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:

```c
int overall_percentage[12];
```

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>37</td>
<td>overall_percentage[0]</td>
</tr>
<tr>
<td>12349</td>
<td>37</td>
<td>overall_percentage[1]</td>
</tr>
<tr>
<td>12353</td>
<td>68</td>
<td>overall_percentage[2]</td>
</tr>
<tr>
<td>12357</td>
<td>31</td>
<td>overall_percentage[3]</td>
</tr>
<tr>
<td>12361</td>
<td>31</td>
<td>overall_percentage[4]</td>
</tr>
<tr>
<td>12365</td>
<td>35</td>
<td>overall_percentage[5]</td>
</tr>
<tr>
<td>12369</td>
<td>49</td>
<td>overall_percentage[6]</td>
</tr>
<tr>
<td>12373</td>
<td>27</td>
<td>overall_percentage[7]</td>
</tr>
<tr>
<td>12377</td>
<td>26</td>
<td>overall_percentage[8]</td>
</tr>
<tr>
<td>12381</td>
<td>49</td>
<td>overall_percentage[9]</td>
</tr>
<tr>
<td>12385</td>
<td>60</td>
<td>overall_percentage[10]</td>
</tr>
<tr>
<td>12389</td>
<td>28</td>
<td>overall_percentage[11]</td>
</tr>
</tbody>
</table>

So, if the address of the entire array is 12345, and the length of an `int` is 4 bytes, then the addresses of the elements are:

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 meanings?
A Company and Its Employees
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?

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

Well, we could have an array of 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];
```

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.

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

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, consisting 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.

In C, this construct is referred to as a structure definition, and (not surprisingly) it defines a structure.

Other languages use different terms: in Fortran 90 the same construct is referred to as a derived type definition, in Pascal it’s a record definition, and so on.

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

NOTE: a structure definition, as above, only defines the new data type; it does NOT declare any actual instances of data of the new data type.

Breaking Down a Structure Definition

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 three parts:
- a typedef struct statement, which consists of the keywords typedef struct;
- a block open {
- a sequence of field definitions, which tell us the pieces of data that constitute an instance of the structure;
- a block close };
- the name of the user-defined structure.
Declaring an Instance of a Structure

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 does not declare any instance of data of the Employee data type.

To declare an instance of an Employee, we need a structure declaration:

Employee worker_bee;

The above statement 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.

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.

Extracting Fields from an Instance of a Structure

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 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?

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

worker_bee.social_security_number

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

worker_bee.social_security_number = 123456789;

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

printf("%d\n",
    worker_bee.social_security_number);
Extracting Fields from an Instance of a Structure (continued)

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

Employee worker_bee;

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 array indices:

overall_percentage[0] = 97.5;

printf("%f\n", overall_percentage[0]);

In the case of arrays, we said that an individual element of an array behaves exactly like a scalar of the same basic type.

Likewise, a field of a struct behaves exactly like a variable of the same type as the field.

Thus, 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 Program

#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;
    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';
    else if (dummy_name[strlen(dummy_name)-1] == '\n') /* if (dummy_name[strlen(dummy_name)-1] == '\n') */
        worker_bee.first_name = 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';
    else if (dummy_name[strlen(dummy_name)-1] == '\n') /* if (dummy_name[strlen(dummy_name)-1] == '\n') */
        worker_bee.last_name = malloc(sizeof(char) * (strlen(dummy_name) + 1));
        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);
    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.pay_rate);
    printf(" at a rate of $%2.2f per hour,\n", 
        worker_bee_pay);
    printf(" earning $%2.2f.\n", worker_bee_pay);
} /* main */
Structure Example Run

```bash
% 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.
```

An Array of Instances of a Structure

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 define 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?

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

Not surprisingly, an array whose elements are a `struct` can also be declared to be dynamically allocatable:

```c
Employee* worker_bee_array = (Employee*)NULL;
```

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

```c
worker_bee_array[index]
```

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

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

For example:

```c
worker_bee_array[index].pay_rate = 6.50;
printf("%f\n", worker_bee_array[index].pay_rate);
```
#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 = (Employee*)NULL;
    float* worker_bee_pay = (float*)NULL;
    int number_of_worker_bees, index;

    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);
    worker_bee_pay = (float*)malloc(sizeof(float) * number_of_worker_bees);

    for (index = 0; index < number_of_worker_bees; index++)
    {
        printf("What is the first name of 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';
        worker_bee[index].first_name = (char*)malloc(sizeof(char) * (strlen(dummy_name) + 1));
        strcpy(worker_bee[index].first_name, dummy_name);

        printf("What is the last name of the 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';
        worker_bee[index].last_name = (char*)malloc(sizeof(char) * (strlen(dummy_name) + 1));
        strcpy(worker_bee[index].last_name, dummy_name);

        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 this week?\n", worker_bee[index].first_name, worker_bee[index].last_name);
        scanf("%f", &worker_bee[index].hours_worked_this_week);

        printf("What is %s %s's social security number?\n", worker_bee[index].first_name, worker_bee[index].last_name);
        scanf("%d", &worker_bee[index].social_security_number);
    }

    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 = 0; index < number_of_worker_bees; index++)
    {
        printf("Employee %s %s (%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]);
    }

    return 0;
}