C Programming: Structures

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

January 16, 2026

Topics Covered

What are Structures?

- User-defined data type
- Groups related variables together
- Variables can be of different types
- Members accessed using dot operator
- Also called "struct" or "record"

Why Use Structures?

- Organize related data
- Represent real-world entities
- Better than separate variables
- Pass multiple values to functions
- Create complex data types

Example Use Cases:

- Student (name, roll, marks)
- Book (title, author, price)
- Point (x, y coordinates)

Structure Declaration and Definition

Declaration:

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

Creating Variables:

```
struct structure_name variable_name;
```

Accessing Members:

```
variable_name.member1 = value;
```

Note: Structure declaration ends with semicolon

Program 1: Simple Structure

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
int main() {
  struct Point p1;
  p1.x = 10;
  p1.y = 20;
  printf("Point p1:\n");
  printf("x = %d\n", p1.x);
  printf("y = %d\n", p1.y);
  printf("\nCoordinates: (%d, %d)\n",
         p1.x, p1.y);
  return 0;
}
```

Output:

```
Point p1:
x = 10
y = 20

Coordinates: (10, 20)
```

Explanation:

- struct Point has 2 members
- p1 is variable of type Point
- Dot operator accesses members
- Each member is separate

Program 2: Structure with Different Types

```c
#include <stdio.h>
#include <string.h>
struct Student {
  int roll;
  char name[50];
  float marks;
};
int main() {
  struct Student s1;
  s1.roll = 101;
  strcpy(s1.name, "Alice");
  s1.marks = 85.5;
  printf("Student Details:\n");
  printf("Roll: %d\n", s1.roll);
  printf("Name: %s\n", s1.name);
  printf("Marks: %.2f\n", s1.marks);
  return 0;
}
```

Output:

```
Student Details:
Roll: 101
Name: Alice
Marks: 85.50
```

Note:

- int, char array, float members
- Different data types
- String needs strcpy
- All related to one student

Program 3: Structure Initialization

```c
#include <stdio.h>
struct Book {
  char title[30];
  char author[30];
  float price;
};
int main() {
  struct Book b1 = {"C Programming",
                    "Dennis Ritchie",
                    45.50};
  printf("Book Details:\n");
  printf("Title: %s\n", b1.title);
  printf("Author: %s\n", b1.author);
  printf("Price: $%.2f\n", b1.price);
  return 0;
}
```

Output:

```
Book Details:
Title: C Programming
Author: Dennis Ritchie
Price: $45.50
```

Note:

- Initialize at declaration
- Values in curly braces
- Order matches members
- Cleaner than separate assignments

Program 4: Multiple Structure Variables

```c
#include <stdio.h>
struct Rectangle {
  int length;
  int width;
};
int main() {
  struct Rectangle r1 = {10, 5};
  struct Rectangle r2 = {8, 6};
  printf("Rectangle 1:\n");
  printf("Length: %d, Width: %d\n",
         r1.length, r1.width);
  printf("Area: %d\n",
         r1.length * r1.width);
  printf("\nRectangle 2:\n");
  printf("Length: %d, Width: %d\n",
         r2.length, r2.width);
  printf("Area: %d\n",
         r2.length * r2.width);
  return 0;
}
```

Output:

```
Rectangle 1:
Length: 10, Width: 5
Area: 50

Rectangle 2:
Length: 8, Width: 6
Area: 48
```

Note:

- Two variables of same type
- Independent instances
- Each has own members

Program 5: Array of Structures

```c
#include <stdio.h>
struct Student {
  int roll;
  float marks;
};
int main() {
  struct Student students[3] = {
    {101, 85.5},
    {102, 90.0},
    {103, 78.5}
  };
  int i;
  printf("Student Records:\n");
  for (i = 0; i < 3; i++) {
    printf("Roll: %d, Marks: %.1f\n",
           students[i].roll,
           students[i].marks);
  }
  return 0;
}
```

Output:

```
Student Records:
Roll: 101, Marks: 85.5
Roll: 102, Marks: 90.0
Roll: 103, Marks: 78.5
```

Note:

- Array of 3 students
- Each element is struct
- Access: array[i].member
- Nested initialization

Program 6: Find Highest Marks

