
    char name [20];
};
```

- Variables **emp1** and **emp2** of the type **employee** can be declared and initialized as:

```
struct employee emp1 = {346, "Abraham"};
struct employee emp2 = {347, "John"};
```

Initializing Structures - 1

Example:

```
struct employee
{
    int no;
    char name [20];
};

int main()
{
    struct employee emp1 = {2016, "Trump"};
    struct employee emp2 = {2021, "Biden"};
    return 0;
}
```


Section 3

ACCESS ELEMENTS

There are two ways to access structure members:

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



`structure_variable.structure_member`

`structure_pointer->structure_member`

Access elements: Example

```
#include <stdio.h>
struct Distance {
    int feet;
    float inch;
} dist1, dist2, sum;

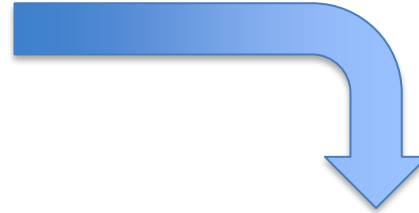
int main() {
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);
    printf("2nd distance\n");

    printf("Enter feet: ");
    scanf("%d", &dist2.feet);
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);

    sum.feet = dist1.feet + dist2.feet;    // adding feet
    sum.inch = dist1.inch + dist2.inch;    // adding inches

    // changing to feet if inch is greater than 12
    while (sum.inch >= 12) {
        ++sum.feet;
        sum.inch = sum.inch - 12;
    }
    printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
    return 0;
}
```

Program to add two distances (feet-inch)



```
1st distance
Enter feet: 2
Enter inch: 3
2nd distance
Enter feet: 4
Enter inch: 5
Sum of distances = 6'-8.0"
```

Section 4

TYPDEF KEYWORD

The **typedef** keyword

- A new data type name can be defined by using the keyword **typedef**
- It does not create a new data type, but defines a new name for an existing type
- **Syntax:**

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

- **Example:**

```
typedef float decimal;
```

- typedef cannot be used with storage classes

The **typedef** keyword: Example

```
int main()
{
    typedef float decimal;
    decimal i,j;
    i = 20;
    j = 21;
    printf("Value of i is :%f",i);
    printf("\nValue of j is :%f",j);
    return 0;
}
```

Define a new alias
for data type: **float**

```
Value of i is :20.000000
Value of j is :21.000000
```



Using typedef with structures

Consider the below structure declaration:

```
struct student
{
    char name[20];
    int age;
};

struct student s1;
```

In the above structure declaration, we have created the variable of student type by writing the following s `struct student s1;`

Using typedef with structures - 1

The above statement shows the creation of a variable, i.e., s1, but the statement is quite big. To avoid such a big statement, we use the **typedef** keyword to create the variable of type **student**.

```
struct student
{
    char name[20];
    int age;
};

typedef struct student stud;

stud s1, s2;
```

In the above statement, we have declared the variable **stud** of type **struct student**. Now, we can use the **stud** variable in a program to create the variables of type **struct student**.

Using typedef with structures - 2

The above typedef can be written as:

```
typedef struct student
{
    char name[20];
    int age;
} stud;
stud s1,s2;
```

From the above declarations, we conclude that **typedef** keyword reduces the length of the code and complexity of data types. It also helps in understanding the program.

Using typedef with structures - 3

```
#include <stdio.h>
typedef struct student
{
    char name[20];
    int age;
}stud;
int main()
{
    stud s1;
    printf("Enter the details of student s1: ");
    printf("\nEnter the name of the student:");
    scanf("%s",&s1.name);
    printf("\nEnter the age of student:");
    scanf("%d",&s1.age);
    printf("\n Name of the student is : %s", s1.name);
    printf("\n Age of the student is : %d", s1.age);
    return 0;
}
```



```
Enter the details of student s1:
Enter the name of the student: Peter
Enter the age of student: 28
Name of the student is : Peter
Age of the student is : 28
```

Section 5

NESTED STRUCTURE

Nested structure: Separate

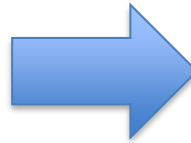
- It is possible to have one structure within another structure. A structure cannot be nested within itself. Look at example below, `cat_books` is a member of `issue`:

