

Structure, Union and Enum

By

Dr. Jyoti Ranjan Nayak

Asst. Prof.

Institute of Technical Education and Research

(Siksha 'O' Anusandhan University)

Structures

- Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.
- The *struct* keyword is used to define the structure in the C++ programming language.

Example:- An entity Student may have its name (string), roll number (int), marks (float). To store such type of information regarding an entity student.

Features of structure

- i. It is possible to copy the contents of all structural elements of different data types to another structure variable of its type by using an assignment operator.
- ii. To handle complex datatypes, it is possible to create a structure within another structure, which is called nested structures.
- iii. It is possible to pass an entire structure, individual elements of structure, and address of structure to a function.
- iv. It is possible to create structure pointers.

Declaration:-

In structure declaration, we specify its member variables along with their datatype.

Syntax:

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

➤ tag_name is optional

Example

```
struct student  
{  
    int studentID;  
    char name[50];  
    float grade;  
};
```

Definition:-

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using three methods:

Syntax-1:

```
struct tag_name  
{  
    data_type member1;  
    data_type member2;  
    .  
    .  
    data_type memeberN;  
}struct_var;
```

Syntax-2:

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

Syntax-3:

```
struct tag_name  
{  
    data_type member1;  
    data_type member2;  
    .  
    .  
    data_type memeberN;  
};  
  
struct tag_name struct_var;
```

Keyword
struct

→ struct myStruct ←

Structure tag
or structure
name

{

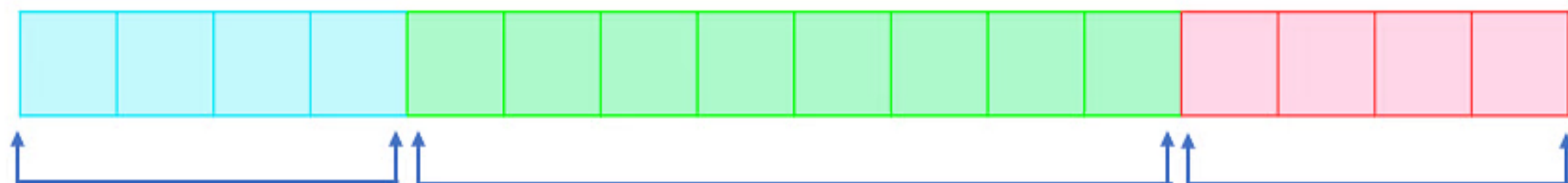
int var1;

char var2[8];

float var3;

}struct var;

Structure
members



int var1

char var2[8]

float var3

Initialize a structure variable:

C++ language supports multiple ways to initialize a structure variable. You can use any of the initialization method to initialize your structure.

Syntax of Initialization using dot operator:

```
struct_var . member1 = value;
```

```
struct_var . member2 = value;
```

In C++, we initialize or access a structure variable either through dot . or arrow -> operator.

Example:

```
struct student
{
    char name[100];
    int roll;
    float marks;
}s1;
s1.name = "Jrn";
s1.roll = 12;
s1.marks = 65.50;
```

Syntax of Value initialization structure variable:

C++ language also supports value initialization for structure variable. Means, you can initialize a structure to some default value during its variable declaration.

struct tag_name struct_var = {member1 value, member2 value, . . . , memberN value};

Example:

```
struct student
{
    char name[100];
    int  roll;
    float marks;
}s1, s2;

struct student s1 = {"Jrn", 12, 65.50};
struct student s2 = {"Pankaj", 13, 88.12};
```

Accessing Structure Members:

- To access the members of a structure variable, we use the dot (.) operator. The dot operator is used to specify the name of the structure member we want to access.

