STRUCTURE AND UNION

Structures

Definition and declaration of Structures

Structure is a collection of different data types. A single structure might contain integer elements, floating point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members. In general, a structure may be defined as

```
struct tag{
        data_type member 1;
        data_type member 2;
        ……
        data_type member n;
        };
```

For example,

```
struct book_bank
{
     char       title[20];
     char       author[15];
     int              pages;
     float           price;
};
```

The keyword **struct** declares a structure to hold the details of four data fields, namely, title, author, pages and price. These fields are called structure elements or members. Each member may belong to a different type of data.

Declaring Structure Variables

A structure variable declaration includes the following elements

1. The keyword **struct**
2. The structure tag name
3. List of variable names separated by commas
4. A terminating semicolon

For example, the statement

```
     struct book_bank book1, book2, book3;
```

The above statement declares book1, book2 and book3 as variables of type struct book bank.

The complete declaration is given by

```
struct book_bank
{
      char       title[20];
      char       author[15];
      int               pages;
      float            price;
};
struct book_bank book1, book2, book3;   ---→ variable declaration
```

Structure definition and variable declaration can be combined to one statement

```
struct book_bank
{
      char       title[20];
      char       author[15];
      int               pages;
      float            price;
} book1, book2, book3;
```

The use of tag name is optional.

```
struct
{
      char       title[20];
      char       author[15];
      int               pages;
      float            price;
} book1, book2, book3;
```

Arrays Vs Structures

Both the arrays and structures are classified as structured data types. But they differ in a number of ways.

1. An array is a collection of related data elements of the same type. Structure can have elements of different types.
2. An array is a derived data type whereas a structure is a programmer-defined one.
3. An array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

Initialization of structures

We cannot initialize individual members inside the structure template. The order of values enclosed in braces must match the order of members in the structure definition. It is permitted to have a partial initialization. We can initialize the first few members and leave the remaining blank. The uninitialized members will be assigned default values.

Examples

1)
```
main()
{
  struct
 {
   int weight;
     float height;
    } student = {60, 180.75};
   .............
}
```

2)
```
main()
{
   struct st_record
   {
       int weight;
       float height;
    };
   struct st_record student1 = {60, 180.75};
   struct st_record student2 = {53, 170.60};
 ............
..............
}
```

3)
```
   struct st_record
   {
      int weight;
      float height;
     } student1 = {60, 180.75};

    main()
    {
          struct st_record student2 = {53, 170.60};
           .........................
           ..........................
    }
```

4) student1 and student2 belong to the same structure,
```
student1=student2;
```

Accessing Structure Members

The link between a member and a variable is established using the member operator which is also known as 'dot operator'. For example,
book1.price can be used to access the member price in the structure book1.

```
strcpy(book1.title, “BASIC”);
strcpy(book1.author, “Balagurusamy”);
book1.pages = 250;
book1.price = 120.50;
```

Arrays of Structures

Structures can be used to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, a template can be used to describe student name and marks obtained in various subjects, then declare all the students as structure variables.
In such cases, we can declare an array of structures.

defines an array called student, that consist of 100 elements. Each element is defined to be of the type struct class.
Consider the following declaration

```
struct student{
int sub1mark;
int sub2mark;
int sub3mark;
int student_total;
};

struct student{
 int submark[3];
int student_total;
};



main()
{
struct student st[3] = {{45, 68, 81,0}, {75,53,69,0}, {57, 36, 71,0}};
…..
struct student subject_total={0,0,0,0};
……
}
i=1; i<=3;
i=0; i<3
student[i].sub1mark;
```

This declares the student as an array of three structures student[0], student[1] and student[2] and the members can be initialized as

```
student[0]. subject1= 45;
student[0].subject2 = 65;
…….
student[2].subject3 = 71;
```

UNIONS

Unions, like structures, contain members whose individual data types differs from one another. However, there is a major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union share the same location. Hence, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

```
 union item
{
   int m;
   float x;
   char c;
} code;
```

This declares a variable code of type union item. The union contains three members, each with a different data type. However, only one of them can be used at a time. This is because only one location is allocated for a union variable, irrespective of its size.

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the above example, member x requires 4 bytes which is the largest among the members.

