Structures

Darin Brezeale

The University of Texas at Arlington

Introduction

Arrays allow us to create many variables of the same type and reference them using a common name.

In most cases the elements of an array will be related in some way, for example, represent values for the months of a year.

Introduction

What if we want to create objects of variables that may or may not be of different types? In C, we can do this by creating a structure.

Structures

The form of a structure is

```
struct struct_name
{
    var_type var1;
    var_type var2;
     . . .
};
```

where

- `struct` is the keyword for defining a structure
- `struct_name` is the name for this structure (since there may be more than one)
- `var1, var2, ...` are the specific variables that make up the structure

Structures

To create a structure variable of the type `struct_name`, we would use this:

```
struct struct_name variable_name;
```

We are allocating memory for a variable called `variable_name`. The type of variable is a structure, specifically a `struct_name` structure.

This is analogous to creating other types of variables, for example,

```
int sum;
```

Structures

We can think of a structure as being a template for an object. For example, we may wish to create a structure for storing information about a person:

```
struct person
{
    char name[20];
    int age;
    double weight;
};
```

In our program, we might create two variables using this structure:

```
struct person John = {"John Smith", 25, 170.5};
struct person Mary = {"Mary Jones", 32, 120};
```

Accessing Structure Members

Once we have created a structure variable, we can access the specific members using a period, known as the **structure member operator**.

```
struct person John = {"John Smith", 25, 170.5};

printf("%s is %d years old.", John.name, John.age);
```

See `example-structures.c` on the course website.

Initializing Structure Variables

With other variables we have seen that we can declare the variable and later initialize it:

```
int x;
x = 5;
```

We can do the same with structures:

```
struct person Darin;

strcpy(Darin.name, "Darin Brezeale");
Darin.age = 41;
Darin.weight = 185;
```

Initializing Structure Variables

It’s also possible to define the structure and declare variables of this type simultaneously:

```
int main(void)
{
    struct person
    {
        char name[15];
        int age;
    } John, Mary;

    strcpy(John, "John Smith");
    John.age = 25;

    strcpy(Mary, "Mary Jones");
    Mary.age = 32;
}
```

Arrays as Structure Elements

We can have arrays as elements of a structure also:

```
struct arrayStruct
{
    int data[2][3];
    double values[4];
};
int main(void)
{
    struct arrayStruct numbers = { {{1, 2, 3}, {4, 5, 6}},
                                   {10, 20, 30 ,40} };
    int i, j;
    for(i = 0; i < 2; i++)
        for(j = 0; j < 3; j++)
            printf("%d ", *(*(numbers.data + i) + j) );

    for(i = 0; i < 4; i++)
        printf("%2.0f ", numbers.values[i] );
}
```

Arrays of Structures

Instead of creating many structure variables individually, we can create an array of structures.

```
#include <stdio.h>

struct id
{
    char name[20];
    int age;
};
int main(void)
{
    struct id person[2] = { {"John Smith", 25},
                            {"Mary Jones", 32} };
    int i;

    for(i = 0; i < 2; i++)
        printf("%s, %d\n", person[i].name, person[i].age );
}
```

Pointers to Structures

Creating a pointer to a structure:

```
#include <stdio.h>

struct id
{
    char name[20];
    int age;
};

int main(void)
{
    struct id person = {"John Smith", 25};
    struct id *ptr = &person;  /* requires & */
}
```

Notice this requires using the address operator, &, to get the address of the beginning of the structure. This demonstrates that structures and arrays are different things.

Pointers to Structures

To access an element of the structure using the pointer, we could do the following:

```
#include <stdio.h>

struct id
{
    char name[20];
    int age;
};

int main(void)
{
    struct id person = {"John Smith", 25};
    struct id *ptr = &person;  /* requires & */

    printf("%d\n", (*ptr).age );
}
```

Pointers to Structures

To access a structure member, we use

```
(*ptr).age
```

This is not the same as

```
*(ptr.age)
```

which would be legal if `age` was a pointer that was a member of a structure called `ptr`, for example,

```
struct info
{
    int* age;
    double weight;
};

struct info ptr;
```

Pointers to Structures

Instead of using `(*ptr).age`, there is an alternative syntax:

```
ptr->age
```

where `->` is the **structure pointer operator**.

```
struct id person = {"John Smith", 25};
struct id *ptr = &person;

ptr->age = 50; /* change John's age to 50 */
```

Structures and Functions

We can pass either a copy of a structure or a pointer to a structure to a function. Structures can also be the return type of a function.

Keep in mind that passing copies of structures requires more memory, which may be an issue in some situations.

More Examples

The following examples on the course website demonstrate the use of structures and functions:

- `example-structures789.c` – pass by value, pass by reference, and structure as function return type
- `example-structures10.c` – pass array of structures
- `example-structures11.c` – strings and structures

More Examples cont.

The following examples on the course website demonstrate the use of structures and functions:

- `example-structures12.c` – pass array elements to a function
- `example-structures13.c` – change array values from function
- `example-structures14.c` – pass array of structures several times
- `example-struct-unique.txt` – create array of unique items