CS 137

Structures

Fall 2025

Victoria Sakhnini

Table of Contents

Structures 2
Function returns a structure 3
Array of Structures 4
Structures Containing Structures 5
Structures Containing Arrays 6
Additional Examples 7
Extra Practice Problems 10

Structures

Previously, we've learned about Arrays in C that permit you to group elements of the same type into one logical entity. C provides another tool for grouping elements of different kinds. The structure consists of named member variables that are stored together in memory. For example, suppose we wanted to model a clock; we would want the hours and minutes. Let's call this `tod` for **t**ime **o**f **d**ay.

```
struct tod {
      int hours;
      int minutes;
      };
```

We could then use our `struct` type to create instances of the time. For example

```
struct tod now = {16,50};
```

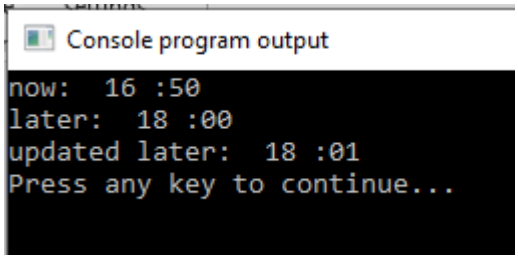
`now` is the variable name, and its members `hours` and `minutes` are initialized with values `16` and `50`, respectively

```
struct tod later = {.hours = 18};
```

what do you think is the value of the member `minutes`?[1]

Example:

```
1.  #include <stdio.h>
2.
3.  struct tod {
4.    int hours;
5.    int minutes;
6.  };
7.
8.  void todPrint(struct tod when){
9.    printf(" %0.2d :%0.2d\n", when.hours, when.minutes);
10.   }
11.
12.   int main(void){
13.           struct tod now = { 16, 50 };
14.           struct tod later = {.hours = 18 };
15.           printf("now: ");
16.           todPrint(now);
17.           printf("later: ");
18.           todPrint(later);
19.           later.minutes = 1;
20.           printf("updated later: ");
21.           todPrint(later);
22.           return 0;
23.   }
```



[1] In C there is no such thing as "partial initialization". If a struct is initialized by specifying a value for a single member, all the other members are automatically initialized (to 0 of the proper kind).

Function returns a structure

Example: The function `todAddTime` returns a value of type `struct tod`.

```
1. #include <stdio.h>
2.
3. struct tod {
4.   int hours;
5.   int minutes;
6. };
7.
8. void todPrint(struct tod when)
9. {
10.          printf(" %0.2d :%0.2d\n", when.hours, when.minutes);
11.  }
12.
13.  struct tod todAddTime(struct tod when, int hours, int minutes)
14.  {
15.          when.minutes += minutes;
16.          when.hours += hours + when.minutes / 60;
17.          when.minutes %= 60;
18.          when.hours %= 24;
19.          return when;
20.  }
21.
22.  int main(void)
23.  {
24.          struct tod now = { 16, 50 };
25.          struct tod later;
26.          printf("now: ");
27.          todPrint(now);
28.          printf("later: ");
29.          todPrint(later);     // Check the output. Why do we see those numbers?
30.                               // Because later was NOT initialized,
31.                               // thus it has arbitrary values in memory.
32.          later = todAddTime(now, 1, 10);
33.          printf("now: ");
34.          todPrint(now);  // When passing structs to functions, these values are also
35.                          // passed-by-value, which mean a copy of the value is
36.                          // passed to the parameter
37.          printf("updated later: ");
38.          todPrint(later);
39.          return 0;
40.  }
```

```
Console program output
now:  16 :50
later:  -1014577184 :32759
now:  16 :50
updated later:  18 :00
Press any key to continue...
```

If you wanted to modify the original struct in memory, you would need to pass a pointer to it and then modify the contents of the pointers (more on this later).