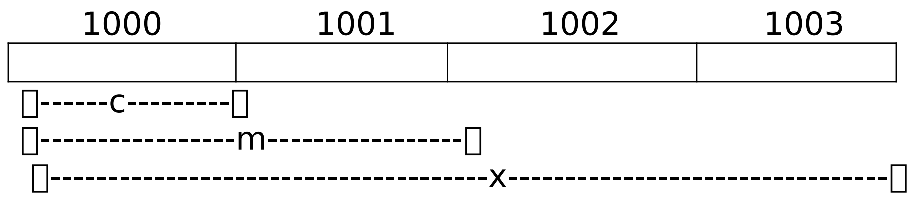
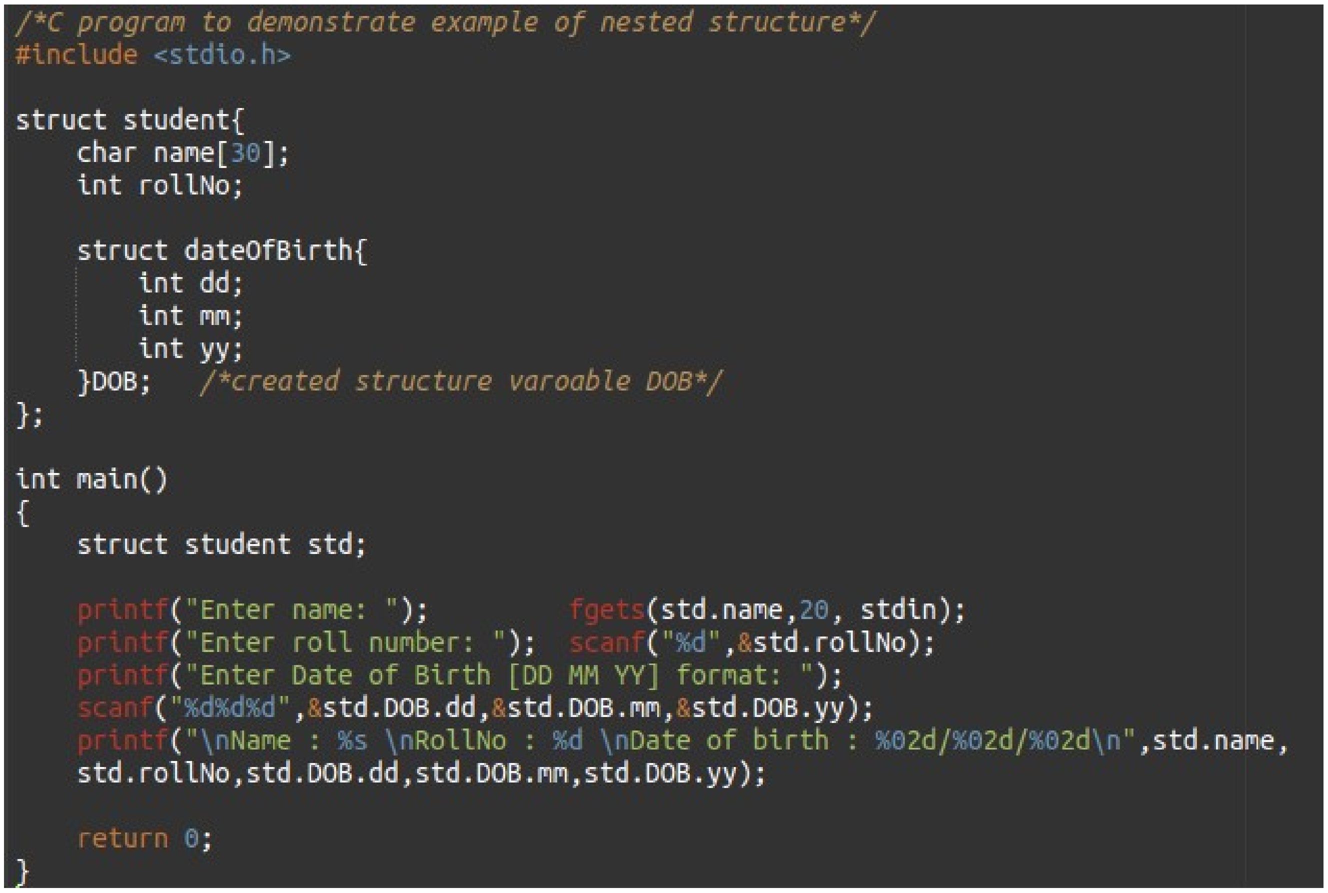


Figure shows that all the three variables share the same address.
To access a union member, we can use the same syntax as for structure members, that is,
code.m
code.x
code.c

During accessing, we should make sure that we are accessing the member whose value is currently stored. Hence, a union creates a storage location that can be used by any of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's value.

