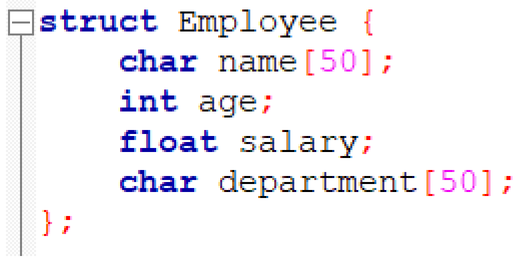
## 1. Structures and Their Purpose:-

1) A structure is a user-defined data type that allows the grouping of different data types under a single name. This is useful when you need to represent an entity with different characteristics, such as a "person" with a name, age, and address. The primary purpose of using structures is to model complex data in an organized manner. They allow you to keep related data together and access it as a single unit.
2) For example, consider a scenario where you need to keep track of an employee’s details. Instead of using separate variables for each attribute, a structure helps group all these attributes together.

Example of a structure declaration for an employee:



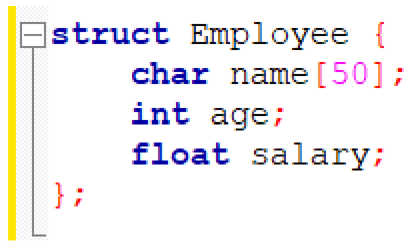
```
struct Employee {
    char name[50];
    int age;
    float salary;
    char department[50];
};
```

Here, the Employee structure holds four fields: a string for the name, an integer for age, a floating-point number for salary, and a string for the department.

## 2. Declaration and Initializing the Structure:-

**Declaration:** To declare a structure, the struct keyword is used, followed by the structure name and a set of curly braces containing the variables (members) of the structure. Each member can be of a different type, allowing flexibility in representing complex data.

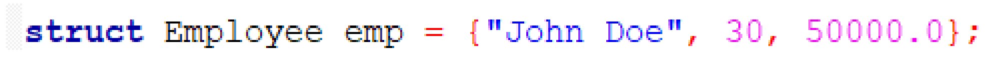
Example:



```
struct Employee {
    char name[50];
    int age;
    float salary;
};
```

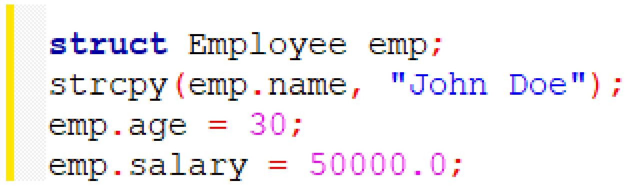
**Initialization:** After declaration, a structure can be initialized with specific values either at the time of declaration or after it’s declared. The fields can be initialized with specific values using curly braces.

Example of initialization during declaration:



```
struct Employee emp = {"John Doe", 30, 50000.0};
```

Example of later initialization:

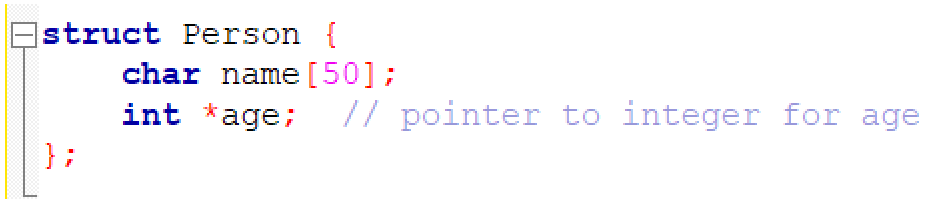


```
struct Employee emp;
strcpy(emp.name, "John Doe");
emp.age = 30;
emp.salary = 50000.0;
```

## 3. Pointers and Arrays in Structures:-

1) **Pointers in Structures:** Structures can contain pointers as members. A pointer is a variable that stores the memory address of another variable. By using pointers inside structures, it becomes possible to dynamically allocate memory and manage complex data relationships. Pointers in structures are often used for dynamically allocating memory for large objects or arrays and managing relationships between structures.

Example with a pointer:

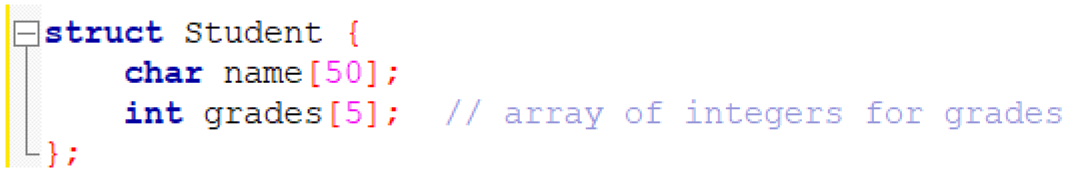


```
struct Person {
    char name[50];
    int *age;   // pointer to integer for age
};
```

Here, the age field is a pointer that could point to dynamically allocated memory, allowing changes to the age value without having to modify the structure directly.

- **Arrays in Structures:** Just like individual variables, arrays can also be members of structures. Arrays allow you to store multiple values of the same type under a single member. Arrays within structures are helpful when you want to group multiple related items, such as storing multiple grades for a student.

