Fundamentals of Programming 1

Structures and Unions

Arkadiusz Chrobot

Department of Information Systems

January 18, 2023

Outline

Structures

The structure in the C language is a data structure that makes it possible to store values of a different types in a single variable. In other programming languages such data structures are called records. To apply a structure in a program it is necessary to define its type first. The overall pattern of such a definition is as follows:

```
struct name_of_the_structure_type
{
        field_type field_name_1;
        field_type field_name_2;
        ...
        field_type field_name_n;
};
```

Fields (or members) inside the structure are declared in the same way as regular variables and they can be of any data type including an array, another structure and a union. The last data type will be introduced in this lecture.

Structures

Variables

To declare a variable of a structure type it is necessary to use the `struct` keyword before the name of the structure type, just as in the following pattern:

```
struct name_of_structure_type name_of_variable;
```

It is also possible to declare a structure variable together with the structure type:

```
struct name_of_the_structure_type
{
        field_type field_name_1;
        ...
        field_type field_name_n;
} name_of_variable_1, name_of_variable_2;
```

There are two variables declared in the pattern above. If only one was necessary than its name would be placed between the closing brace and the semicolon.

Structures

Examples of Structured Types and Variables

Structures have many applications. They are used for gathering data of different types, but some common characteristics. They can for example store personal data:

```
struct personal_data
{
        char name[LENGTH], surname[LENGTH];
        unsigned char age, height, weight;
};
```

Variables of `struct personal_data` type can store such data as a name, surname, age, height and weight of a specific person. Please observe, that just like in the case of regular variables it is possible to declare several fields of the same type just by placing the name of the type first and then by giving the names of the variables, separating them with commas and putting at the end a semicolon.

Structures

Examples of Structured Types and Variables

Structures can also by applied for storing some number of values of the same type that have a special meaning in the problem to be solved. For example, a structure can store coordinates of a point in three dimensional space:

```
struct coordinates
{
        double x,y,z;
} point;
```

Structures

Nested Structures

As it was mentioned before, the C language allows for nesting structures. For example, the `struct personal_data` structure can be supplemented with a field, for storing the address of a person, which itself is a structure of the `struct address` type:

```
struct personal_data {
    char name[LENGTH], surname[LENGTH];
    unsigned char age, height, weight;
    struct address {
        char street_name[LENGTH], postal_code[LENGTH];
        unsigned short int house_number;
    } personal_address;
};
```

Structures

Arrays of Structures

The C language makes it possible to create arrays of structures. For example an array of structures of the `struct personal_data` type defined in previous slide may be declared as follows:

```
struct personal_data people[NUMBER_OF_ELEMENTS];
```

It is assumed that the `NUMBER_OF_ELEMENTS` constant specifies the number of the array's elements and that it is defined before the array declaration. The array of structures can also be created at the place of structure type definition, just like a regular variable.

Structures

Accessing Fields

A structure member can be accessed by prefixing its name with the name of the variable in which it is embedded and separating those two names with a dot, just like in the following pattern:

`name_of_variable.name_of_field`

For example a value to the `x` filed of the `point` structure can be assigned like this:

```
point.x = 3;
```

Referencing a field in a nested structure requires using the dot more than once:

```
person.personal_address.house_number = 127;
```

If the structure is an element of an array, then the name of the variable in the pattern has to be replaced with the reference to a specific element of the array:

```
people[0].age = 37;
```

Structures

Initialization of Structures

The variables of a structural type may be declared as global or local. The former are by default initialized with zeros, whereas the latter are uninitialized and the programmer is responsible for assigning them initial values. There are also cases when other than default values should be assigned to a global structure. At least three ways of initializing a structure exist.

Structures

Initialization of Structures — the First Method

The variable of a structural type may be initialized in the place of its declaration, similarly to an array – the values for the fields have to be embraced with braces and separated by commas:

```
#include <stdio.h>

struct coordinates
{
    double x, y, z;
} point = {1.0, 2.0, 3.0};

int main(void)
{
    struct coordinates another_point = {4.0, 5.0, 6.0};
    printf("x: %f ",another_point.x);
    printf("y: %f ", another_point.y);
    printf("z: %f\n", another_point.z);
    return 0;
}
```

Structures

Initialization of Structures — the First Method

The example in the previous slide shows initialization of two variables of the `struct coordinates` type: the `point` and `another_point`. The first variable is declared and initialized in the place where its type is defined. The `x` field gets the value of `1.0`, the `y` field, the value of `2.0` and the `z`, the value of `3.0`. The second variable is declared as local and it is also initialized in the place of its declaration. If one of the initial values was missing then, according to the C language standard, the third field would get the value of `0`. Neither the `scanf()` nor the `printf()` function has a formatting string suitable for simultaneous reading or displaying the values of a structure. The value of every structure member has to be passed to those functions separately, just like in the example program.