```c
#include <stdio.h>
struct Student {
  int roll;
  char name[20];
  float marks;
};
int main() {
  struct Student s[3] = {
    {101, "Alice", 85.5},
    {102, "Bob", 92.0},
    {103, "Charlie", 78.5}
  };
  int i, maxIndex = 0;
  for (i = 1; i < 3; i++) {
    if (s[i].marks > s[maxIndex].marks) {
      maxIndex = i;
    }
  }
  printf("Highest marks:\n");
  printf("Name: %s\n", s[maxIndex].name);
  printf("Marks: %.1f\n",
         s[maxIndex].marks);
  return 0;
}
```

Output:

```
Highest marks:
Name: Bob
Marks: 92.0
```

Logic:

- Array of 3 students
- Compare marks
- Track index of max
- Display winner

Program 7: Calculate Average Marks

```c
#include <stdio.h>
struct Student {
  int roll;
  float marks;
};
int main() {
  struct Student s[4] = {
    {101, 85.0},
    {102, 90.0},
    {103, 78.0},
    {104, 88.0}
  };
  float sum = 0;
  int i;
  for (i = 0; i < 4; i++) {
    sum += s[i].marks;
  }
  printf("Total students: 4\n");
  printf("Average marks: %.2f\n",
         sum / 4);
  return 0;
}
```

Output:

```
Total students: 4
Average marks: 85.25
```

Logic:

- Sum all marks
- Divide by count
- Loop through array
- Access marks member

Program 8: Nested Structures

```c
#include <stdio.h>
struct Date {
  int day;
  int month;
  int year;
};
struct Employee {
  int id;
  char name[30];
  struct Date joinDate;
};
int main() {
  struct Employee emp = {
    101,
    "John Doe",
    {15, 6, 2020}
  };
  printf("Employee Details:\n");
  printf("ID: %d\n", emp.id);
  printf("Name: %s\n", emp.name);
  printf("Join Date: %d/%d/%d\n",
         emp.joinDate.day,
         emp.joinDate.month,
         emp.joinDate.year);
  return 0;
}
```

Output:

```
Employee Details:
ID: 101
Name: John Doe
Join Date: 15/6/2020
```

Note:

- Date struct inside Employee
- Access: var.nested.member
- Two levels deep
- Initialization nested too

Program 9: Complex Nested Structure

```c
#include <stdio.h>
struct Address {
  int houseNo;
  char city[30];
};
struct Person {
  char name[30];
  int age;
  struct Address addr;
};
int main() {
  struct Person p = {
    "Alice",
    25,
    {123, "New York"}
  };
  printf("Person Details:\n");
  printf("Name: %s\n", p.name);
  printf("Age: %d\n", p.age);
  printf("Address: House %d, %s\n",
         p.addr.houseNo,
         p.addr.city);
  return 0;
}
```

Output:

```
Person Details:
Name: Alice
Age: 25
Address: House 123, New York
```

Note:

- Person contains Address
- Logical grouping
- Represents real-world relation
- Clean organization

Program 10: Pass Structure to Function

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
void printPoint(struct Point p) {
  printf("Point: (%d, %d)\n",
         p.x, p.y);
}
int main() {
  struct Point p1 = {10, 20};
  struct Point p2 = {30, 40};
  printf("Printing points:\n");
  printPoint(p1);
  printPoint(p2);
  return 0;
}
```

Output:

```
Printing points:
Point: (10, 20)
Point: (30, 40)
```

Note:

- Structure passed by value
- Copy passed to function
- Original unchanged
- Reusable function

Program 11: Return Structure from Function

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
struct Point createPoint(int x, int y) {
  struct Point p;
  p.x = x;
  p.y = y;
  return p;
}
int main() {
  struct Point p1, p2;
  p1 = createPoint(5, 10);
  p2 = createPoint(15, 20);
  printf("p1: (%d, %d)\n", p1.x, p1.y);
  printf("p2: (%d, %d)\n", p2.x, p2.y);
  return 0;
}
```

Output:

```
p1: (5, 10)
p2: (15, 20)
```

Note:

- Function returns struct
- Creates and returns Point
- Constructor-like function
- Convenient initialization

Program 12: Calculate Distance

```c
#include <stdio.h>
#include <math.h>
struct Point {
  int x;
  int y;
};
float distance(struct Point p1,
               struct Point p2) {
  int dx = p2.x - p1.x;
  int dy = p2.y - p1.y;
  return sqrt(dx*dx + dy*dy);
}
int main() {
  struct Point a = {0, 0};
  struct Point b = {3, 4};
  printf("Point A: (%d, %d)\n",
         a.x, a.y);
  printf("Point B: (%d, %d)\n",
         b.x, b.y);
  printf("Distance: %.2f\n",
         distance(a, b));
  return 0;
}
```

Output:

```
Point A: (0, 0)
Point B: (3, 4)
Distance: 5.00
```

Logic:

- Two Point parameters
- Calculate dx, dy
- Pythagorean theorem
- sqrt(3*3 + 4*4) = 5

Program 13: Using typedef

```c
#include <stdio.h>
typedef struct {
  int x;
  int y;
} Point;
int main() {
  Point p1 = {10, 20};
  Point p2 = {30, 40};
  printf("p1: (%d, %d)\n", p1.x, p1.y);
  printf("p2: (%d, %d)\n", p2.x, p2.y);
  printf("\nNo 'struct' keyword needed\n");
  return 0;
}
```

Output:

```
p1: (10, 20)
p2: (30, 40)

No 'struct' keyword needed
```

Note:

- typedef creates alias
- Point instead of struct Point
- Shorter, cleaner
- Common practice

Program 14: typedef with Named Struct

```c
#include <stdio.h>
#include <string.h>
typedef struct Student {
  int roll;
  char name[30];
  float marks;
} Student;
int main() {
  Student s1;
  s1.roll = 101;
  strcpy(s1.name, "Alice");
  s1.marks = 85.5;
  printf("Student Details:\n");
  printf("Roll: %d\n", s1.roll);
  printf("Name: %s\n", s1.name);
  printf("Marks: %.1f\n", s1.marks);
  return 0;
}
```

Output:

```
Student Details:
Roll: 101
Name: Alice
Marks: 85.5
```

Note:

- struct Student and Student both work
- Typedef after struct definition
- More flexible
- Both names available

Program 15: Structure Assignment

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
int main() {
  struct Point p1 = {10, 20};
  struct Point p2;
  printf("Before assignment:\n");
  printf("p1: (%d, %d)\n", p1.x, p1.y);
  p2 = p1;
  printf("\nAfter p2 = p1:\n");
  printf("p2: (%d, %d)\n", p2.x, p2.y);
  p1.x = 100;
  printf("\nAfter p1.x = 100:\n");
  printf("p1: (%d, %d)\n", p1.x, p1.y);
  printf("p2: (%d, %d)\n", p2.x, p2.y);
  return 0;
}
```

Output:

```
Before assignment:
p1: (10, 20)

After p2 = p1:
p2: (10, 20)

After p1.x = 100:
p1: (100, 20)
p2: (10, 20)
```

Note:

- Can assign one struct to another
- All members copied
- Independent copies

Program 16: Structure Comparison

```c
#include <stdio.h>
struct Point {
  int x;
  int y;
};
int areEqual(struct Point p1,
             struct Point p2) {
  return (p1.x == p2.x) &&
         (p1.y == p2.y);
}
int main() {
  struct Point a = {10, 20};
  struct Point b = {10, 20};
  struct Point c = {30, 40};
  printf("a and b: %s\n",
         areEqual(a,b) ? "Equal" :
                         "Not equal");
  printf("a and c: %s\n",
         areEqual(a,c) ? "Equal" :
                         "Not equal");
  return 0;
}
```

Output:

```
a and b: Equal
a and c: Not equal
```

Note:

- Cannot use == on structs
- Must compare member by member
- Function for comparison
- Returns 1 if equal, 0 if not

Program 17: Time Structure

```c
#include <stdio.h>
struct Time {
  int hours;
  int minutes;
  int seconds;
};
void printTime(struct Time t) {
  printf("%02d:%02d:%02d\n",
         t.hours, t.minutes, t.seconds);
}
int main() {
  struct Time t1 = {9, 30, 45};
  struct Time t2 = {14, 5, 30};
  printf("Time 1: ");
  printTime(t1);
  printf("Time 2: ");
  printTime(t2);
  return 0;
}
```

Output:

```
Time 1: 09:30:45
Time 2: 14:05:30
```

Note:

- Time representation
- Format: HH:MM:SS
- %02d for leading zeros
- Clean display function

Program 18: Complex Number Operations

```c
#include <stdio.h>
typedef struct {
  float real;
  float imag;
} Complex;
Complex add(Complex c1, Complex c2) {
  Complex result;
  result.real = c1.real + c2.real;
  result.imag = c1.imag + c2.imag;
  return result;
}
void print(Complex c) {
  printf("%.1f + %.1fi\n",
         c.real, c.imag);
}
int main() {
  Complex c1 = {3.0, 4.0};
  Complex c2 = {1.5, 2.5};
  Complex sum = add(c1, c2);
  printf("c1 = ");
  print(c1);
  printf("c2 = ");
  print(c2);
  printf("Sum = ");
  print(sum);
  return 0;
}
```

