



Structures Lesson Outline

1. Structures Lesson Outline
2. Beyond Arrays
3. A Company and Its Employees #1
4. A Company and Its Employees #2
5. Multiple Employees #1
6. Multiple Employees #2
7. Multiple Employees #3
8. A New Data Type #1
9. A New Data Type #2
10. A New Data Type #3
11. Structure Definition Breakdown
12. Structure Instance Declaration #1
13. Structure Instance Declaration #2
14. Structure Instance Declaration #3
15. Structure Instance Declaration #4
16. Fields of a Structure Instance #1
17. Fields of a Structure Instance #2
18. Fields of a Structure Instance #3
19. Fields of a Structure Instance #4
20. Structure Fields Like Array Elements #1
21. Structure Fields Like Array Elements #2
22. Structure Example #1
23. Structure Example #2
24. Structure Example #3
25. Structure Example #4
26. Structure Example #5
27. Structure Array
28. Structure Array: Static vs Dynamic
29. Structure Array: Dynamic Allocation
30. Structure Array: Indexing
31. Structure Array: Element's Field Access
32. Structure Array Example #1
33. Structure Array Example #2
34. Structure Array Example #3
35. Structure Array Example #4
36. Structure Array Example #5
37. Structure Array Example #6
38. Structure Array Example #7
39. Structure Array Example #8
40. Structure Array Example #9



Beyond Arrays

An array is a collection of values, all of which have the same data type, and all of which have the same essential meaning:

```
int OU_score[12];
```

In memory, the elements of the array are contiguous: they occur one after the other in memory.

37	37	68	31	31	35	49	27	26	49	60	28
----	----	----	----	----	----	----	----	----	----	----	----

What if, instead of having a collection of data that all have the same data type and meaning, we had a collection of data that had different data types and different meanings?





A Company and Its Employees #1

Suppose that we work for the Depressingly Dull Corporation (DDC), and our boss tells us to write a program that tracks DDC's employees.

What data will we need?

Well, we'll probably need to know things like:

- first name;
- last name;
- pay rate;
- number of hours worked this week;
- social security number.

How could we implement this in C?





A Company and Its Employees #2

How could we implement this in C?

Well, we could simply set up a scalar variable to represent each of these values (and strings for the names):

```
char* first_name;  
char* last_name;  
float pay_rate;  
float hours_worked_this_week;  
int    social_security_number;
```

Of course, this arrangement would work if our company had exactly one employee. But what if our company has **multiple employees?**





Multiple Employees #1

Okay, so suppose that DDC has multiple employees. How could we store the data for them?

Well, we could have an array for each of the pieces of data:

```
char* first_name[number_of_employees];  
char* last_name[number_of_employees];  
float pay_rate[number_of_employees];  
float hours_worked_this_week[number_of_employees];  
int    social_security_number[number_of_employees];
```





Multiple Employees #2

```
char* first_name[number_of_employees];  
char* last_name[number_of_employees];  
float pay_rate[number_of_employees];  
float hours_worked_this_week[number_of_employees];  
int    social_security_number[number_of_employees];
```

This approach will work fine, but it'll be unwieldy to work with. Why? Because it doesn't match the way that we **think** about our employees.

That is, we don't think of having several first names, several last names, several social security numbers and so on; we have several **employees**.





Multiple Employees #3

We don't think of having several first names, several last names, several social security numbers and so on.

Instead, we think of having **several employees**, each of whom has a first name, a last name, a social security number, etc.

In general, it's much easier to write a program if we can write it in a way that matches the way we think as much as possible.

So: What if we could **create a new data type**, named **Employee**, that represented an employee?





A New Data Type #1

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;
```

The above declaration **creates a new data type**, named Employee.

This is known as a **user-defined data type** or a **user-defined data structure**.





A New Data Type #2

```
typedef struct {
    char* first_name;
    char* last_name;
    float pay_rate;
    float hours_worked_this_week;
    int    social_security_number;
} Employee;
```

The ***user-defined data type*** Employee consists of:

- a character string, `first_name`;
- a character string, `last_name`;
- a float scalar, `pay_rate`;
- a float scalar, `hours_worked_this_week`;
- an int scalar, `social_security_number`.





A New Data Type #3

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;
```

In C, this construct is referred to as a *structure definition*, and (surprise!) it defines a *structure*.

The general term for this is a *user-defined data type*.