Syntax:

structure_name.member1;

strcuture_name.member2;


```
#include<iostream>
#include<string>
using namespace std;
struct student
{
    string name;
    int roll;
    float mark;
};
int main()
{
    struct student s;
    cout<<"Enter student roll"<<endl;
    cin>>s.roll;
    cout<<"Enter student mark"<<endl;
    cin>>s.mark;
    cout<<"Enter student name"<<endl;
    cin>>s.name;
    cout<<s.name<<"\t"<<s.roll<<"\t"<<s.mark;
    return 0;
}
```

```
Enter student roll
15
Enter student mark
85.25
Enter student name
JRN
JRN  15  85.25
```

```

#include<iostream>
#include<string>
using namespace std;
struct student
{
    string name;
    int roll;
    float mark;
}s[2];
int main()
{
    int i;
    for (i=0;i<2;i++)
    {
        s[i].roll=i+1;
        cout<<"Enter student name"<<endl;
        cin>>s[i].name;
        cout<<"Enter student mark"<<endl;
        cin>>s[i].mark;
    }
}

```

```

for (i=0;i<2;i++)
{
    cout<<"Details of student having Roll:"<<s[i].roll<<endl;
    cout<<"student name"<<s[i].name<<endl;
    cout<<"student mark"<<s[i].mark<<endl;
}
return 0;
}

```

Enter student name

JRN

Enter student mark

85.23

Enter student name

SD

Enter student mark

87.58

Details of student having Roll:1

student nameJRN

student mark85.23

Details of student having Roll:2

student nameSD

student mark87.58

Accessing Structure Members through Pointers:

In C++, we can also use pointers to access structure members. To access structure members using pointers, we use the arrow (->) operator.

```
#include<iostream>
#include<string>
using namespace std;
struct student
{
    string name;
    int roll;
    float mark;
}s;
```

```
int main()
{
    struct student* sptr;
    sptr=&s;
    sptr->name="JRN";
    sptr->roll=15;
    sptr->mark=85.25;
    cout<<"Details of student having Roll:"<<sptr->roll<<endl;
    cout<<"student name "<<sptr->name<<endl;
    cout<<"student mark "<<sptr->mark<<endl;
    return 0;
}
```

Output:

```
Details of student having Roll:15
student name JRN
student mark 85.25
```

Union:

- **Union** can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location.
- The union can also be defined as many members, but only one member can contain a value at a particular point in time.
- Syntax of union is same as structure.

```
union union_tag  
  
{  
  
    member definition;  
  
    member definition;  
  
    ...  
  
    member definition;  
  
} one or more union variables;
```

```

#include <iostream>
#include<string>
#include<cstring>
using namespace std;
union person {
    int age;
    char name[20];
    float hgt;
};

int main()
{
    union person p1;
    p1.age = 25;
    strcpy(p1.name, "JRN");
    p1.hgt = 5.9;
    cout<< p1.name<<"\t"<<p1.age<<"\t"<<p1.hgt;
    return 0;
}

```

Output: = || @-o Lu@→@ 1086115021 5.9

Here, we can see that the values of **age** and **name** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **hgt** member is getting printed very well.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
union union_job {
    char name[20];
    float salary;
    int empid;
}ujob;
struct struct_job {
    char name[20];
    float salary;
    int empid;
}sjob;
int main()
{
    cout<< "Size of union: "<<sizeof(ujob)<<endl;
    cout<< "Size of structure: "<<sizeof(sjob);
    return 0;
}
```

Output:

Size of union: 20

Size of structure: 28

Difference between Structure and Union

Structure	Union
The struct keyword is used to define a structure.	The union keyword is used to define union.
When the variables are declared in a structure, the compiler allocates memory to each variables member. The size of a structure is equal or greater to the sum of the sizes of each data member.	When the variable is declared in the union, the compiler allocates memory to the largest size variable member. The size of a union is equal to the size of its largest data member size.
Each variable member occupied a unique memory space.	Variables members share the memory space of the largest size variable.
Changing the value of a member will not affect other variables members.	Changing the value of one member will also affect other variables members.
Each variable member will be assessed at a time.	Only one variable member will be assessed at a time.
We can initialize multiple variables of a structure at a time.	In union, only the first data member can be initialized.
All variable members store some value at any point in the program.	Exactly only one data member stores a value at any particular instance in the program.
The structure allows initializing multiple variable members at once.	Union allows initializing only one variable member at once.
It is used to store different data type values.	It is used for storing one at a time from different data type values.
It allows accessing and retrieving any data member at a time.	It allows accessing and retrieving any one data member at a time.

Bit Fields:

In C++, we can specify the size (in bits) of the structure and union members. The idea of bit-field is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range. C++ Bit fields are used when the storage of our program is limited.

Need of Bit Fields in C++

- i. Reduces memory consumption.
- ii. To make our program more efficient and flexible.
- iii. Easy to Implement.

You cannot use pointers to the bit field member.

Example: Let's having a struct data type which has three data members: date, month and year of integer type. This data type will represent the date of birth, and three integers will occupy 12 bytes of memory. But we need not store these values in huge memory since we know the maximum value of date can be 31 and month can be 12, and year can be of maximum digits. So we use the bit fields to save the memory. In the bit field, we can explicitly give the width or the range to the data member in terms of bytes.

Syntax:

```
struct  
  
{  
  
    data_type member_name : width_of_bit-field;  
  
};
```

data_type: It is an integer type that determines the bit-field value which is to be interpreted. The type may be int, signed int, or unsigned int.

member_name: The member name is the name of the bit field.

width_of_bit-field: The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
struct dob {
    int date;
    int month;
    int year;
}sdob;
struct dob1 {
    int dd:5;
    int mm:4;
    int yy:12;
}sdob1;
int main()
{
    cout<< "Size of structure: "<<sizeof(sdob)<<endl;
    cout<< "Size of bitfield: "<<sizeof(sdob1);
    return 0;
}
```

Output:

Size of structure: 12

Size of bitfield: 4

Nested structure:

C++ provides us the feature of nesting one structure within another structure by using which, complex data types are created. It can help to improve a program's readability and maintainability.

Syntax:

```
struct outer_structure {  
    type member1;  
    type member2;  
    struct inner_structure {  
        type inner_member1;  
        type inner_member2;  
    } inner;  
}outer;
```

Syntax to access inner_member1

outer.inner.inner_member1 = value;

Syntax to define a pointer to a nested structure

```
struct outer_structure *ptr;  
ptr = &outer;  
ptr->inner.inner_member1 = value;
```

The structure can be nested in the following different ways: (i) By separate nested structure
(ii) By embedded nested structure.

(i) By separate nested structure:

In this method, the two structures are created, but the dependent structure should be used inside the main structure as a member.

Syntax:

```
struct dependent_structure
{
    type member1;
    type member2;
}var1:

struct main_structure {
    type inner_member1;
    type inner_member2;
    struct dependent_structure var1;
}var2;
```

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
struct empl {
    int empid;
    char name[20];
    int salary;
}emp;
struct organization {
    char org_name[20];
    struct empl emp;
}org;
```

```
int main() {
    org.emp.empid=101;
    strcpy(org.emp.name,"JRN");
    org.emp.salary=100000;
    strcpy(org.org_name,"ITER");
    cout<< "Organization: "<<org.org_name<<endl;
    cout<< "Employee: "<<org.emp.name<<endl;
    cout<< "Employee ID: "<<org.emp.empid<<endl;
    cout<< "Salary: "<<org.emp.salary;
    return 0;
}
```

Output:

Organization: ITER
Employee: JRN
Employee ID: 101
Salary: 100000

(ii) By embedded nested structure:

- Using this method, allows to declare structure inside a structure and it requires fewer lines of code.
- The variable declaration is compulsory at the end of the inner structure, which acts as a member of the outer structure.
- Error will occur if the structure is present but the structure variable is missing.

Syntax:

```
struct outer_structure {  
    type member1;  
    type member2;  
    struct inner_structure {  
        type inner_member1;  
        type inner_member2;  
    } inner;  
}outer;
```