To make life easier, we can use a `typedef` to help create structures:

```
1. #include <stdio.h>
2.
3. typedef struct {
4.   int hours ;
5.   int minutes ;
6. } tod;
7.
8. int main(void) {
9.   tod now = {14 ,40};        // instead of struct tod
10.  return 0;
11.  }
```

Array of Structures

There is nothing new here; thus, I am just providing an example to introduce the syntax.

```
1. #include <stdio.h>
2.
3. struct student {
4.   int id;
5.   int assgn;
6.   int mid;
7.   int fexam;
8.   int fgrade;
9. };
10.
11.  void calc_fgrade(struct student a[], int n)
12.  {
13.          for (int i = 0; i < n; i++)
14.                  a[i].fgrade = 0.2 * a[i].assgn + 0.3 * a[i].mid + 0.5 * a[i].fexam;
15.
16.  }
17.
18.  int main(void)
19.  {
20.          struct student cs137[5] = { {1111, 80, 88, 78}, {2222, 77, 90, 81},
21.          {3333, 67, 66, 78}, {4444, 90, 100, 98},
22.          {5555, 88, 77, 84}
23.          };
24.          calc_fgrade(cs137, 5);
25.          for (int i = 0; i < 5; i++)
26.               printf("student %d, final-grade=%d\n", cs137[i].id, cs137[i].fgrade);
27.          return 0;
28.  }
```

Console program output

```
student 1111, final-grade=81
student 2222, final-grade=82
student 3333, final-grade=72
student 4444, final-grade=97
student 5555, final-grade=82
Press any key to continue...
```

`cs137` is an array of length `5` (for simplicity) where each element is of type `struct student`, thus `cs137[i]` is a structure.

For simplicity, the array was initialized with the data rather than reading the data.

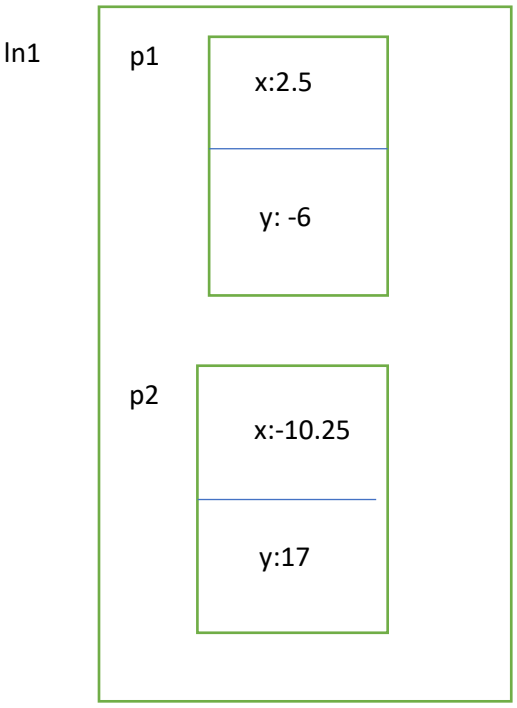
Structures Containing Structures

There is nothing new here; thus, I am just providing an example to introduce the syntax.

```
1. #include <stdio.h>
2.
3. typedef struct {
4.   double x;
5.   double y;
6. } point;
7.
8. typedef struct {
9.   point p1;
10.  point p2;
11.  } line;
12.
13.  int main(void){
14.          line ln1 = { {2.5, -6}, {-10.25, 17} };
15.          double a, b;
16.          a = (ln1.p1.y - ln1.p2.y) / (ln1.p1.x - ln1.p2.x);
17.          b = ln1.p1.y - a * ln1.p1.x;
18.          printf("y = %fx + %f\n", a, b);
19.          return 0;
20.  }
```

```
Console program output
y = -1.803922x + -1.490196
Press any key to continue...
```

`ln1` represents a line defined by two points. (`ln1.p1` is of type `struct point`, same for `ln1.p2`)

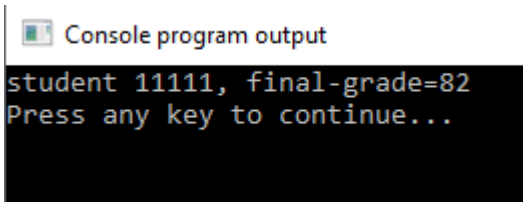


The output is the same line presented as `y=ax+b`

Structures Containing Arrays

There is nothing new here; thus, I am just providing an example to introduce the syntax.

```c
#include <stdio.h>

typedef struct {
  int id;
  int assgn[10];
  int mid;
  int fexam;
} student;

int main(void)
{
        student stud = { 11111,
                {80, 90, 88, 99, 77, 78, 76, 89, 90, 100}, 88, 78
        };
        int fgrade, sum = 0;
        for (int i = 0; i < 10; i++)
                sum += stud.assgn[i];
        fgrade = 0.2 * sum / 10 + 0.3 * stud.mid + 0.5 * stud.fexam;
        printf("student %d, final-grade=%d\n", stud.id, fgrade);
        return 0;
}
```