NOTE: A structure definition, as above, only **defines** the new data type; it **DOESN'T DECLARE** any actual instances of data of the new data type.





Structure Definition Breakdown

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;
```

A structure definition consists of:

- a `typedef struct` statement and block open `{`;
- a sequence of ***field*** definitions, which tell us the pieces of data that constitute an instance of the structure;
- a block close `}` and the name of the structure, followed by a statement terminator.





Structure Instance Declaration #1

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;
```

The above struct definition defines the struct named Employee, but **DOESN'T DECLARE** any instance of data whose data type is Employee.





Structure Instance Declaration #2

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;
```

To **declare** an **instance** of an Employee, we need to do like so:

```
Employee worker_bee;
```





Structure Instance Declaration #3

```
typedef struct {
    char* first_name;
    char* last_name;
    float pay_rate;
    float hours_worked_this_week;
    int    social_security_number;
} Employee;
Employee worker_bee;
```

The last statement above **declares** that `worker_bee` is an **instance** of type `Employee`; it tells the compiler to grab a group of bytes, name them `worker_bee`, and think of them as storing an `Employee`.





Structure Instance Declaration #4

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;  
Employee worker_bee;
```

How many bytes?

That depends on the platform and the compiler, but the short answer is that it's the sum of the sizes of the fields.





Fields of a Structure Instance #1

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;
```

```
Employee worker_bee;
```

Okay, so now we have an instance of data type `Employee` named `worker_bee`.

But how can we use the values of its field data? For example, how do we get the social security number of `worker_bee`?





Fields of a Structure Instance #2

```
typedef struct {
    char* first_name;
    char* last_name;
    float pay_rate;
    float hours_worked_this_week;
    int    social_security_number;
} Employee;
```

```
Employee worker_bee;
```

To use an individual field of a struct, we use the *field operator*, which is the **period**:

```
worker_bee.social_security_number
```





Fields of a Structure Instance #3

```
typedef struct {  
    char* first_name;  
    char* last_name;  
    float pay_rate;  
    float hours_worked_this_week;  
    int    social_security_number;  
} Employee;
```

```
Employee worker_bee;
```

For example, we can assign a value to the social security number of worker_bee:

```
worker_bee.social_security_number = 123456789;
```

This is equivalent to using an index in an array:

```
OU_score[element] = 101010101;
```





Fields of a Structure Instance #4

```
typedef struct {
    char* first_name;
    char* last_name;
    float pay_rate;
    float hours_worked_this_week;
    int    social_security_number;
} Employee;
Employee worker_bee;
```

Likewise, we can output the social security number of
worker_bee:

```
printf("%d\n", worker_bee.social_security_number);
```

This is equivalent to using an index in an array:

```
printf("%d\n", OU_score[element]);
```



Structure Fields Like Array Elements #1

We said that we can use the *field operator* (period) to get an individual field of an instance of a struct:

```
worker_bee.social_security_number = 123456789;  
printf("%d\n", worker_bee.social_security_number);
```

Notice that this usage is **analogous** to the use of an index with an array:

```
OU_score[element] = 123456789;  
printf("%d\n", OU_score[element]);
```





Structure Fields Like Array Elements #2

In the case of arrays, we said that an individual element of an array behaves exactly like a scalar of the same data type. Likewise, a field of a `struct` behaves exactly like a variable of the same data type as the field.

So, `worker_bee.social_security_number` can be used **exactly** like an `int` scalar, and `worker_bee.first_name` can be used **exactly** like a character string.





Structure Example #1

```
#include <stdio.h>

int main ()
{ /* main */
    typedef struct {
        char* first_name;
        char* last_name;
        float pay_rate;
        float hours_worked_this_week;
        int    social_security_number;
    } Employee;
    const int maximum_name_length = 32;
    Employee worker_bee;
    char dummy_name[maximum_name_length + 1];
    float worker_bee_pay;
```



Structure Example #2

Huh?

