Fundamentals of Programming 1

Enumerated Types and Linear Arrays

Arkadiusz Chrobot

Department of Computer Science

December 7, 2020

Outline

Data Abstraction

Abstraction can be applied not only to statements but also to data. The C programming language allows a programmer to create custom data types specific to a problem that is to be solved. An example of such a data type is the enumerated type.

Enumerated Types

An enumerated type makes it possible to give names to characters or numbers that belong to a subset of integers. In other words, the enumerated type may be considered as a set of constants. The overall pattern for defining an enumerated type is the following:

```
enum type_name {ELEMENT_1=value, …, ELEMENT_N};
```

Please note, that the name (identifier) of each element is written in uppercases, just like names of constants. This is only a matter of convention. Any name can be used, provided it is legal in the C language. To each element of the enumerated type can be assigned a character or an integer number. However, if no value is given to the elements by the programmer, the compiler will assign to them successive numbers of `int` type, starting from 0. It is also possible for the programmer to assign the same value to more than one element or to assign a value only to one element or a selected group or groups of elements. The rest of them will get default values.

Enumerated Types

Examples

```
enum names_of_days {MONDAY=0, TUESDAY=1, WEDNESDAY=2, THURSDAY=3,
                    FRIDAY=4, SATURDAY=5, SUNDAY=6};
```

In the above definition of an enumerated type an integer number is assigned to each name of a day. The same definition can be written as follows:

```
enum names_of_days {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
                    SATURDAY, SUNDAY};
```

However, if a programmer wants the values of the days to start with 1 instead of 0 she or he should define the type like this:

```
enum names_of_days {MONDAY=1, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
                    SATURDAY, SUNDAY};
```

Each subsequent day, after Monday, gets a number greater by one than its predecessor, i.e. `TUESDAY=2`, `WEDNESDAY=3`, etc.

Enumerated Types

Examples

If the programmer wants to distinguish the days of weekend, by assigning them for example numbers 9 and 10, then she or he can do that like this:

```
enum names_of_days {MONDAY=1, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
                    SATURDAY=9, SUNDAY};
```

The values can also be assigned to the elements in a descending order:

```
enum directions {NORTH=3, WEST=2, EAST=1, SOUTH=0};
```

Variables of an Enumerated Type

A variable of an enumerated type may be declared as local (also as a function's parameter) or global. The overall pattern for such a declaration is as follows:

```
enum name_of_enumerated_type name_of_variable;
```

Any element of its enumerated type can be assigned to the variable. The variable may be applied as a loop counter in a `for` loop, a selector in a `switch` statement or it may be used in conditional statements and in the condition-controlled loops.

Unfortunately, the C language implementation of the enumerated types is imperfect. They are only a facilitation for the programmer. The compiler does not verify the correctness of using the enumerated type variables, considering them to be of `int` type. Thus it is possible to assign to such variables any integer number. This can lead to many mistakes. The C language makes it also possible to use the elements of enumerated type as common constants and to define constants of enumerated type with the use of the `const` keyword.

Enumerated Types

Examples

```c
#include <stdio.h>

enum names_of_days {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY};

void print_message(enum names_of_days day)
{
    switch(day) {
    case MONDAY:
    case TUESDAY:
    case WEDNESDAY:
    case THURSDAY:
    case FRIDAY:
        puts("Go to work!");
        break;
    default:
        puts("Relax!");
    }
}

int main(void)
{
    enum names_of_days day;
    for(day=MONDAY;day<=SUNDAY;day++)
        print_message(day);
    return 0;
}
```

Enumerated Types

Comment to the Example

In the program from the previous slide a local enumerated type variable is declared in the `main()` function. It is the `day` variable. Also in the `print_message()` function a parameter of the same type and name is declared. The variable in the `main()` function is used as a loop counter and the one in the `print_message()` is applied as a `switch` selector. It is worth noticing, that in the `switch` statement most of the cases are empty, but it is not a mistake but an intended effect. That way, the code for Friday is performed also for all other days, except for Saturday and Sunday, and it is not repeated. The code for those two mentioned days is performed in the default case.
There is no easy way to print the value of an enumerated type variable. It is however possible to display its numerical value with the use of `printf()` function and the `"%d"` formatting string.