```
struct issue
{
    char borrower[20];
    char dt_of_issue[21];
    struct cat_books;
}iss1;
```

- Nested structure is also called structures within structures

Nested structure: Embedded

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures



```
struct Employee
{
    int id;
    char name[20];
    struct Date
    {
        int dd;
        int mm;
        int yyyy;
    }doj;
}emp1;
```

Nested structure: Member access

You can create structures within a structure in C programming. For example:

```
struct complex
{
    int imag;
    float real;
};

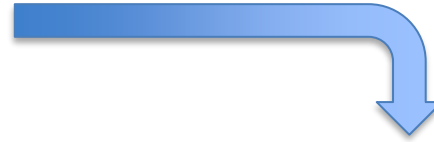
struct number
{
    struct complex comp;
    int integers;
} num1, num2;
```

Suppose, you want to set **imag** of **num2** variable to **11**. Here's how you can do it:

```
num2.comp.imag = 11;
```

Nested structure: Example

```
#include <stdio.h>
struct complex
{
    int imag;
    float real;
};
struct number
{
    struct complex comp;
    int integers;
} num1, num2;
int main()
{
    num2.comp.imag = 123;
    num2.integers = 321;
    printf("Value of integers is :%d", num2.integers);
    printf("\nValue of imag is :%d", num2.comp.imag);
    return 0;
}
```



```
Value of integers is :321
Value of imag is :123

...Program finished with exit code 0
Press ENTER to exit console.□
```

Section 6

ASSIGNMENT STATEMENTS

- It is possible to assign the values of one structure variable to another variable of the same type using a simple assignment statement
- For example, if **books1** and **books2** are structure variables of the same type, the following statement is valid:

books2 = books1;

- In cases where direct assignment is not possible, the in-built function **memcpy()** can be used
- Syntax:
memcpy (char * destn, char &source, int nbytes);
- Example:
memcpy (&books2, &books1, sizeof(struct cat));

Example with Assignment Operator:

```
#include <stdio.h>

struct student
{
    int id;
    char name[60];
}st1,st2; //declaring st1 and st2 variables for structure

int main()
{
    printf("Enter id of student: ");
    scanf("%d", &st1.id);
    printf("Enter name of student: ");
    scanf("%s", st1.name);
    st2 = st1;
    printf("\nId of employee is : %d", st2.id);
    printf("\nName of employee is : %s", st2.name);
    return 0;
}
```

OUTPUT SCREEN:

```
Enter id of student: 10
Enter name of student: Mickey

Id of employee is : 10
Name of employee is : Mickey

...Program finished with exit code 0
Press ENTER to exit console.
```

Example with **memcpy()** function:

```
#include <stdio.h>

struct student
{
    int id;
    char name[60];
}st1,st2; //declaring st1 and st2 variables for structure

int main()
{
    printf("Enter id of student: ");
    scanf("%d", &st1.id);
    printf("Enter name of student: ");
    scanf("%s", st1.name);
    memcpy(&st2, &st1, sizeof(struct student));
    printf("\nId of employee is : %d", st2.id);
    printf("\nName of employee is : %s", st2.name);
    return 0;
}
```

OUTPUT SCREEN:

```
Enter id of student: 10
Enter name of student: Mickey

Id of employee is : 10
Name of employee is : Mickey

...Program finished with exit code 0
Press ENTER to exit console.
```

Section 7

STRUCTURE PADDING IN C

Structure Padding

- Structure padding is a concept in C that adds the one or more empty bytes between the memory addresses to align the data in memory.
- Let's first understand the structure padding in C through a simple scenario which is given below:
- Suppose we create a user-defined structure. When we create an object of this structure, then the contiguous memory will be allocated to the structure members.