Structures

Initialization of Structures — the Second Method

The second way of initializing of structures is similar to the first one, but involves using names of the members that are to be assigned a value. They are placed inside brackets and prefixed with a dot, like in the example program:

```
#include<stdio.h>

struct coordinates
{
    double x, y, z;
} point = {.x=1.0, .z=2.0, .y=3.0};


int main(void)
{
    struct coordinates another_point = point;
    printf("x: %f ",another_point.x);
    printf("y: %f ", another_point.y);
    printf("z: %f\n", another_point.z);
    return 0;
}
```

Structures

Initialization of Structures — the Second Method

Using names of fields explicitly allows initializing them in any order. It is also possible to initialize only some of them. The rest of them will be assigned the value of zero. The example program shows also that a variable of structural type may be assigned a value of another variable of the same type. As a result of such an assignment the values of the fields of the variable on the right side of the assignment operator are assigned to appropriate fields of the variable on the left side of the operator. This is not possible in the case of structures of different types. It is also not possible to cast a structure on a different structure type.

Structures

Initialization of Structures — the Third Method

The last way of initializing a structure consist in assigning values to its fields outside of the place of declaring the variable:

```c
#include <stdio.h>

struct coordinates
{
    double x, y, z;
};


int main(void)
{
    struct coordinates point;

    point.x = point.y = point.z = 1.0;

    printf("x: %f ", point.x);
    printf("y: %f ", point.y);
    printf("z: %f\n", point.z);
    return 0;
}
```

Structures

Initialization of Structures — the Third Method

In the example program every field of the `point` variable gets a value of `1.0`. If any of the fields was omitted then its value would depend on the scope of the structure. The field in a global variable would get a value of zero. In case of a local variable, its value would be undefined.

Unions

The union is similar to the structure. The main difference between those two construct is that, unlike in the case of the structure, fields in the union overlap in the memory. It means that they share the same area of the memory. As a result the union typically occupies less place in the memory than the structure with the same members and modification of the value of one of its fields may influence the values of the other fields.

Unions

Example of a Union

The union type is defined similarly to structure type. Also a union variable is declared in a similar fashion to a structure variable.

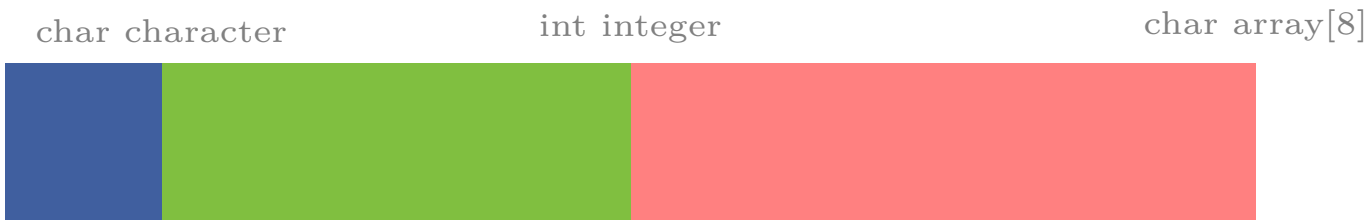
```
union union_type_example
{
    char character;
    int  integer;
    char array[8];
} union_example;
```

In the listing a union type is defined and a variable of this type is declared. The variable is called a *union* for short. The size of the union (the number of bytes it occupies) can be calculated with the use of the `sizeof` operator. The results may vary depending on a computer and a compiler configuration, but usually the union takes less memory than a corresponding structure.

Unions

Fields Overlapping

The picture below illustrates overlapping of the fields.



The illustration is for reference only. The way in which the fields overlap depends on the type and configuration of a computer and a compiler. However, modification of the value of one of the fields may result in modification of the value of one or more of the others.

