rtp

week 5_1:

struct

rajin teaches programming

INTRODUCTION:

In C programming, a **`struct`** (short for "**structure**") is a **user-defined data type** that allows the **combination** of data items of different kinds. **`structs`** provide a way to **group related variables** together, making it **easier** to manage **complex data structures.** This is especially useful in scenarios where you want to model **real-world entities**, such as a **student**, an **employee**, or a **product**, which have **multiple attributes.**

– IMPORTANCE:

Organization: Helps in grouping related data together.
Clarity: Enhances code readability by representing real-world entities more naturally.
Modularity: Facilitates better data management and manipulation.
Efficiency: Reduces complexity when handling large amounts of data.

– WHAT IS A ‘STRUCT’?

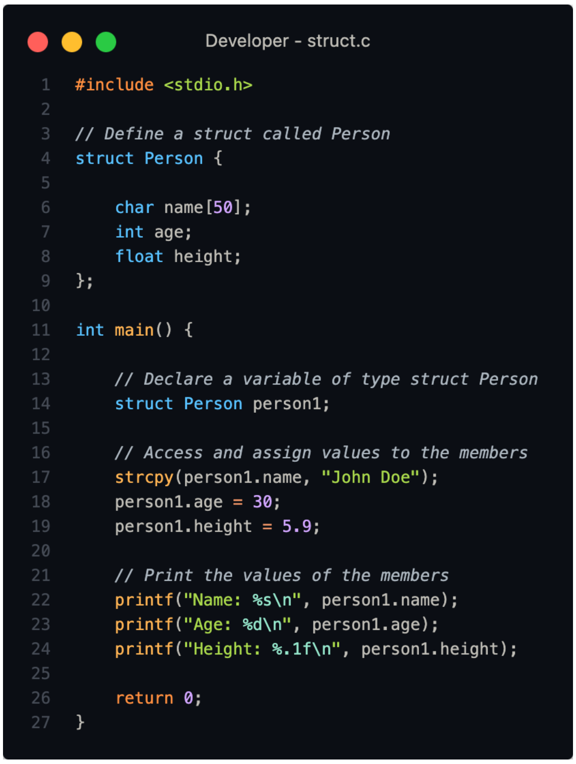
A **`struct`** is a **composite** data type that **groups variables** of **different types** under a **single** name. Each **variable** within a **`struct`** is called a **member.** By using `structs`, you can create **complex** data types that model **real-world** objects more **accurately.**

– HOW IT WORKS:

Each **member** of a **`struct`** is **stored** in **contiguous** memory locations. **Members** of a **`struct`** can be **accessed** using the **dot operator “.”** if you have a **`struct` variable,** or the **arrow operator** (->) if you have a **pointer** to a `struct` (to be discussed later).

– USAGE:

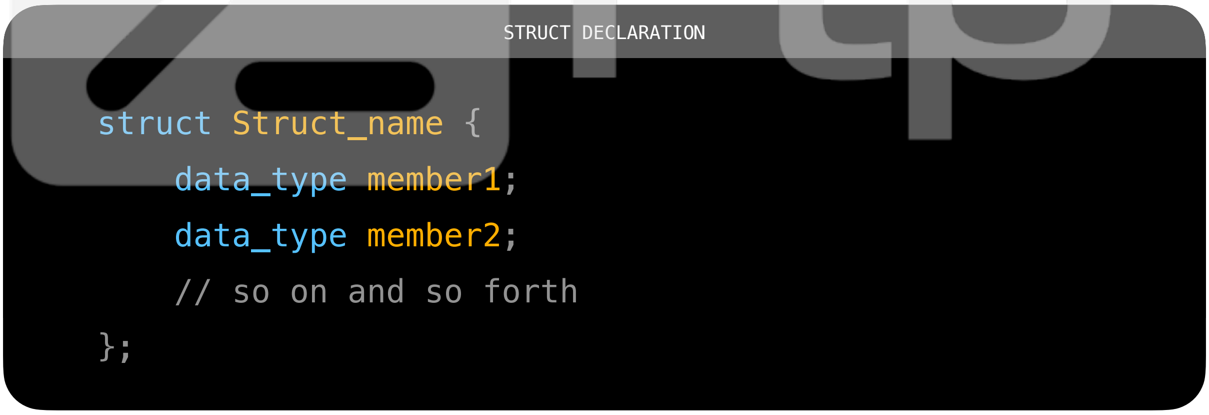
To **declare** a **`struct`**, you use the **`struct` keyword** followed by the **structure definition.** A **basic** usage has been shown in the following:



```c
#include <stdio.h>

// Define a struct called Person
struct Person {

    char name[50];
    int age;
    float height;
};

int main() {

    // Declare a variable of type struct Person
    struct Person person1;

    // Access and assign values to the members
    strcpy(person1.name, "John Doe");
    person1.age = 30;
    person1.height = 5.9;

    // Print the values of the members
    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    printf("Height: %.1f\n", person1.height);

    return 0;
}
```