Enumerated Types

Examples

```c
#include <stdio.h>

enum names_of_days {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY=9, SUNDAY};

void print_message(enum names_of_days day)
{
    switch(day) {
    case MONDAY:
    case TUESDAY:
    case WEDNESDAY:
    case THURSDAY:
    case FRIDAY:
        puts("Go to work!");
        break;
    default:
        puts("Relax!");
    }
}

int main(void)
{
    enum names_of_days day;
    for(day=MONDAY;day<=SUNDAY;day++)
        print_message(day);
    return 0;
}
```

Enumerated Types

Comment to the Example

A small change in the program, that assigns the value of 9 to the `SATURDAY` element shows the imperfections of the enumerated types. The output of the program suggests that the week has 11 days, most of them free. The problem here is the `day` variable which is applied as a loop counter. When the value of the counter is 4, what corresponds to the `FRIDAY` element, it is incremented in the next iteration of the loop to 5 which cannot be mapped to any of the type's element, but it still a proper value, because for the program the `day` variable is of the `int` type. In the C language the enumerated type is only a simple container for constants and should be used carefully.

The typedef Keyword

It is easy to forget that the declaration of an enumerated type variable always starts with the **enum** keyword. To avoid issues with the missing **enum** keyword, the **typedef** keyword may be applied. The latter keyword allows for giving the enumerated type a shorter name:

```
typedef enum names_of_days {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY=9, SUNDAY} days;
```

The variable of the type may now be declared like this:

```
days day = MONDAY;
```

Generally, the **typedef** keyword makes it possible to give an alternative name to any data type including the predefined data types of the C language. Therefore, it should be used carefully. Many programmers advise against using it, since it can make the program difficult to read by concealing the true types of variables.

Linear Arrays

One-dimensional arrays or simply the linear arrays are an example of a *data structure* i.e. a variable that can store more than one value at the same time. In case of arrays all those values are of the same type. The picture below shows the construction of a linear array that can store up to 8 integer numbers.

0	1	2	3	4	5	6	7
-7	7	10	-3	0	7	15	0

The numbers on the top of the array are the indices that uniquely locate an element inside the array. In the C language the indices are natural numbers. The first element of the array always has an index of 0. The element is a single location in the array that holds a single value. All indices are unique and form an ascending sequence. The values of the elements may repeat and do not have to be sorted. The array is sometimes called (not entirely correctly) a *vector*.

Linear Arrays

Declaration

Just like other variables the array has to be declared before using. It can be declared as global or local variable. The elements of the former are initialized with zeros and the initial value of the elements of the latter is unspecified. The overall pattern of declaration of an array is as follows:

```
data_type array_name[NUMBER_OF_ELEMENTS];
```

An element of the array may be of any of the types that were already introduced in the lecture. The arrays of characters have a special meaning and they will be discussed separately in a future lecture. The name (identifier) has to be legal in the C language. The number of elements determines how many elements will have the array. It can be given as a literal or a constant defined with the use of `#define` statement. According to the ISO C99 standard, the number of elements has to be greater than zero. The indices range stars from zero and ends with the number of elements minus one.

Linear Arrays

Accessing the elements

It is worth noticing, that the array resembles the structure of RAM. In order to read or modify the content of a single memory cell an address has to be provided. Similarly, if a programmer wants to read or write an element of an array she or he has to use its index. Referencing an element of an array follows this pattern:

```
array_name[index]
```

In the C language the array name is equivalent to a pointer. Thus there are three additional ways of referencing the element:

```
*(array_name+index)
*(index+array_name)
index[array_name]
```

The first two expressions use so-called pointers arithmetic. The most frequently used way of accessing an array element is the one introduced as first. The second one is also sometimes applied. The last two are rarely used because they are less legible.

Linear Array

Array Size and Number of Elements

The size of an element (the number of bytes it occupies in memory) can be acquired with the use of `sizeof` operator. The number of array's elements can be computed using the following expression:

```
sizeof(array_name)/sizeof(array_name[0])
```

The expression can be a little simplified with the use of the pointer arithmetic:

```
sizeof(array_name)/sizeof(*array_name)
```

Unfortunately, both expressions and the `sizeof` operator give incorrect results when the array is a function's parameter.

Linear Array

Passing to a Function

Arrays can and should be passed by parameters to functions. The array parameter can be declared the same way as a regular array, although in a function's list of parameters. However, the number of elements in such a parameter is ignored. An array of any legal number of elements can be an argument for this parameter. Only the data types of the elements of the argument and the parameter have to be compatible. Therefore, the number of elements in the array parameter is usually omitted. The array may be also passed to the function with the use of a pointer parameter. The pointer should be of the same type as the elements of the array or it should be declared as a `void *` pointer. In the second case, the data type of the pointer means that it can be assigned a pointer of any other type. Effectively, the array is always passed by a pointer, which means that changing a value of any of the parameter's elements results in the modification of the value of the same element in the argument.