Unions

Similarities Between Structures and Unions

Not only definitions of types and declarations of variables of structures and unions are similar. The unions may be initialized in the same way as structures, but if the first method is used for initializing more than one field then the compiler will issue a warning, that the resulting values of the fields may vary from the expected ones.
Using the `union` or `struct` keywords while declaring a union or structure may be inconvenient. This drawback may be removed with the use of the `typedef` keyword which allows the programmer to give a different (usually shorter) name for a type. It should however be used carefully, because it degrades the legibility of the source code.

Unions

Applications

Due to fields overlapping unions are often used for converting types of different values, for example the IP address may be converted from binary into decimal and vice versa or number can be converted from decimal into BCD or the other way. The conversion consists in storing a value in specific fields and reading other union members. However, the C language standard does not recommend using this technique because the result of such a conversion depends on the architecture of the computer system and thus can be in some cases incorrect. The aforementioned document suggests always reading only the recently modified union field. Hence, it is better to apply unions as a fields of structures, to save some memory. This method requires declaring an additional filed in the structure for signaling which field of the union should be used. Such usage of a union is shown in a program presented at the end of the lecture.

Bit Fields

The C language makes it possible to declare fields of structure which size is expressed in bits. Those fields are called simply *bit fields*. The `sizeof` operator cannot be applied to such fields. It is also impossible to assign to them greater or smaller value than it is allowed by their size. Yet, it does not mean that the overall size of the structure is a sum of bit fields sizes. A structure that contains only two bit fields of the size of five bits each has a size of at least two bytes, but not a size of ten bits. The size of every structure is always a positive integer multiple of a byte. Bit fields are just a special notation that forces the program to use only as many bits in the fields as the programmer has specified. As the type of a bit field should be used any type that allows for storing integer or natural numbers, like the `unsigned char` or `int`. Unions can also have bit fields, but there are less useful than those in structures.

Bit Fields

Example of a Structure with Bit Fields

```
struct bit_field_example
{
    int flag:1;
    char small_number:2;
};
```

A single bit is often used as so-called *flag*, i.e. a variable that stores information that signals for example an occurrence of an exception. Thus the field of the "size" of one bit in the structure is named that way.

Structures, Unions and the Functions

Unions and structures can be returned by functions. The listing presents a source code of a function that returns a structure. A function that returns a union may be declared similarly.

```
struct coordinates get_point(double x, double y, double z)
{
    struct coordinates point;

    point.x = x;
    point.y = y;
    point.z = z;

    return point;
}
```

Structures, Unions and Functions

The Function Returning a Structure — a Comment To the Example

The function shown in the previous slide stores values passed to it by parameters in a structure which is declared locally. Next, it returns the structure. The function may be invoked in the following way:

```
struct coordinates start = get_point(0.0, 0.0, 0.0);
```

As a result of the function call the values from its local structure are stored in the `start` variable.

Structures, Unions and Functions

Passing By Parameters

Both structures and unions can be declared as parameters of a function. By default structures and unions are passed by value, just like any others variables, apart from arrays. If the structure or union parameter should be also an output parameter, then it has to be declared as a pointer. Access to the fields of a union or structure passed or just pointed by a pointer may be gained with the use of one of the two notation. The first one is less readable and follows the pattern:

```
(*name_of_pointer_to_structure).field_name
```

The second one in more legible thanks to the use of a special operator: `->` and thus it is applied more often:

```
name_of_pointer_to_structure->field_name
```

Structures, Unions and Functions

Examples

```
void move(struct coordinates *point,
                                    struct coordinates vector)
{
    point->x += vector.x;
    point->y += vector.y;
    point->z += vector.z;
}
```

Structures, Unions and Functions

A Comment to the Example

The function from the previous slide determines coordinates of a point in a three dimensional space after it is shifted by a given vector. The first parameter points to a structure that before the function call stores original coordinates of the point, and after the function finishes it stores the resulting coordinates. The structure that describes the vector is passed to the function by the second parameter. The values of each of its fields are components of the vector (lengths of the vector in each of the dimensions). The function adds the corresponding fields of the two structures and stores the results in the `point` variable. As a first argument of the function call should be passed an address of a variable of the `struct coordinates` type or a pointer to such a variable. As the second parameter should be passed a structure of the aforementioned type. The function may be invoked like this:

```
move(&start,distance);
```

Examples

Array of Structures

A program that stores personal data, such as name, surname and age, in an array of structures is presented as the first example. The aforementioned data are created with the use of PRNG. The age is chosen randomly in the range from 1 to 120 and the name and surname are chosen randomly from predefined arrays. There is one array for surnames and another for first names which contains male and female names.