an example of struct being used.

SYNTAX:

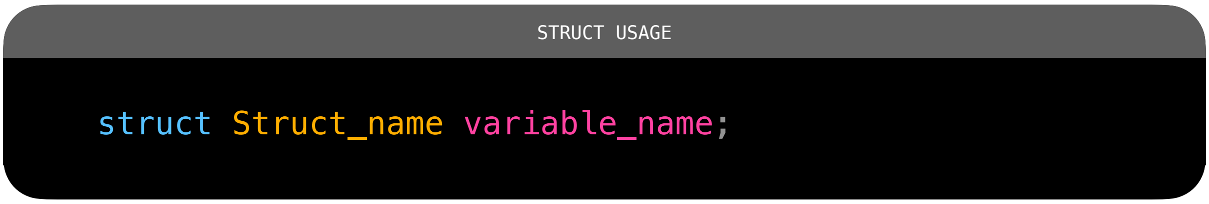


```c
struct Struct_name {
    data_type member1;
    data_type member2;
    // so on and so forth
};
```

As we can see above, the **struct** data structure is created using the ‘**struct**’ keyword. Curly braces enclose **primitive data types** (such as **int**, **char**, etc.). Once assigned, these data types, that make up the **struct object**, are called **members.**

Like **functions**, **structs** must be **declared** in **advance**, before the **main** function, so that the **compiler** knows which struct is being **referred** to. Since structs do **not** have any **prototypes**, it **cannot** be declared **after** the **main** function.

– USING A STRUCT:



STRUCT USAGE

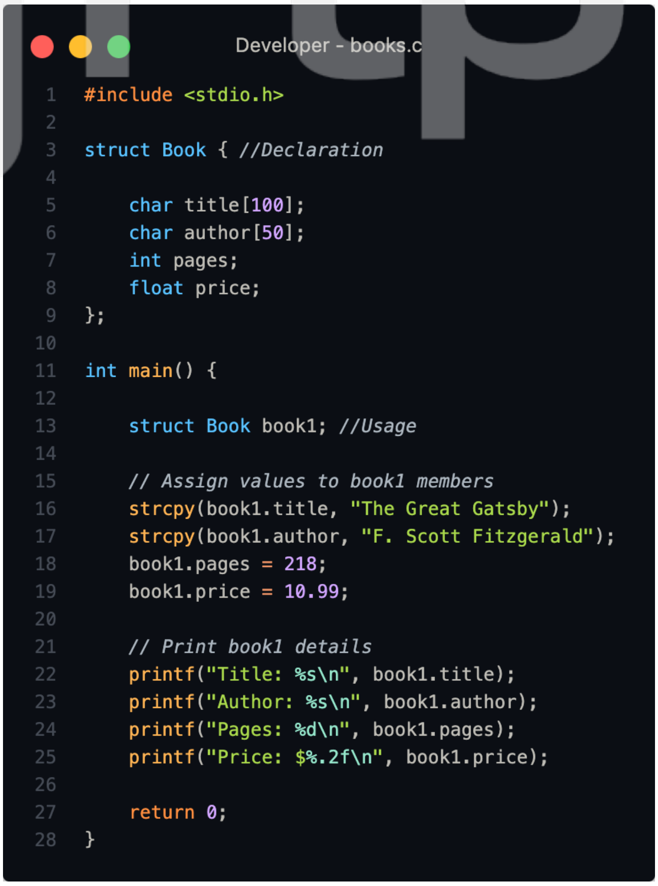
```c
struct Struct_name variable_name;
```