```
#include <iostream>

#include<string>

#include<cstring>

using namespace std;

struct organization {

    char org_name[20];

    struct empl {

        int empid;

        char name[20];

        int salary;

    }empl;

}org;
```

```
int main() {

    org.emp.empid=101;

    strcpy(org.emp.name,"JRN");

    org.emp.salary=100000;

    strcpy(org.org_name,"ITER");

    cout<< "Organization: "<<org.org_name<<endl;

    cout<< "Employee: "<<org.emp.name<<endl;

    cout<< "Employee ID: "<<org.emp.empid<<endl;

    cout<< "Salary: "<<org.emp.salary;

    return 0;

}
```

Output:

Organization: ITER
Employee: JRN
Employee ID: 101
Salary: 100000

Arrays of Structures:

- An array whose elements are of type structure is called array of structure. It is generally useful when we need multiple structure variables in our program.
- Suppose we have 50 employees and we need to store the data of 50 employees. So for that, we need to define 50 variables of struct Employee type and store the data within that. However, declaring and handling the 50 variables is not an easy task. Let's imagine a bigger scenario, like 1000 employees.

Syntax:

```
struct tag_name  
  
{  
  
    data_type member1;  
    data_type member2;  
  
    .  
  
    data_type memberN;  
  
}variable[numbers of variables];
```



```

#include<iostream>
#include<string>
using namespace std;
struct student
{
    string name;
    int roll;
    float mark;
}s[2];
int main()
{
    int i;
    for (i=0;i<2;i++)
    {
        s[i].roll=i+1;
        cout<<"Enter student name"<<endl;
        cin>>s[i].name;
        cout<<"Enter student mark"<<endl;
        cin>>s[i].mark;
    }
}

```

```

for (i=0;i<2;i++)
{
    cout<<"Details of student having Roll:"<<s[i].roll<<endl;
    cout<<"student name"<<s[i].name<<endl;
    cout<<"student mark"<<s[i].mark<<endl;
}
return 0;
}

```

Enter student name

JRN

Enter student mark

85.23

Enter student name

SD

Enter student mark

87.58

Details of student having Roll:1

student nameJRN

student mark85.23

Details of student having Roll:2

student nameSD

student mark87.58

Structure and Functions

Passing Structure Members as Arguments to Function

We can pass individual structure members as arguments to functions like any other ordinary variable.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
struct empl {
    int empid;
    char name[20];
    int salary;
}emp;

void display(int id, char name[], int salary)
{
    cout<< "Employee: "<<name<<endl;
    cout<< "Employee ID: "<<id<<endl;
    cout<< "Salary: "<<salary;
}

int main()
{
    emp.empid=101;
    strcpy(emp.name,"JRN");
    emp.salary=100000;
    display(emp.empid, emp.name, emp.salary);
    return 0;
}
```

Output:

Employee: JRN
Employee ID: 101
Salary: 10000

Passing Structure Variable as Argument to Function

If a structure contains two to three members, we can easily pass them to function, but it is a tiresome and error-prone process when there are many members. So in such cases, instead of passing members individually, we can pass the whole structure as an argument.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
struct empl {
    int empid;
    char name[20];
    int salary;
}emp;
void display(struct empl emp)
{
    cout<< "Employee: "<<emp.name<<endl;
    cout<< "Employee ID: "<<emp.empid<<endl;
    cout<< "Salary: "<<emp.salary;
}
```

```
int main()
{
    emp.empid=101;
    strcpy(emp.name,"JRN");
    emp.salary=100000;
    display(emp);
    return 0;
}
```

Output:

```
Employee: JRN
Employee ID: 101
Salary: 10000
```

Passing Structure Pointers as Argument to Function

C++ provides the efficient method known as Call by reference to pass large structures to functions. In this method, we send the address of the structure to the function.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
struct empl {
    int empid;
    char name[20];
    int salary;
}emp;
void display(struct empl *ptr)
{
    cout<< "Employee: "<<ptr->name<<endl;
    cout<< "Employee ID: "<<ptr->empid<<endl;
    cout<< "Salary: "<<ptr->salary;
}
```

```
int main()
{
    emp.empid=101;
    strcpy(emp.name,"JRN");
    emp.salary=100000;
    display(&emp);
    return 0;
}
```

Output:

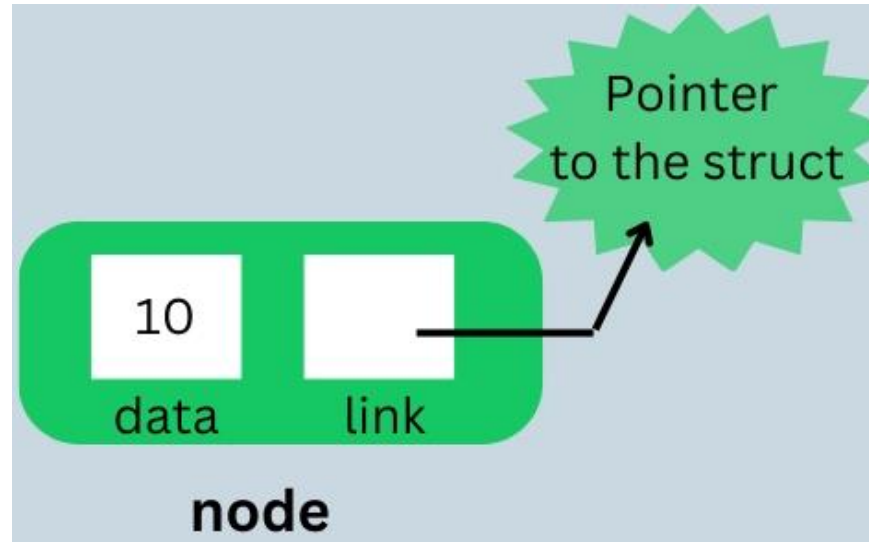
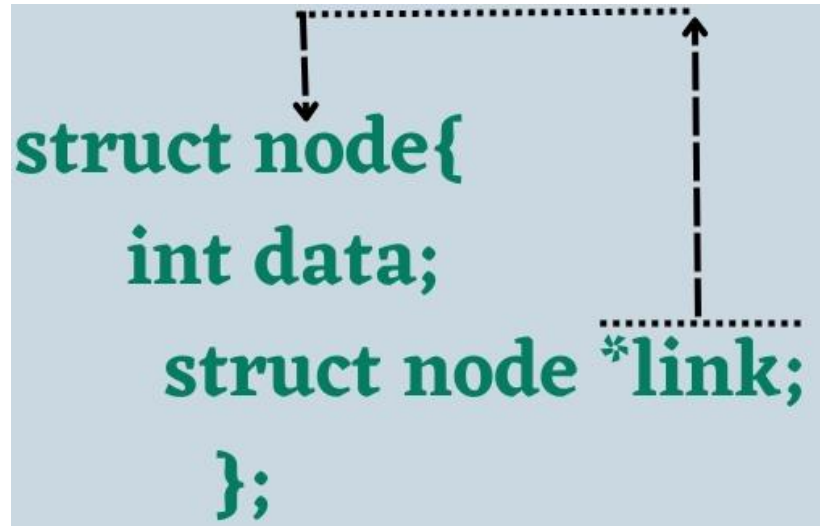
```
Employee: JRN
Employee ID: 101
Salary: 10000
```

Self referential structure:

A Self referential structure is a structure that contains a member that points to the same type of structure. It is also called linked structure or recursive structure.