Examples

Array of Structures

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>

#define LENGTH 50
#define NUMBER_OF_PEOPLE 5

enum gender {MALE, FEMALE};
```

Examples

Array of Structures — a Comment

The part of the source code presented in the previous slide contains, besides preprocessor directives that include header files in the program, definitions of constants. The first one describes how many characters can store the name and surname arrays. The second one defines the number of people for whom personal data should be generated. There is also defined an enumerated type which elements are used for describing the gender of a person.

Examples

Array of Structures

```
struct name_forms
{
    char male_name[LENGTH], female_name[LENGTH];
} names[] = {{.male_name = "Andrew", .female_name = "Anne"},
             {.male_name="Edward", .female_name="Katherina"},
             {.male_name="Henry", .female_name="Margaret"},
             {.male_name="John", .female_name="Barbara"},
             {.male_name="Jacob", .female_name="Joanna"}};

char surnames[NUMBER_OF_PEOPLE][LENGTH] = {"Smith", "Brown",
                                           "Green", "White", "Johnson"};
```

Examples

Array of Structures — a Comment

The part of the source code from the previous slide contains declarations and initializations of two arrays: `names` and `surnames`. The elements of the first array are structures of the type `struct name_forms`. Each of them stores two names, one for a male and another for a female. The `surnames` array is an array of strings which elements store surnames. The elements of those arrays are chosen randomly with the use of PRNG in order to create records (structures) of personal information.

Examples

Array of Structures

```
struct personal_data
{
    char name[LENGTH], surname[LENGTH];
    unsigned char age;
} people_data[NUMBER_OF_PEOPLE];
```

Examples

Array of Structures — a Comment

The previous slide contains a definition of an array for storing the personal data. The number of elements of this array is defined by the constant `NUMBER_OF_PEOPLE`. Each of them is a structure that stores a name, surname and age of a single person.

Examples

Array of Structures

```
struct personal_data get_randomized_data(struct name_forms names[], char surnames[][LENGTH])
{
   struct personal_data person;

   person.age = 1+rand()%120;
   strncpy(person.surname, surnames[rand()%NUMBER_OF_PEOPLE], LENGTH-1);
   unsigned char gender = rand()%2;
   if(gender==FEMALE)
        strncpy(person.name,names[rand()%NUMBER_OF_PEOPLE].female_name,LENGTH-1);
   else
       strncpy(person.name,names[rand()%NUMBER_OF_PEOPLE].male_name,LENGTH-1);

   return person;
}
```

Examples

Array of Structures — a Comment

The `get_randomized_data()` function generates data about a single person. The arrays with names and surnames are passed to it by parameters. The function chooses randomly the age of the person as the first personal data item. Next, it choses randomly an index of a single element in the `surnames` array and the value of the element is then coppied to the `surname` field of the `person` structure. Then the function chooses randomly a gender of the person. If it is a female the function should choose randomly her name from the female names and if it is a male the function should choose randomly his name from the male names. The structure with the chosen personal data is returned by the function.

Examples

Array of Structures

```
void fill_array(struct personal_data array[],
                struct name_forms names[],
                char surnames[][LENGTH])
{
    srand(time(0));
    int i;
    for(i=0;i<NUMBER_OF_PEOPLE;i++)
        array[i]=get_randomized_data(names, surnames);
}
```

Examples

Array of Structures — a Comment

The `fill_array()` function initializes the PRNG and assigns to each of the elements of the array, passed to it by the first parameter the value returned by the `get_randomized_data()` function.

Examples

Array of Structures

```
void print_array(struct personal_data array[])
{
    int i;
    for(i=0;i<NUMBER_OF_PEOPLE;i++) {
        printf("Name: %s\n",array[i].name);
        printf("Surname: %s\n",array[i].surname);
        printf("Age: %u\n",array[i].age);
        puts("");
    }
}
```

Examples

Array of Structures — a Comment

The `print_array()` function prints the data from the array on the screen. The array is passed to the function by the parameter. The value of each field of each element is printed in a separate line. Moreover, after all data from a single element are printed on the screen the cursor is moved one additional line lower to separate data of different elements.

Examples

Array of Structures