Note:

`stud.id` is `11111`

`stud.ass` is an array of length `10` with the values: `{80,90,……..,100}`

`stud.mid` is `88`

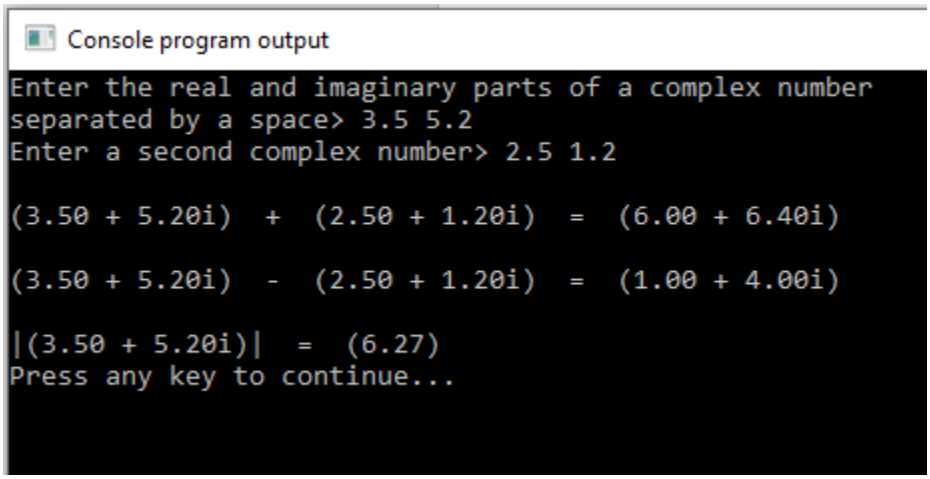
`stud.fexam` is `78`

Additional Examples

```c
/* Operators to process complex numbers */

#include <stdio.h>
#include <math.h>

/*  User-defined complex number type */
typedef struct {
        double real, imag;
} complex_t;

int scan_complex(complex_t *c);  // pointer to structure. Motivation for next chapter.
void print_complex(complex_t c);
complex_t add_complex(complex_t c1, complex_t c2);
complex_t subtract_complex(complex_t c1, complex_t c2);
complex_t multiply_complex(complex_t c1, complex_t c2);
complex_t divide_complex(complex_t c1, complex_t c2);
complex_t abs_complex(complex_t c);

int main(void)
{
        complex_t com1, com2;

        /*  Gets two complex numbers  */
        printf("Enter the real and imaginary parts of a complex number\n");
        printf("separated by a space> ");
        scan_complex(&com1);
        printf("Enter a second complex number> ");
        scan_complex(&com2);

        /*  Forms and displays the sum  */
        printf("\n");
        print_complex(com1);
        printf("  +  ");
        print_complex(com2);
        printf("  =  ");
        print_complex(add_complex(com1, com2));

        /*  Forms and displays the difference */
        printf("\n\n");
        print_complex(com1);
        printf("  -  ");
        print_complex(com2);
        printf("  =  ");
        print_complex(subtract_complex(com1, com2));

        /*  Forms and displays the absolute value of the first number */
        printf("\n\n|");
        print_complex(com1);
        printf("|  =  ");
        print_complex(abs_complex(com1));
        printf("\n");

        return (0);
}
```

```c
/*  Complex number input function returns standard scanning error code
 *    1 => valid scan, 0 => error, negative EOF value => end of file   */

int scan_complex(complex_t *c)   /* output - address of complex variable to fill */
{
        int status;
        status = scanf("%lf%lf", &c->real, &c->imag);
        if (status == 2)
                status = 1;
        else if (status != EOF)
                status = 0;
        return (status);
}

/*  Complex output function displays value as (a + bi) or (a - bi),
 *  dropping a or b if they round to 0 unless both round to 0   */

void print_complex(complex_t c)  /* input - complex number to display */
{
        double a, b;
        char sign;
        a = c.real;
        b = c.imag;
        printf("(");
        if (fabs(a) < .005 && fabs(b) < .005)
        {
                printf("%.2f", 0.0);
        }
        else if (fabs(b) < .005)
        {
                printf("%.2f", a);
        }
        else if (fabs(a) < .005)
        {
                printf("%.2fi", b);
        }
        else
        {
                if (b < 0)
                        sign = '-';
                else
                        sign = '+';
                printf("%.2f %c %.2fi", a, sign, fabs(b));
        }
        printf(")");
}

/*  Returns sum of complex values c1 and c2  */

complex_t add_complex(complex_t c1, complex_t c2) /* input - values to add    */
{
        complex_t csum;
        csum.real = c1.real + c2.real;
        csum.imag = c1.imag + c2.imag;
        return (csum);
}
```

```c
/*  Returns difference c1 - c2   */

complex_t subtract_complex(complex_t c1, complex_t c2)     /* input parameters    */
{
        complex_t cdiff;
        cdiff.real = c1.real - c2.real;
        cdiff.imag = c1.imag - c2.imag;
        return (cdiff);
}

/* Returns product of complex values c1 and c2  */

complex_t multiply_complex(complex_t c1, complex_t c2)     /* input parameters    */
{
        // try to complete this function and use it in main
}

/*  Returns quotient of complex values (c1 / c2) */

complex_t divide_complex(complex_t c1, complex_t c2)      /* input parameters  */
{
        // try to complete this function and use it in main}

/* Returns absolute value of complex number c */

complex_t abs_complex(complex_t c)        /* input parameter */
{
        complex_t cabs;
        cabs.real = sqrt(c.real * c.real + c.imag * c.imag);
        cabs.imag = 0;
        return (cabs);
}
```