Output:

```
c1 = 3.0 + 4.0i
c2 = 1.5 + 2.5i
Sum = 4.5 + 6.5i
```

Note:

- Complex number arithmetic
- Add real and imaginary parts
- typedef for convenience
- Functions for operations

Program 19: Student Grade System

```c
#include <stdio.h>
#include <string.h>
typedef struct {
  int roll;
  char name[30];
  float marks;
  char grade;
} Student;
void assignGrade(Student *s) {
  if (s->marks >= 90) s->grade = 'A';
  else if (s->marks >= 80) s->grade = 'B';
  else if (s->marks >= 70) s->grade = 'C';
  else if (s->marks >= 60) s->grade = 'D';
  else s->grade = 'F';
}
int main() {
  Student s = {101, "Alice", 85.5, '-'};
  assignGrade(&s);
  printf("Name: %s\n", s.name);
  printf("Marks: %.1f\n", s.marks);
  printf("Grade: %c\n", s.grade);
  return 0;
}
```

Output:

```
Name: Alice
Marks: 85.5
Grade: B
```

Note:

- Pointer to struct
- Arrow operator: -¿
- Modifies original struct
- Grade based on marks

Program 20: Library Book System

```c
#include <stdio.h>
#include <string.h>
typedef struct {
  int id;
  char title[40];
  char author[30];
  int available;
} Book;
void displayBook(Book b) {
  printf("\nID: %d\n", b.id);
  printf("Title: %s\n", b.title);
  printf("Author: %s\n", b.author);
  printf("Status: %s\n",
         b.available ? "Available" :
                       "Issued");
}
int main() {
  Book books[2] = {
    {1, "C Programming", "K&R", 1},
    {2, "Data Structures", "Tanenbaum", 0}
  };
  int i;
  for (i = 0; i < 2; i++) {
    displayBook(books[i]);
  }
  return 0;
}
```

Output:

```
ID: 1
Title: C Programming
Author: K&R
Status: Available

ID: 2
Title: Data Structures
Author: Tanenbaum
Status: Issued
```

Note:

- Array of Book structs
- Boolean-like field
- Display function
- Real-world application

Structures - Summary

Key Points:

- Structure = user-defined data type
- Groups related variables together
- Dot operator (.) for member access
- Arrow operator (-¿) for pointer to struct
- Can be passed to functions
- Can be returned from functions
- Array of structures possible
- Nested structures allowed
- typedef creates alias
- Assignment copies all members
- Cannot use == for comparison

Structure Syntax Reference

Operation	**Syntax**
Declaration	`struct Name { members; };`
Variable	`struct Name var;`
Access member	`var.member`
Pointer access	`ptr->member`
Initialize	`struct Name v = {val1, val2};`
Assignment	`var2 = var1;`
Array	`struct Name arr[10];`
typedef	`typedef struct {...} Name;`

Best Practices

1. **Use meaningful names** for structs and members
2. **Group related data** logically
3. **Use typedef** for convenience
4. **Initialize all members** at declaration
5. **Pass by pointer** for large structs
6. **Use const** for read-only struct parameters
7. **Keep structs simple** - not too many members
8. **Document member meanings** with comments
9. **Use structs for** related data, not random grouping
10. **Consider alignment** for memory efficiency

Common Mistakes

1. **Forgetting semicolon** after struct declaration
2. **Using == for comparison** - compare members instead
3. **Missing struct keyword** - use typedef to avoid
4. **Wrong member access** - dot vs arrow operator
5. **Uninitialized members** - always initialize
6. **Not passing size** with array of structs
7. **Modifying in function** - pass pointer if needed
8. **Forgetting to copy strings** - use strcpy
9. **Wrong initialization order** - must match members
10. **Returning local struct address** - return value, not pointer

Practice Exercises

Try these programs:

1. Create Date structure with validation
2. Student database with sorting
3. Employee payroll system
4. Contact list management
5. Bank account operations
6. Inventory management system
7. Circle structure with area/circumference
8. Fraction arithmetic (add, subtract, multiply)
9. Playing card deck representation
10. Polynomial representation and addition