Nested Structures



```c
/*C program to demonstrate example of nested structure*/
#include <stdio.h>

struct student{
    char name[30];
    int rollNo;

    struct dateOfBirth{
        int dd;
        int mm;
        int yy;
    }DOB;   /*created structure varoable DOB*/
};

int main()
{
    struct student std;

    printf("Enter name: ");          fgets(std.name,20, stdin);
    printf("Enter roll number: ");  scanf("%d",&std.rollNo);
    printf("Enter Date of Birth [DD MM YY] format: ");
    scanf("%d%d%d",&std.DOB.dd,&std.DOB.mm,&std.DOB.yy);
    printf("\nName : %s \nRollNo : %d \nDate of birth : %02d/%02d/%02d\n",std.name,
    std.rollNo,std.DOB.dd,std.DOB.mm,std.DOB.yy);

    return 0;
}
```

Structure padding

```
#include <stdio.h>
struct student
{
   char a; //1 byte
   char b; //1 byte
   int c; // 4 bytes
};
int main()
{
   struct student stud1; // variable declaration of the student type..
   // Displaying the size of the structure student.
   printf("the size of int is %ld\n", sizeof(int));
   printf("the size of char is %ld\n", sizeof(char));
   printf("The size of the student structure is %ld\n", sizeof(stud1));
   return 0;
}
```

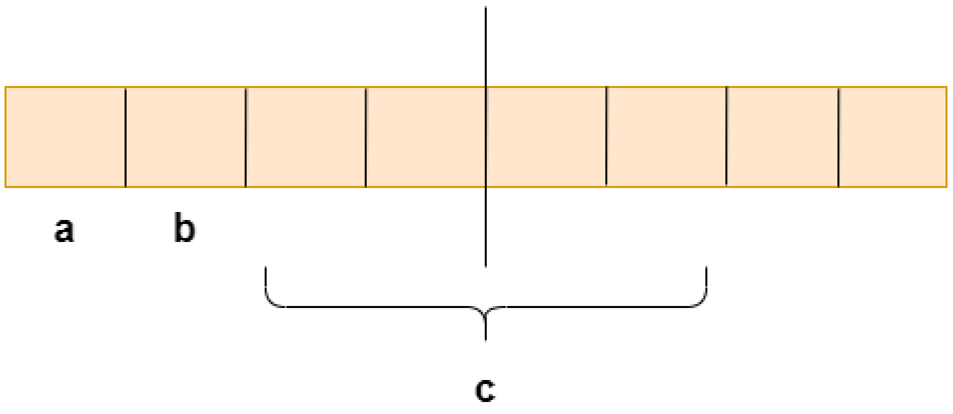
Output:

```
the size of int is 4
the size of char is 1
The size of the student structure is 8
```

32-bit processor ----→ reads 4 bytes at a time, which means that 1 word is equal to 4 bytes.

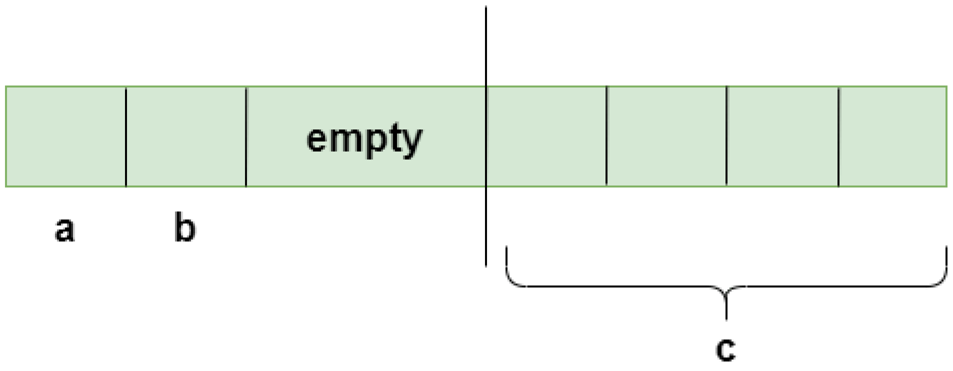
64-bit processor ----→ reads 8 bytes at a time, which means that 1 word is equal to 8 bytes.

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.



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 but in this scenario, it is utilizing 2 cycles. This is an unnecessary wastage of CPU cycles.

The structure padding concept was introduced to save the number of CPU cycles. The structure padding is done automatically by the compiler.



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.

We can avoid the structure padding in C in two ways:

- **Using #pragma pack(1) directive**
- **Using attribute**

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;
    }
```

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;
    }
```