```
printf("What is the first name of the employee?\n");
fgets(dummy_name, maximum_name_length, stdin);
if (dummy_name[strlen(dummy_name)-1] == '\n') {
    dummy_name[strlen(dummy_name)-1] = '\0';
} /* if (dummy_name[strlen(dummy_name)-1]=='\n') */
worker_bee.first_name =
    (char*)malloc(sizeof(char) *
                  (strlen(dummy_name) + 1));
strcpy(worker_bee.first_name, dummy_name);
printf("What is the last name of the employee?\n");
fgets(dummy_name, maximum_name_length, stdin);
if (dummy_name[strlen(dummy_name)-1] == '\n') {
    dummy_name[strlen(dummy_name)-1] = '\0';
} /* if (dummy_name[strlen(dummy_name)-1]=='\n') */
worker_bee.last_name =
    (char*)malloc(sizeof(char) *
                  (strlen(dummy_name) + 1));
```





Structure Example #3

```
strcpy(worker_bee.last_name, dummy_name);
printf("What is %s %s's pay rate in $/hour?\n",
       worker_bee.first_name, worker_bee.last_name);
scanf("%f", &worker_bee.pay_rate);
printf("How many hours did %s %s work this week?\n",
       worker_bee.first_name, worker_bee.last_name);
scanf("%f", &worker_bee.hours_worked_this_week);
printf("What is %s %s's social security number?\n",
       worker_bee.first_name, worker_bee.last_name);
scanf("%d", &worker_bee.social_security_number);
```





Structure Example #4

```
worker_bee_pay =
    worker_bee.pay_rate *
    worker_bee.hours_worked_this_week;
printf("Employee %s %s (%9.9d)\n",
    worker_bee.first_name,
    worker_bee.last_name,
    worker_bee.social_security_number);
printf("    worked %2.2f hours this week\n",
    worker_bee.hours_worked_this_week);
printf("    at a rate of $%2.2f per hour,\n",
    worker_bee.pay_rate);
printf("    earning $%2.2f.\n", worker_bee_pay);
} /* main */
```





Structure Example #5

```
% gcc -o employee_test employee_test.c
```

```
% employee_test
```

```
What is the first name of the employee?
```

```
Henry
```

```
What is the last name of the employee?
```

```
Neeman
```

```
What is Henry Neeman's pay rate in $/hour?
```

```
12.5
```

```
How many hours did Henry Neeman work this week?
```

```
22.75
```

```
What is Henry Neeman's social security number?
```

```
123456789
```

```
Employee Henry Neeman (123456789)
```

```
worked 22.75 hours this week
```

```
at a rate of $12.50 per hour,
```

```
earning $284.38.
```





Structure Array

When we started working on this task, we wanted to work out a convenient way to store the many employees of the Depressingly Dull Corporation (DDC).

So far, we've worked out how to defined a structure, how to declare an individual instance of the `struct`, and how to use the fields of the instance.

So, how would we declare and use an **array** of instances of a `struct`?

```
Employee worker_bee_array[maximum_employees];
```





Structure Array: Static vs Dynamic

```
Employee worker_bee_array[maximum_employees];
```

Not surprisingly, an array whose elements are a struct can either be declared to be statically allocated (above) or dynamically allocatable (below):

```
Employee* worker_bee_array2 =  
    (Employee*)NULL;
```





Structure Array: Dynamic Allocation

```
Employee* worker_bee_array2 =  
    (Employee*)NULL;
```

If a struct array is declared to be dynamically allocatable, then allocating it looks just like allocating an array of a scalar data type:

```
worker_bee_array2 =  
    (Employee*)malloc(sizeof(Employee) *  
        number_of_employees);
```





Structure Array: Indexing

An individual element of an array of some `struct` data type can be accessed using indexing, exactly as if it were an element of an array of scalar data type:

```
worker_bee_array[index]
```





Structure Array: Element's Field Access

Fields of an individual element of an array of a struct data type can be accessed thus:

```
worker_bee_array[index].pay_rate
```

For example:

```
worker_bee_array[index].pay_rate = 6.50;
```

```
printf("%f\n", worker_bee_array[index].pay_rate);
```





Structure Array Example #1

```
#include <stdio.h>

int main ()
{ /* main */
    typedef struct { char* first_name;
                    char* last_name;
                    float pay_rate;
                    float hours_worked_this_week;
                    int  social_security_number;
    } Employee;
    const int maximum_name_length = 32;
    const int error_exit_code     = -1;
    Employee* worker_bee         = (Employee*)NULL;
    float*    worker_bee_pay     = (float*)NULL;
    char      dummy_name[maximum_name_length + 1];
    int       number_of_worker_bees, index;
```



Structure Array Example #2