Using a **struct** in the **main** function is fairly easy. We just need to **call** it in the main function **like** a **data type**, using the **struct** keyword, and give the **instance** of that struct a **new name.** Conventionally, structs are **always** declared with **capital letters** to make the **differentiation** between the **struct name** and the **instance/ variable** easier.

– ACCESSING THE DATA TYPES IN STRUCT (MEMBERS):

Data types in struct can be **accessed** using the dot "**.**" operator. An example of declaration usage, and access has been shown below:

the strcpy() method is always used to assign values to string members. can you guess why?



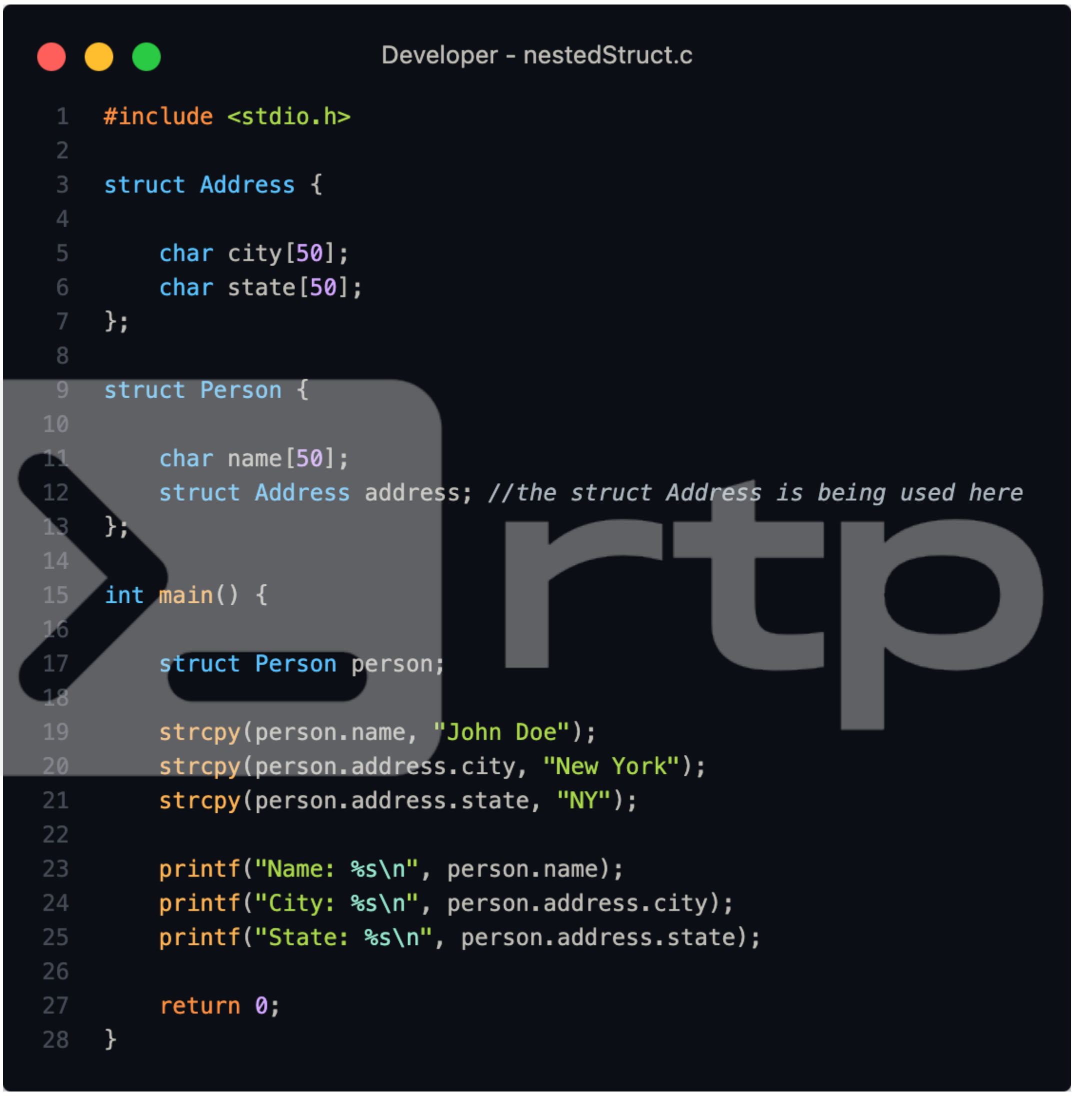
Developer - books.c

```c
#include <stdio.h>

struct Book { //Declaration

    char title[100];
    char author[50];
    int pages;
    float price;
};

int main() {

    struct Book book1; //Usage

    // Assign values to book1 members
    strcpy(book1.title, "The Great Gatsby");
    strcpy(book1.author, "F. Scott Fitzgerald");
    book1.pages = 218;
    book1.price = 10.99;

    // Print book1 details
    printf("Title: %s\n", book1.title);
    printf("Author: %s\n", book1.author);
    printf("Pages: %d\n", book1.pages);
    printf("Price: $%.2f\n", book1.price);

    return 0;
}
```