```
struct student  
{  
    char a;  
    char b;  
    int c;  
} stud1;
```

Structure Padding - 1

In the above example, we have created a structure of type **student**. We have declared the object of this structure named as '**stud1**'. After the creation of an object, a contiguous block of memory is allocated to its structure members. First, the memory will be allocated to the '**a**' variable, then '**b**' variable, and then '**c**' variable.

Now, we calculate the size of the **struct student**. We assume that the size of the int is 4 bytes, and the size of the char is 1 byte.



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

In the above case, when we calculate the size of the struct student, size comes to be 6 bytes. But this answer is wrong. Now, we will understand why this answer is wrong? We need to understand the concept of structure padding.

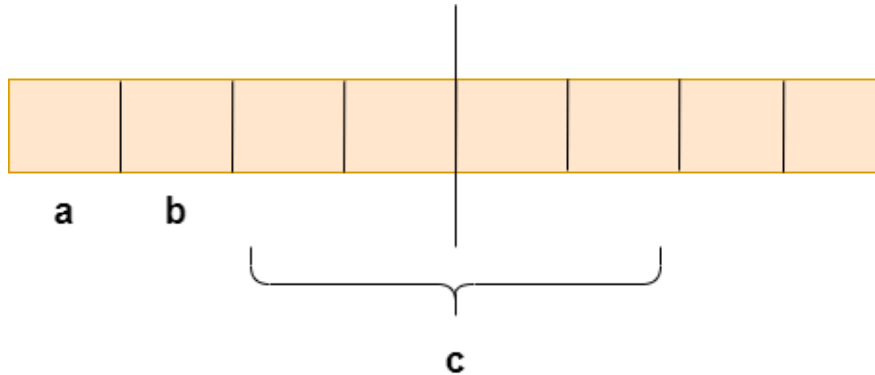
Structure Padding - 2

- The processor does not read 1 byte at a time. It reads 1 word at a time.
- If we have a 32-bit processor, then the processor reads 4 bytes at a time, which means that 1 word is equal to 4 bytes.
- If we have a 64-bit processor, then the processor reads 8 bytes at a time, which means that 1 word is equal to 8 bytes.
- Therefore, we can say that a 32-bit processor is capable of accessing 4 bytes at a time, whereas a 64-bit processor is capable of accessing 8 bytes at a time. It depends upon the architecture that what would be the size of the word.

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


Structure Padding - 3

If we have a 32-bit processor (4 bytes at a time), then the pictorial representation of the memory for the above structure would be:



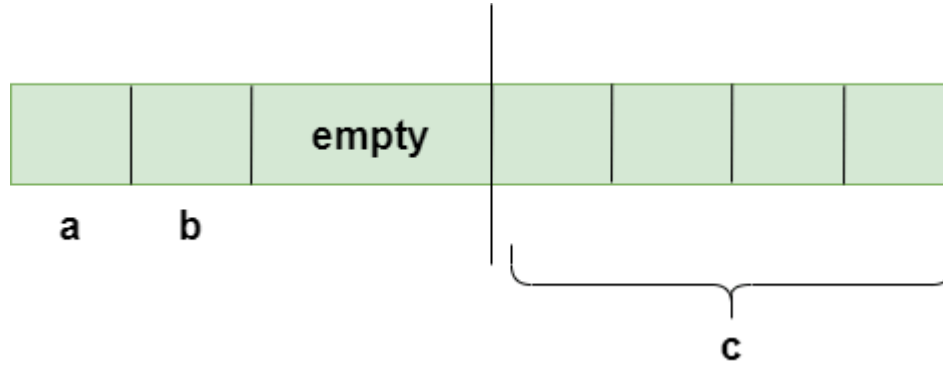
As we know that structure occupies the contiguous block of memory as shown in the above diagram, i.e., 1 byte for char a, 1 byte for char b, and 4 bytes for int c, then what problem do we face in this case.

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

Structure Padding - 4

- The 4-bytes can be accessed at a time as we are considering the 32-bit architecture. The problem is that in one CPU cycle, one byte of **char a**, one byte of **char b**, and 2 bytes of **int c** can be accessed. We will not face any problem while accessing the **char a** and **char b** as both the variables can be accessed in one CPU cycle, but we will face the problem when we access the **int c** variable as 2 CPU cycles are required to access the value of the 'c' variable. In the first CPU cycle, the first two bytes are accessed, and in the second cycle, the other two bytes are accessed.
- Suppose we do not want to access the 'a' and 'b' variable, we only want to access the variable 'c', which requires two cycles. The variable 'c' is of 4 bytes, so it can be accessed in one cycle also, but in this scenario, it is utilizing 2 cycles. This is an unnecessary wastage of CPU cycles. Due to this reason, the structure padding concept was introduced to save the number of CPU cycles. The structure padding is done automatically by the compiler. Now, we will see how structure padding is done.


Structure Padding - 5



In order to achieve the structure padding, an empty row is created on the left, as shown in the above diagram, and the two bytes which are occupied by the 'c' variable on the left are shifted to the right. So, all the four bytes of 'c' variable are on the right. Now, the 'c' variable can be accessed in a single CPU cycle. After structure padding, the total memory occupied by the structure is 8 bytes (1 byte+1 byte+2 bytes+4 bytes), which is greater than the previous one. Although the memory is wasted in this case, the variable can be accessed within a single cycle.

Structure Padding - 6