Syntax:

```
struct node{  
    int data;  
    struct node *link;  
};
```



The struct contains two members in this above code, data and link. The link is a pointer pointing to the same struct node type. This makes the node a self referential structure.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
struct node {
    int data;
    node *next;
}obj;
int main()
{
    obj.data=60;
    obj.next=NULL;
    cout<< obj.data<<'\\t'<<obj.next<<endl;
    return 0;
}
```

Output:

60 0

- The value of the pointer is initialized by **NULL** so that the pointer should not contain any garbage value.

Enumeration data type:

Enumeration (or enum) is a user defined data type in C++. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

Syntax:

enum enum_name{int_const1, int_const2, int_const3, int_constN};

or

enum enum_name{int_const1, int_const2, int_const3, int_constN}var;

or

enum enum_name{int_const1, int_const2, int_const3, int_constN};

enum enum_name var;

```
#include <iostream>
using namespace std;
enum Bool{False, True};
int main()
{
    enum Bool var;
    var=True;
    cout<< var;
    return 0;
}
```

Output: 1

- If we do not assign values to enum names then automatically compiler will assign values to them starting from 0.
- Enums can be declared in the local scope.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
int main()
{
    enum Bool{False, True}var;
    var=False;
    cout<<"False is assigned as "<<var<<endl;
    var=True;
    cout<<"True is assigned as "<<var<<endl;
    return 0;
}
```

Output:

*False is assigned as 0
True is assigned as 1*

- Two or more names can have same value.

```
#include <iostream>
using namespace std;
int main()
{
    enum Bool{False=0, True=0}var;
    cout<<"False is "<<False<<endl;
    cout<<"True is "<<True<<endl;
    return 0;
}
```

Output:
False is 0
True is 0

- We can assign values in any order. All unassigned names will get value as value of previous name+1.

```
#include <iostream>
using namespace std;
int main()
{
    enum point{w=4,x=8,y, z=32};
    cout<<w<<"\t"<<x<<"\t"<<y<<"\t"<<z<<endl;
    return 0;
}
```

Output:
4 8 9 32

y is unassigned, So, y value = x value +1 = 8 + 1 = 9

- Only integral values are allowed.

```
#include <iostream>
using namespace std;
int main()
{
    enum point{w=4,x=8.5,z=32};
    cout<<w<<"\t"<<x<<"\t"<<z<<endl;
    return 0;
}
```

Error: enumerator value for 'x' is not an integer constant

- All enum constant must be unique in their scope.

```
#include <iostream>
using namespace std;
int main()
{
    enum point1{w=4,x=8,z=32};
    enum point2{x=14,p=5,q=2};
    cout<<w<<"\t"<<x<<"\t"<<z<<endl;
    return 0;
}
```

error: redeclaration of 'x'

```
#include <iostream>

using namespace std;

enum year{Jan=1, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec};

int main()
{
    for(int i=Jan; i<=Dec; i++)
    {
        cout<<i<<"\t";
    }
    return 0;
}
```

Output:

1 2 3 4 5 6 7 8 9 10 11 12

typedef keyword:

- 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.

Syntax:

```
typedef existing_name alias_name;
```

Syntax of typedef for structure:

```
typedef struct tag_name
```

```
{
```

```
    data_type member1;
```

```
    data_type member2;
```

```
    data_type memeberN;
```

```
}variable;
```

```
variable alias_name;
```

```

#include <iostream>

using namespace std;

typedef float f;

int main()
{
    f x,y,z;

    x=1.05; y=2.22; z=4.53;

    cout<<x<<"\t"<<y<<"\t"<<z;

    return 0;
}

```

Output:

1.05 2.22 4.53

```

#include <iostream>
#include<string>
#include<cstring>
using namespace std;
typedef struct student{
    char name[20];
    char dept[10];
    int id;
}stud;
int main()
{
    stud s;
    strcpy(s.name, "JRN");
    strcpy(s.dept, "Training");
    s.id=12;
    cout<<"Name: "<< s.name<<endl;
    cout<<"Department: "<< s.dept<<endl;
    cout<<"ID: "<< s.id<<endl;
    return 0;
}

```

Output:

Name: JRN

Department: Training

ID: 12

Thank You