another example of struct usage

- NESTED STRUCTS:

You can **nest** \`structs\` **within other** \`structs\` to model more complex data. An example is provided below:



```c
#include <stdio.h>

struct Address {

    char city[50];
    char state[50];
};

struct Person {

    char name[50];
    struct Address address; //the struct Address is being used here
};

int main() {

    struct Person person;

    strcpy(person.name, "John Doe");
    strcpy(person.address.city, "New York");
    strcpy(person.address.state, "NY");

    printf("Name: %s\n", person.name);
    printf("City: %s\n", person.address.city);
    printf("State: %s\n", person.address.state);

    return 0;
}
```

an example of nested struct usage

Notice how, in order to access the **Address** struct’s **members,** we first have to use the **struct Person, then call** the **Address** struct using the **dot operator,** and **then** finally **call** the **Address** struct’s **members.**

COMMON USE CASES:

Database Records: `structs` can represent records in a database, where each record consists of multiple fields.

Linked Lists: Nodes in a linked list can be represented using `structs`, where each node contains data and a pointer to the next node.

Complex Data Types: Modeling complex data types that consist of multiple attributes, such as geometric shapes (e.g., a rectangle with width and height).

Student Records: Each student has attributes like name, roll number, and marks.

Employee Records: Each employee has attributes like name, ID, and salary.

BEST PRACTICES:

Always initialize `struct` **members** to avoid undefined behavior. Choose **meaningful names** for `struct` **members** to improve code **readability.** And most importantly, be **cautious** with very **large `structs`**, as they can **impact performance** and **memory usage.**

SOME CONS:

Uninitialized Members: Accessing uninitialized members can lead to unpredictable results.

Memory Alignment Issues: Be aware of memory alignment requirements on different platforms.

While **structs** can be used with **pointers,** (meaning we can create pointers to structs), I do not deem it necessary to learn for our course. Simply knowing **how** to use the **dot operator** is **enough.** You can learn **further** on pointers to structs if you wish to do so.

SUMMARY:

A **`struct` groups** variables of **different** types under a **single** name. They provide **better data organization, clarity**, and **modularity.** They are crucial for modeling complex data types in C. Proper initialization, meaningful names, and cautious memory management are essential for effective use of `structs`.

SOME FAQs:

What is the difference between a `struct` and an array?
An array is a collection of elements of the same type, whereas a `struct` can contain elements of different types.

How do you pass a `struct` to a function?
You can pass a `struct` to a function either by value or by reference using pointers.

rtp

next class 5_2:

file handling

rajin teaches programming

reach out for classes at: rajin.khan2001@gmail.com