```
#include <stdio.h>
struct student
{
    char a;
    char b;
    int c;
};
int main()
{
    struct student stud1; // variable declaration of the student type..
    // Displaying the size of the structure student.
    printf("The size of the student structure is %d", sizeof(stud1));
    return 0;
}
```




```
The size of the student structure is 8
...Program finished with exit code 0
Press ENTER to exit console.□
```

In the example code, we have created a structure named '**student**'. Inside the **main()** method, we declare a variable of student type, i.e., **stud1**, and then we calculate the size of the student by using the **sizeof()** operator. The output would be **8 bytes** due to the concept of structure padding, which we have already discussed in the above.

Structure Padding - 7

Changing order of the variables

```
#include <stdio.h>
struct student
{
    char a;
    int b;
    char c;
};
int main()
{
    struct student stud1; // variable declaration of the student type..
    // Displaying the size of the structure student.
    printf("The size of the student structure is %d", sizeof(stud1));
    return 0;
}
```



```
The size of the student structure is 12

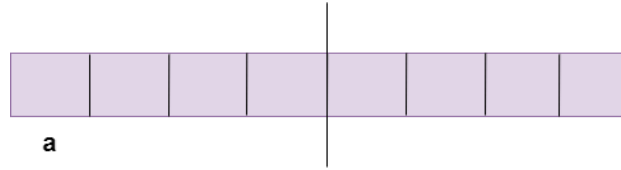
...Program finished with exit code 0
Press ENTER to exit console.
```

The above code is similar to the previous code; the only thing we change is the order of the variables inside the **structure student**. Due to the change in the order, the output would be different in both the cases. In the previous case, the output was 8 bytes, but in this case, the output is 12 bytes, as we can observe in the below screenshot.

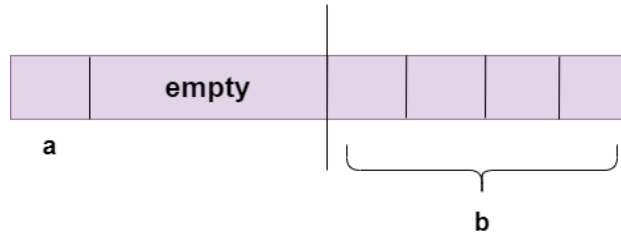
Structure Padding - 8

Explain the output is different in this case:

- First, memory is allocated to the **char a** variable, i.e., 1 byte.

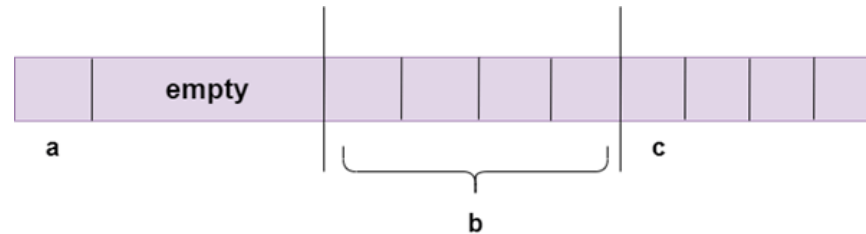


- Now, the memory will be allocated to the **int b**. Since the **int** variable occupies 4 bytes, but on the left, only 3 bytes are available. The empty row will be created on these 3 bytes, and the int variable would occupy the other 4 bytes so that the integer variable can be accessed in a single CPU cycle.



Structure Padding - 9

- And now, the memory will be given to the **char c**. At a time, CPU can access 1 word, which is equal to 4 bytes, so CPU will use 4 bytes to access a 'c' variable. Therefore, the total memory required is 12 bytes (4 bytes + 4 bytes + 4 bytes), i.e., 4 bytes required to access **char a** variable, 4 bytes required to access **int b** variable, and other 4 bytes required to access a single character of 'c' variable.



Structure Padding: Avoid padding

- The structural padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.
- We can avoid the structure padding in C in two ways:
 - **Using #pragma pack(1) directive**
 - **Using attribute**

Structure Padding: Avoid padding - 1

Using `#pragma pack(1)` directive:

```
#include <stdio.h>
#pragma pack(1)
struct base
{
    int a;
    char b;
    double c;
};
int main()
{
    struct base var; // variable declaration of type base
    // Displaying the size of the structure base
    printf("The size of the var is : %d", sizeof(var));
    return 0;
}
```



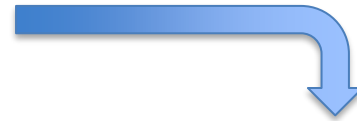
```
The size of the var is : 13
...Program finished with exit code 0
Press ENTER to exit console.□
```

In this code, we have used the **#pragma pack(1)** directive to avoid the structure padding. If we do not use this directive, then the output of the above program would be **16** bytes. But the actual size of the structure members is **13** bytes, so **3** bytes are wasted. To avoid the wastage of memory, we use the **#pragma pack(1)** directive to provide the **1-byte** packaging.

Structure Padding: Avoid padding - 2

Using attribute:

```
#include <stdio.h>
struct base
{
    int a;
    char b;
    double c;
}__attribute__((packed));
int main()
{
    struct base var; // variable declaration of type base
    // Displaying the size of the structure base
    printf("The size of the var is : %d", sizeof(var));
    return 0;
}
```



```
The size of the var is : 13

...Program finished with exit code 0
Press ENTER to exit console.
```

Section 8

BIT FIELDS

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follow:

```
struct {  
    unsigned int widthValidated;  
    unsigned int heightValidated;  
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

Bit Fields - 1

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows:

```
struct {  
    unsigned int widthValidated : 1;  
    unsigned int heightValidated : 1;  
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

Bit Fields - 2

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept:

```
#include <stdio.h>
#include <string.h>
/* define simple structure */
struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;
/* define a structure with bit fields */
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;
int main( ) {
    printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
    return 0;
}
```



```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Fields: Declaration

The declaration of a bit-field has the following form inside a structure:

```
struct {  
    type [member_name] : width ;  
};
```

The following table describes the variable elements of a bit field:

<u>Element</u>	<u>Description</u>
type	An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
member_name	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Bit Fields: Declaration - 1

- The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows:

```
struct {  
    unsigned int age : 3;  
} Age;
```

- The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Look at the following example below.

Bit Fields: Declaration - 2

```
#include <stdio.h>
#include <string.h>
struct {
    unsigned int age : 3;
} Age;
int main( ) {
    Age.age = 4;
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
    printf( "Age.age : %d\n", Age.age );
    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );
    Age.age = 8;
    printf( "Age.age : %d\n", Age.age );
    return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result:



```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Section 9

STRUCTURE AND FUNCTION

Passing Structures as Arguments

- A structure variable can be passed as an argument to a function
- This facility is used to pass groups of logically related data items together instead of passing them one by one
- The type of the argument should match the type of the parameter

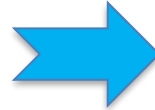
Passing Structures as Arguments - 1

```
#include <stdio.h>
struct student {
    char name[55];
    int age;
};

// function prototype
void display(struct student s);

int main() {
    struct student s1;
    printf("Enter name: ");
    scanf("%s", s1.name);
    printf("Enter age: ");
    scanf("%d", &s1.age);
    display(s1); // passing struct as an argument
    return 0;
}

void display(struct student s) {
    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nAge: %d", s.age);
}
```



```
Enter name: Tom
Enter age: 10

Displaying information
Name: Tom
Age: 10

...Program finished with exit code 0
Press ENTER to exit console.
```

a **struct** variable **s1** of type **struct student** is created. The variable is passed to the **display()** function using **display(s1);** statement.