Extra Practice Problems

Consider the following definitions for questions 1-3:

```
struct point{
    int x;
    int y;
};
```

1) Write the C function `scale_point`[i] that takes `struct point` parameter `p` and an integer `f`, and returns a new `struct point` value as `p` after multiplying by `f`.

2) Write the C function `distance` that takes two `struct point` parameters `p1` and `p2`, and returns the squared distance between `p1` and `p2`.

3) Write the C function `find` that reads pairs of integers where each pair represents a point, and returns the point with the largest distance from `(0,0)`. If there is more than one, it returns the one that was read last. You should not use float or double in this question.

Here is the main function for testing:

```
int main(void){
      struct point p1 = {5,10};
      struct point p2 = {0,0};
      struct point p3 = {-2,0};
      struct point pt;
      pt = scale_point(p1, 2);
      assert(pt.x==10);
      assert(pt.y==20);
      pt = scale_point(p3, -1);
      assert(pt.x==2);
      assert(pt.y==0);
      assert(distance(p1,p1)==0);
      assert(distance(p1,p2)==125);

      pt = find();  // as input enter -2 0  5 10
      assert(pt.x==p1.x);
      assert(pt.y==p1.y);
      return 0;
}
```

4) Write a function that has two parameters, an array of student structures and the length of the array, and prints the `id` of every student who failed the midterm, and how much they need to score in the final to pass the course.

```
struct student{
      int id;
      int assgn;
      int mid;
      int fexam;
      int fgrade;
};
```

(consider the marking scheme: assign 30%, midterm 20%, final 50%).

5) You are tasked with creating an inventory management system for a large retail company that operates in multiple locations. The company has a vast inventory of various products, each with a unique item ID, a name, a detailed description, a supplier name, a purchase price, a selling price, and the quantity in stock at multiple store locations. Your task is to create a C program that effectively manages this complex inventory system using structures.

Your program should provide the following functionalities:

a) Add a new product to the inventory, specifying its item ID, name, description, supplier name, purchase price, and the initial quantity in stock for a specific store location.

b) Update an existing product's purchase or selling price by searching for it using the item ID and specifying the store location.

c) Print a list of all products in the inventory at a specific store location.

d) Find and print the product with the highest profit margin (selling price minus purchase price) across all store locations.

e) Find and print the product with the lowest quantity in stock across all store locations.