```
int main(void)
{
    fill_array(people_data, names, surnames);
    print_array(people_data);
    return 0;
}
```

Examples

Array of Structures — a Comment

In the `main()` function the `fill_array()` and `print_array()` functions are invoked. All needed arguments are passed to them. Please notice, that the `people_array` is passed by the `array` parameter. In the C language, and most of the other programming languages, the name of the parameter may differ from the name of an argument. Only their types should be compatible.

Examples

Structure and Union

The next example shows how to apply a union as a member (field) of a structure. In other words the union is nested in the structure. Both variables are used in a program that creates and displays information about a computer game character.

Examples

Structure and Union

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define LENGTH 10

enum character_type {WARRIOR, SORCERER};
```

Examples

Structure and Union — a Comment

The code from previous slide contains preprocessor directives that include header files to the program, a definition of a constant that describes the number of elements of an array used for storing the name of the game character and a definition of an enumerated type which describes the character type: a sorcerer or a warrior.

Example

Structure and Union

```
struct playable_character {
    char name[LENGTH];
    enum character_type type;
    union {
        float strength;
        double magic_power;
    } abilities;
};
```

Example

Structure and Union — a Comment

The previous slide contains a definition of a type of a structure for storing information about the game character. The first field is an array in which the name of the character will be stored. The second field is for storing the type of the character. The third field is a union. If the character is a warrior then the information about her/his strength will be stored in the `strength` field. Otherwise, if she/he is a sorcerer the `magic_power` field will store data about her/his magic power.

Example

Structure and Union

```
void generate_character(struct playable_character *character)
{
    puts("Please name Your character:");
    scanf("%9s",character->name);
    srand(time(0));
    if(rand()%2==WARRIOR) {
        character->type = WARRIOR;
        character->abilities.strength = rand()%1000+(float)rand()/(RAND_MAX+1.0);
    } else {
        character->type = SORCERER;
        character->abilities.magic_power = 1000+rand()%RAND_MAX
                                           + (double)rand()/(RAND_MAX+1.0);
    }
}
```

Examples

Structure and Union — a Comment

The `generate_character()` function fills the structure, that is passed to it by a parameter, with data. First it asks a user to enter the name of the character. The name is stored in the `name` field with the use of the `scanf()` function. Please notice, that the field is an argument of the aforementioned function. The arrow operator is used to access that field, because the structure is passed by pointer. Number of characters read by `scanf()` function from the keyboard is limited to 9, because of the number of characters the field can store. Next, the `generate_character()` function initiates the PRNG and chooses the character's type. If it is a warrior then the function stores this information in the `type` field and chooses a value for the `strength` field in the `abilities` union. Please notice the way it references the union fields. The structure is passed by a pointer, but the union is a regular variable that is a member of this structure, so a regular dot is used for its fields. If the chosen character's type is a sorcerer, the function performs similar activities.

Examples

Structure and Union

```
void print_character(struct playable_character character)
{
    printf("Name: %s\n",character.name);
    if(character.type==WARRIOR) {
        printf("Type: warrior\n");
        printf("Strength: %f\n",character.abilities.strength);
    } else {
        printf("Type: sorcerer\n");
        printf("Magic power: %f\n",
                        character.abilities.magic_power);
    }
}
```

Examples

Structure and Union — a Comment

The `print_character()` function displays information about the game character that are stored in the structure. This time the structure is passed to the function by value, so its fields as well as the fields of the union are referenced with the use of the dot. If the value of the `type` field was displayed directly then the user would see on the screen 0 or 1 depending of the character's type. To avoid this issue the value of the field is used only to determine which message and which field of the `abilities` union should be displayed on the screen.

Examples

Structure and Union

```c
int main(void)
{
    struct playable_character character;
    generate_character(&character);
    print_character(character);
    return 0;
}
```

Examples

Structure and Union — a Comment

A structure of the name `character` is declared in the `main()` function and filled with data by the `generate_character()` function. Next, its content is displayed on the screen with the use of `print_character()` function.

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, MSc for helping me to complete the Polish version of this slides.

Questions

?

The End

Thank You for Your attention!