Return struct from a function

Here's how you can return structure from a function:

```
#include <stdio.h>
struct student {
    char name[50];
    int age;
};

// function prototype
struct student getInformation();

int main() {
    struct student s;
    s = getInformation();
    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nRoll: %d", s.age);
    return 0;
}

struct student getInformation() {
    struct student s1;
    printf("Enter name: ");
    scanf ("%s", s1.name);
    printf("Enter age: ");
    scanf ("%d", &s1.age);
    return s1;
}
```

```
Enter name: Jerry
Enter age: 10

Displaying information
Name: Jerry
Roll: 10

...Program finished with exit code 0
Press ENTER to exit console.
```

output

Passing struct by reference

During pass **struct** by reference, the memory addresses of **struct** variables are passed to the function.

For first number,
Enter real part: 1.1
Enter imaginary part: -2.4
For second number,
Enter real part: 3.4
Enter imaginary part: -3.2

result.real = 4.5
result.imag = -5.6

```
#include <stdio.h>
typedef struct Complex {
    float real;
    float imag;
} complex;
void addNumbers(complex c1, complex c2, complex *result);
int main() {
    complex c1, c2, result;
    printf("For first number,\n");
    printf("Enter real part: ");
    scanf("%f", &c1.real);
    printf("Enter imaginary part: ");
    scanf("%f", &c1.imag);
    printf("For second number, \n");
    printf("Enter real part: ");
    scanf("%f", &c2.real);
    printf("Enter imaginary part: ");
    scanf("%f", &c2.imag);
    addNumbers(c1, c2, &result);
    printf("\nresult.real = %.1f\n", result.real);
    printf("result.imag = %.1f", result.imag);
    return 0;
}

void addNumbers(complex c1, complex c2, complex *result) {
    result->real = c1.real + c2.real;
    result->imag = c1.imag + c2.imag;
}
```

Three structure variables **c1**, **c2** and the address of **result** is passed to the **addNumbers()** function. Here, **result** is passed by reference. When the **result** variable inside the **addNumbers()** is altered, the result variable inside the **main()** function is also altered accordingly.

Section 10

STRUCTURE AND ARRAY

- A common use of structures is in arrays of structures
- A structure is first defined, and then an array variable of that type is declared

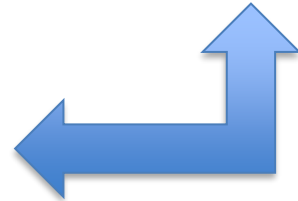
- Example:

struct cat books[50];

- To access the variable author of the fourth element of the array **books**:

books[4].author

```
struct cat {  
    char author[50];  
    int age;  
};
```



Initialization of Structure Arrays

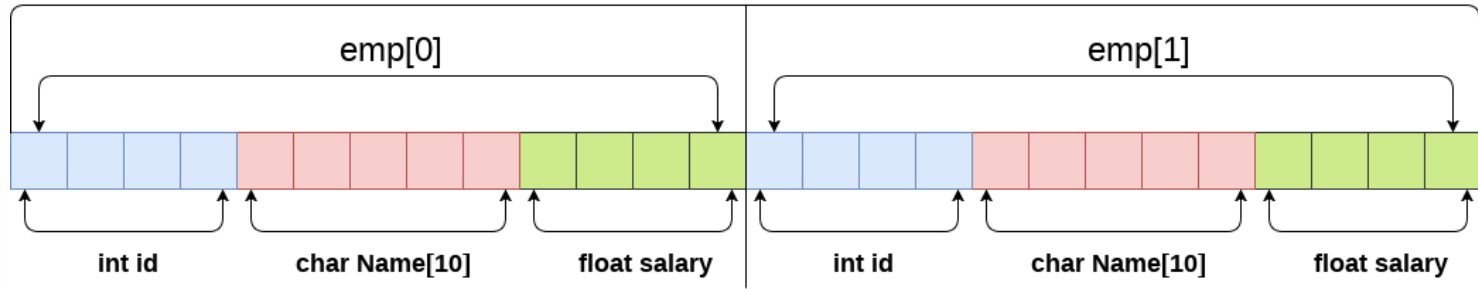
- Structure arrays are initialized by enclosing the list of values of its elements within a pair of braces.
- Example:
Initialization array of **struct unit** include 3 elements.

```
struct unit
{
    char ch;
    int i;
};
struct unit series [3] =
{
    {'a', 100}
    {'b', 200}
    {'c', 300}
};
```