Example of an array in a structure:



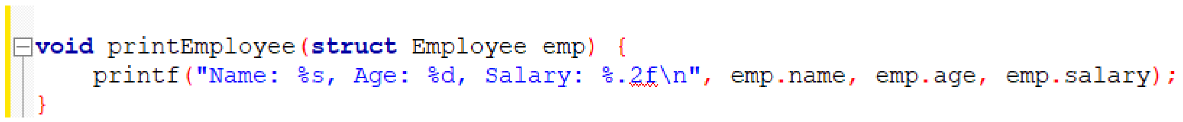
```
struct Student {
    char name[50];
    int grades[5];   // array of integers for grades
};
```

## 4. Passing Structures to Functions:

Structures can be passed to functions in two ways: **by value** and **by reference**.

- **Passing by Value:** When passing a structure by value, a copy of the entire structure is created and passed to the function. Any changes made to the structure inside the function do not affect the original structure outside the function.

Example:

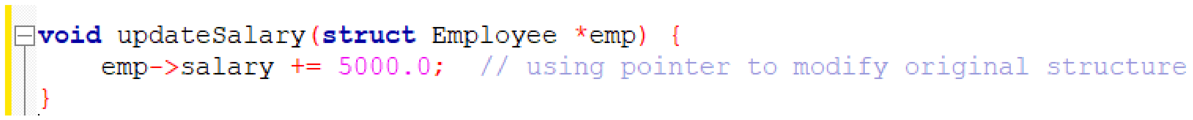


```c
void printEmployee(struct Employee emp) {
    printf("Name: %s, Age: %d, Salary: %.2f\n", emp.name, emp.age, emp.salary);
}
```

In this example, the emp structure is passed by value, so the original structure remains unaffected by any modifications inside the function.

- **Passing by Reference:** To pass a structure by reference, a pointer to the structure is passed to the function. This allows the function to modify the original structure. This method is often more efficient than passing by value, especially when the structure is large.

  Example:



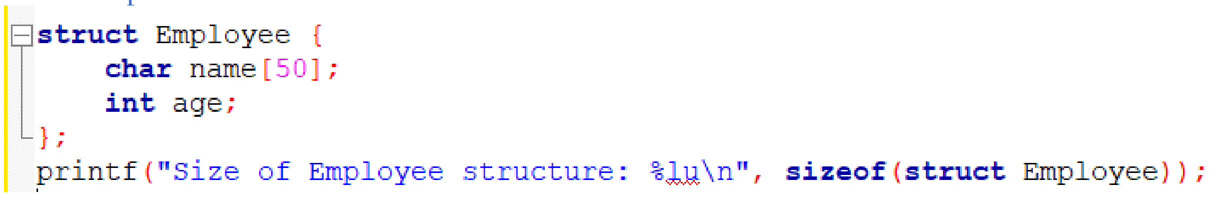
```c
void updateSalary(struct Employee *emp) {
    emp->salary += 5000.0;  // using pointer to modify original structure
}
```

  In this example, the emp structure is passed by reference, and the original structure is modified directly inside the function.

## 5. Size of Structure:-

1) The size of a structure is determined by the sum of the sizes of its individual members. However, due to memory alignment and padding, the total size may not always be exactly the sum of the member sizes. Memory alignment is the process of aligning data types in memory according to their size, ensuring that they are placed at addresses that are multiples of their size.
2) To calculate the size of a structure, the `sizeof()` operator is used.

Example:



```
struct Employee {
    char name[50];
    int age;
};
printf("Size of Employee structure: %lu\n", sizeof(struct Employee));
```

In the above example, the size of the structure may be larger than the sum of the size of its members (50 bytes for name and 4 bytes for age) due to padding added by the compiler to maintain alignment.

## 6. Memory Padding, Aligned Memory, and Unaligned Memory (Comparison):

- **Memory Padding:** Memory padding is added by the compiler to ensure that the data members of a structure are properly aligned in memory according to their data type. This alignment ensures that data is accessed efficiently, which can improve performance. Padding is often added between members of the structure to make sure that each member starts at a memory address that satisfies the alignment requirements.

- **Aligned Memory:** Aligned memory refers to the arrangement of data so that each data type is stored at a memory address that is a multiple of its size. For example, an integer (which is typically 4 bytes) should be stored at a memory address that is a multiple of 4. Aligned memory improves memory access speed and is more efficient.

- **Unaligned Memory:** Unaligned memory occurs when data is not stored at the proper boundary, leading to inefficient memory access and potentially slower performance. Some architectures (such as older processors) might not allow unaligned memory, causing errors when attempting to access such data.

**Comparison:** Structures with properly aligned memory are more efficient, both in terms of access time and memory usage. Unaligned memory can lead to inefficient memory accesses, increased overhead, and may cause errors on certain hardware.

## 7. Difference Between Structures and Objects (Theoretical, Not Code):

- **Structures:** A structure is a simple data container that holds a collection of variables (data members) together under one name. Structures are widely used in procedural programming and are simply used to represent related data. They are data-centric and lack the concept of methods or functions that act on the data. Structures are ideal for data representation where behavior is not associated with the data.
- **Objects:** An object is an instance of a class in object-oriented programming (OOP). Objects contain both data (attributes) and functions (methods) that operate on the data. OOP allows for encapsulation (bundling data and methods), inheritance (creating new classes from existing ones), and polymorphism (methods that can operate on objects of different classes). Objects are more versatile than structures because they allow modeling of both data and behaviors.

### Comparison:

- **Structures:** A structure is a way of grouping data together, but it lacks the ability to contain methods or behaviors. It is more about holding related data in one place.
- **Objects:** An object is a more complex concept that is capable of holding both data and methods. It provides the foundation for OOP principles like encapsulation, inheritance, and polymorphism.