```
printf("How many employees does the company have?\n");
scanf("%d", &number_of_worker_bees);
worker_bee =
    (Employee*)malloc(sizeof(Employee) *
                      number_of_worker_bees);
if (worker_bee == (Employee*)NULL) {
    printf("ERROR: can't allocate worker_bee array ");
    printf("of length %d Employees\n",
          number_of_worker_bees);
    exit(error_exit_code);
} /* if (worker_bee == (Employee*)NULL) */
worker_bee_pay = (float*)malloc(sizeof(float) *
                                number_of_worker_bees);
if (worker_bee_pay == (float*)NULL) {
    printf("ERROR: can't allocate worker_bee_pay ");
    printf("array of length %d floats",
          number_of_worker_bees);
    exit(error_exit_code);
} /* if (worker_bee_pay == (float*)NULL) */
```





Structure Array Example #3

```
for (index = 0;
     index < number of worker bees; index++) {
    /* I DO NOT UNDERSTAND WHY THIS IS NEEDED! */
    getchar();
    printf("What is the first name of ");
    printf("employee #%d?\n", index);
    fgets(dummy_name, maximum_name_length, stdin);
    if (dummy_name[strlen(dummy_name)-1] == '\n') {
        dummy_name[strlen(dummy_name)-1] = '\0';
    } /* if (dummy_name[strlen(dummy_name)-1]...) */
    worker_bee[index].first_name =
        (char*)malloc(sizeof(char) *
                      (strlen(dummy_name) + 1));
    strcpy(worker_bee[index].first_name,
           dummy_name);
}
```





Structure Array Example #4

```
printf("What is the last name of ");
printf("employee #%d?\n", index);
fgets(dummy_name, maximum_name_length, stdin);
if (dummy_name[strlen(dummy_name)-1] == '\n') {
    dummy_name[strlen(dummy_name)-1] = '\0';
} /* if (dummy_name[strlen(dummy_name)-1]...) */
worker_bee[index].last_name =
    (char*)malloc(sizeof(char) *
                  (strlen(dummy_name) + 1));
strcpy(worker_bee[index].last_name,
        dummy_name);
```





Structure Array Example #5

```
printf("What is %s %s's pay rate in $/hour?\n",
      worker_bee[index].first_name,
      worker_bee[index].last_name);
scanf("%f", &worker_bee[index].pay_rate);
printf("How many hours did %s %s work ",
      worker_bee[index].first_name,
      worker_bee[index].last_name);
printf("this week?\n");
scanf("%f",
      &worker_bee[index].hours_worked_this_week);
printf("What is %s %s's ",
      worker_bee[index].first_name,
      worker_bee[index].last_name);
printf("social security number?\n");
scanf("%d",
      &worker_bee[index].social_security_number);
} /* for index */
```





Structure Array Example #6

```
for (index = 0;
     index < number_of_worker_bees; index++) {
    worker_bee_pay[index] =
        worker_bee[index].pay_rate *
        worker_bee[index].hours_worked_this_week;
} /* for index */
```





Structure Array Example #7

```
for (index = 0;
     index < number_of_worker_bees; index++) {
    printf("Employee %s %s (%9.9d)\n",
          worker_bee[index].first_name,
          worker_bee[index].last_name,
          worker_bee[index].social_security_number);
    printf("  worked %2.2f hours this week\n",
          worker_bee[index].hours_worked_this_week);
    printf("  at a rate of $%2.2f per hour,\n",
          worker_bee[index].pay_rate);
    printf("  earning $%2.2f.\n",
          worker_bee_pay[index]);
} /* for index */
} /* main */
```





Structure Array Example #8

```
% gcc -o employee_array_test employee_array_test.c
% employee_array_test
How many employees does the company have?
2
What is the first name of employee #0?
Henry
What is the last name of the employee #0?
Neeman
What is Henry Neeman's pay rate in $/hour?
12.5
How many hours did Henry Neeman work this week?
22.75
What is Henry Neeman's social security number?
123456789
```



Structure Array Example #9

What is the first name of employee #1?

Lee

What is the last name of the employee #1?

Kim

What is Lee Kim's pay rate in \$/hour?

8.75

How many hours did Lee Kim work this week?

40

What is Lee Kim's social security number?

987654321

Employee Henry Neeman (123456789)

worked 22.75 hours this week
at a rate of \$12.50 per hour,
earning \$284.38.

Employee Lee Kim (987654321)

worked 40.00 hours this week
at a rate of \$8.75 per hour,
earning \$350.00.