Array of Structures: Store data

Array of structures

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types.



```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

`sizeof (emp) = 4 + 5 + 4 = 13 bytes`

`sizeof (emp[2]) = 26 bytes`

Array of Structures: Example

```
#include<stdio.h>
struct student {
    int rollno;
    char name[10];
};
int main() {
    int i;
    struct student st[5];
    printf("Enter Records of 5 students");
    for(i=0;i<5;i++) {
        printf("\nEnter Rollno:");
        scanf("%d",&st[i].rollno);
        printf("\nEnter Name:");
        scanf("%s",st[i].name);
    }
    printf("\nStudent Information List:");
    for(i=0;i<5;i++) {
        printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
    }
    return 0;
}
```



Enter Records of 5 students

Enter Rollno:1

Enter Name:Sonoo

Enter Rollno:2

Enter Name:Ratan

Enter Rollno:3

Enter Name:Vimal

Enter Rollno:4

Enter Name:James

Enter Rollno:5

Enter Name:Sarfraz

Student Information List:

Rollno:1, Name:Sonoo

Rollno:2, Name:Ratan

Rollno:3, Name:Vimal

Rollno:4, Name:James

Rollno:5, Name:Sarfraz

Section 11

POINTERS TO STRUCTURES

Pointers to Structures

Here's how you can create pointers to structs: **ptr** is a **pointer** to **struct**

```
struct name {  
    member1;  
    member2;  
    .  
    .  
};  
  
int main()  
{  
    struct name *ptr, Harry;  
}
```

Pointers to Structures - 1

- Structure pointers are declared by placing an asterisk(*) in front of the structure variable's name
- The -> operator is used to access the elements of a structure using a pointer
- **Example:**

```
struct cat *ptr_bk;  
ptr_bk = &books;  
printf("%s", ptr_bk->author);
```

- Structure pointers passed as arguments to functions enable the function to modify the structure elements directly

Access members using Pointer

To access members of a structure using pointers, use the **->** operator.

```
#include <stdio.h>
struct person
{
    int age;
    float weight;
};

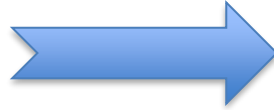
int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);

    return 0;
}
```



```
Enter age: 50
Enter weight: 60
Displaying:
Age: 50
weight: 60.000000
```

In this example, the address of **person1** is stored in the **personPtr** pointer using **personPtr = &person1;**. Now, you can access the members of **person1** using the **personPtr** pointer. By the way:

- +) **personPtr->age** is equivalent to **(*personPtr).age**
- +) **personPtr->weight** is equivalent to **(*personPtr).weight**

Dynamic memory allocation of structs

Sometimes, the number of struct variables you declared may be insufficient. You may need to allocate memory during run-time. Here's how you can achieve this in C programming.

```
Enter the number of persons: 2
Enter first name and age respectively: Harry 24
Enter first name and age respectively: Gary 32
Displaying Information:
Name: Harry      Age: 24
Name: Gary       Age: 32
```

```
#include <stdio.h>
#include <stdlib.h>
struct person {
    int age;
    float weight;
    char name[30];
};

int main() {
    struct person *ptr;
    int i, n;
    printf("Enter the number of persons: ");
    scanf("%d", &n);
    // allocating memory for n numbers of struct person
    ptr = (struct person*) malloc(n * sizeof(struct person));
    for(i = 0; i < n; ++i) {
        printf("Enter first name and age respectively: ");
        // To access members of 1st struct person,
        // ptr->name and ptr->age is used
        // To access members of 2nd struct person,
        // (ptr+1)->name and (ptr+1)->age is used
        scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
    }
    printf("Displaying Information:\n");
    for(i = 0; i < n; ++i)
        printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);
    return 0;
}
```


Dynamic memory allocation of structs - 1

In the above example, **n** number of struct variables are created where **n** is entered by the user.

To allocate the memory for **n** number of **struct person**, we used:

```
ptr = (struct person*) malloc(n * sizeof(struct person));
```

Then, we used the **ptr** pointer to access elements of **person**.

Thank you

Q&A