Your program should also provide an option to switch between store locations, and each store location can have its separate inventory of products.

Complete the program to solve the problem above.

```c
#include <stdio.h>
#include <string.h>
 #define MAX_PRODUCTS 500
#define MAX_LOCATIONS 10
struct Product {
    int itemID;
    char name[100];
    char description[200];
    char supplierName[100];
    float purchasePrice;
    float sellingPrice;
    int quantity;
};

struct StoreLocation {
    char locationName[50];
    struct Product products[MAX_PRODUCTS];
    int productCount;
};
struct InventorySystem {
```

```c
    struct StoreLocation locations[MAX_LOCATIONS];
    int locationCount;
    int currentLocation;
};

// Function to add a new product to the inventory
void addProduct(struct InventorySystem* inventory, int itemID, const char* name, const char* description,
                const char* supplierName, float purchasePrice, float sellingPrice, int quantity) { }
// Function to update purchase price or selling price of an existing product
void updateProductPrice(struct InventorySystem* inventory, int itemID, float newPrice, int locationIndex, int isPurchasePrice) { }
// Function to print a list of all products in a specific location
void listProductsInLocation(struct InventorySystem* inventory, int locationIndex) { }
// Function to find and print the product with the highest profit margin
void findProductWithHighestProfitMargin(struct InventorySystem* inventory) { }
 // Function to find and print the product with the lowest quantity in stock
void findProductWithLowestQuantity(struct InventorySystem* inventory) { }
 int main() {
    struct InventorySystem inventory;
    inventory.locationCount = 0;
    inventory.currentLocation = 0;

    // Implement the menu-driven program for managing the complex inventory system using structures.
    int choice;

    while (1) {
        printf("Current Location: %s\n", inventory.locations[inventory.currentLocation].locationName);
        printf("1. Add a new product\n");
        printf("2. Update purchase price of an existing product\n");
        printf("3. Update selling price of an existing product\n");
        printf("4. Print list of products in the current location\n");
        printf("5. Switch to a different location\n");
        printf("6. Find product with the highest profit margin\n");
        printf("7. Find product with the lowest quantity in stock\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Add a new product
                int itemID, quantity;
                float purchasePrice, sellingPrice;
                char name[100], description[200], supplierName[100];
                printf("Enter Item ID: ");
                scanf("%d", &itemID);
                printf("Enter Name: ");
                scanf("%s", name);
                printf("Enter Description: ");
                scanf("%s", description);
                printf("Enter Supplier Name: ");
                scanf("%s", supplierName);
                printf("Enter Purchase Price: ");
                scanf("%f", &purchasePrice);
                printf("Enter Selling Price: ");
                scanf("%f", &sellingPrice);
                printf("Enter Quantity: ");
                scanf("%d", &quantity);
                addProduct(&inventory, itemID, name, description, supplierName, purchasePrice, sellingPrice, quantity);
                break;
            case 2:
                // Update purchase price of an existing product
                int updateID;
                float newPurchasePrice;
                printf("Enter Item ID to update: ");
                scanf("%d", &updateID);
                printf("Enter new Purchase Price: ");
```

```
                scanf("%f", &newPurchasePrice);
                updateProductPrice(&inventory, updateID, newPurchasePrice, inventory.currentLocation, 1); // 1 indicates purchase price
                break;
            case 3:
                // Update selling price of an existing product
                int updateSellingID;
                float newSellingPrice;
                printf("Enter Item ID to update: ");
                scanf("%d", &updateSellingID);
                printf("Enter new Selling Price: ");
                scanf("%f", &newSellingPrice);
                updateProductPrice(&inventory, updateSellingID, newSellingPrice, inventory.currentLocation, 0); // 0 indicates selling price
                break;
            case 4:
                // Print list of products in the current location
                listProductsInLocation(&inventory, inventory.currentLocation);
                break;
            case 5:
                // Switch to a different location
                int newLocation;
                printf("Enter the location index to switch to: ");
                scanf("%d", &newLocation);
                if (newLocation >= 0 && newLocation < inventory.locationCount) {
                    inventory.currentLocation = newLocation;
                } else {
                    printf("Invalid location index.\n");
                }
                break;
            case 6:
                // Find product with the highest profit margin
                findProductWithHighestProfitMargin(&inventory);
                break;
            case 7:
                // Find product with the lowest quantity in stock
                findProductWithLowestQuantity(&inventory);
                break;
            case 8:
                // Exit the program
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

6) Below is the Date structure:

```
struct Date {
	int day;
	int month;
	int year;
};
```

Complete the following:

- `struct Date date_create(int d, int m, int y)` : returns a `Date` struct with day `d`, month `m`, and year `y`.
- `bool date_eq(struct Date d, struct Date other)`: Dates should be equal if and only if they represent exactly the same date (year, month, and day).
- `struct Date add_days(struct d, int days)`: consumes a positive integer number of days and returns the Date that is precisely that many days after the current date d.

Reminder: 30 days in September, April, June, and November. All the rest have 31... except February, which has 28, or 29 if we are in a leap year. We've provided a function to check if a year is a leap year.

Here are sample starter tests:

```
1. int main() {
2.     struct Date d1 = date_create(1, 1, 2000);
3.     struct Date d2 = date_create(2, 1, 2000);
4.     assert(!date_eq(d1, d2));
5.     d1 = add_days(d1, 130);
6.     d2 = add_days(d2, 129);
7.     assert(date_eq(d1, d2));
8. }
9.
```

7) **Managing Student Records**

You are tasked with creating a C program to manage student records. The program should allow users to input data for a variable number of students and perform several operations on the data. Below are the tasks your program should accomplish:

1. **Input:** Read data for n students from the user (n ≤ 100). For each student, you need to input their ID (integer), exam score (float), and enrollment status (a string that can be "enrolled" or "dropped").
2. **Display:** Display the information of all students, including their IDs, exam scores, and enrollment status.
3. **Average Exam Score:** Calculate and display the average exam score of all enrolled students.
4. **Highest Scorer:** Find and display the details of the student with the highest exam score.
5. **Enrolled Students:** Display the number of students currently enrolled in the course.

6. **Update Enrollment Status:** Allow users to update the enrollment status of a specific student by providing their ID.

Your task is to create a C program that accomplishes these tasks using appropriate functions, data structures, and control structures.

Requirements:

- Use functions to perform each of the tasks mentioned above.
- Ensure that your program handles data for up to 100 students.
- Display appropriate error messages if the user enters invalid data or the requested student is not found.

Sample Output:

Welcome to the Student Record Management System!

1. Input Student Data
2. Display Student Data
3. Calculate Average Exam Score
4. Find Highest Scorer
5. Count Enrolled Students
6. Update Enrollment Status
7. Exit

Please enter your choice: 1

Enter the number of students (up to 100): 3
Enter the details of 3 students:
Student 1:
ID: 1001
Exam Score: 85.5
Enrollment Status: enrolled
Student 2:
ID: 1002
Exam Score: 78.0
Enrollment Status: enrolled
Student 3:
ID: 1003
Exam Score: 92.5
Enrollment Status: dropped

Please enter your choice: 2
Student Details:
Student 1:
ID: 1001
Exam Score: 85.50
Enrollment Status: enrolled
Student 2:
ID: 1002
Exam Score: 78.00
Enrollment Status: enrolled
Student 3:

ID: 1003
Exam Score: 92.50
Enrollment Status: dropped

Please enter your choice: 3
Average Exam Score of Enrolled Students: 81.75

Please enter your choice: 4
Highest Scorer:
ID: 1003
Exam Score: 92.50
Enrollment Status: dropped

Please enter your choice: 5
Enrolled Students: 2

Please enter your choice: 6
Enter the ID of the student whose enrollment status you want to update: 1002
Enter the new enrollment status: enrolled
Enrollment status updated successfully.

Please enter your choice: 7
Goodbye!

8) Suppose we are given a definition for a point structure as follows:

```
struct point {
int x;
int y;
}
```

Write a program `bool colinear(struct point* points, size t len)` that takes an array of points and the array's length and returns `true` if and only if all points lie on the same line; otherwise, `false`.

Answers

---

i

```
struct point scale_point(struct point p, int f)
{
    struct point p2;
    p2.x = p.x * f;
    p2.y = p.y * f;
    return p2;
}
```