Initialization of an Array

Initialized Array

The array may be initialized in the place of its declaration. To this end an assignment operator has to be put after the closing bracket followed by a list of values in curly braces. The values have to be separated by commas. If there is provided less values in the list than the array has elements then only the starting elements of the array will be initialized. It is also possible to skip the number of elements when initializing an array this way. The compiler will figure it out by the number of provided initial values.

```
int main(void)
{
        double fractions[] = {0.1, 0.2, 0.3};
        double fractions_2[3] = {0.1, 0.2, 0.3};
        return 0;
}
```

Initialization of an Array

Initialization Made by User

The initial values for the array can be provided by user, during a program run, with the use of a keyboard. It requires using a loop for indexing the elements of an array and the `scanf()` function. An element of the array is a single variable. Therefore its address should be provided as the second argument of the `scanf()` call. The address is obtained with the use of the address operator. An example:

```
int main(void)
{
        int array[5];
        unsigned int i;

        for(i=0;i<5;i++)
                scanf("%d",&array[i]);

        return 0;
}
```

Initialization of an Array

Initialization with the Use of Indices

Using one of the two introduced ways of initializing arrays in case where the array has huge number of elements could be cumbersome. Alternatively, if the elements of the array are of `int` or compatible type, they can be assigned the values of their indices:

```
int main(void)
{
        int array[1000];
        unsigned int i;
        for(i=0;i<1000;i++)
                array[i] = i;
        return 0;
}
```

It is a simple way of initializing an array. However, the values of the elements are unique and arranged in an ascending sequence, which is not always desired.

Pseudorandom Number Generator

An array may be initialized with the use of a pseudorandom number generator (PNG). It is an algorithm that generates numbers appearing to be random. However, it can be statistically verified that they are not truly random. This usually excludes their use in cryptography, but for the purpose of this lecture they are sufficiently random. To distinguish them from truly random numbers, they are called pseudorandom. The algorithm uses an initial value called a *seed* and a mathematical formula to generate such numbers.

Pseudorandom Number Generator

Using the PNG in the C Language

In the C language there are two functions, declared in the `stdlib.h` header file, that are the implementation of the PNG. Those are `srand()` and `rand()`. The former takes as an argument an `unsigned int` number and makes it the seed for the PNG. The latter does not take any argument and returns a pseudorandom `int` number ranging from 0 to `RAND_MAX`. The `srand()` function **has to be called outside any loop**. The argument for the function may be the result of the `time()` function declared in the `time.h` header file. An argument for the latter function can be `NULL` or zero.

Pseudorandom Number Generator

The PNG Usage

The PNG generates only natural numbers. If a pseudorandom natural number is needed ranging from 0 to 9 (both inclusive) it can be chosen with the following expression:

```
int x = rand()%10;
```

To choose numbers ranging from 1 to 10 (both inclusive) the expression should be modified as follows:

```
int x = 1+rand()%10;
```

To pick integer numbers ranging from -10 to 10 (both inclusive), a following expression can be applied:

```
int x = -10+rand()%21;
```

If a floating-point pseudorandom number is needed from the interval of [-10,11) it could be chosen with the following expression:

```
double x = -10+rand()%21+rand()/(RAND_MAX+1.0);
```

Pseudorandom Number Generator

The PNG Usage

To pick a pseudorandom lowercase letter from the set of 26 lowercases the following expression may be used:

```
char x = 'a'+rand()%26;
```

Array Initialization

Array Initialization with Pseudorandom Numbers

Below is defined a function that initializes an array of `NUMBER_OF_ELEMENTS` elements with pseudorandom numbers ranging from 0 to 199 (both inclusive):

```
void fill_array_with_random_numbers(int array[])
{
    srand(time(0));
    int i;
    for(i=0; i<NUMBER_OF_ELEMENTS; i++)
        array[i]=rand()%200;
}
```

Please observe, that the number may not be unique.

Array Initialization

Shuffling

An array containing unique numbers arranged in an ascending order may be created by assigning to each of its elements the value of the element's index. Such an array can be changed into an unsorted array with unique numbers by applying the shuffle algorithm. The algorithm generates a permutation of the values by visiting each of the elements of the array, except for the last one, and swapping its value with other (pseudo) randomly chosen element. The latter element is selected from a set constituted of the yet not visited elements of the array and the currently visited element. This algorithm is implemented with the help of three functions that are described in the next slides.

Array Initialization

Shuffling — Swapping the Value of Elements

Below is defined a function that swaps the value of two variables. Those variables are passed to the function by pointers.

```
void swap(int *first, int *second)
{
        int tmp;
        tmp = *first;
        *first = *second;
        *second = tmp;
}
```

Array Initialization

Shuffling — Choosing an Element

The function presented in this slide chooses the index of an element of the array, that belongs to the set consisting of the currently visited element and the yet not visited elements. The index of the currently visited element is passed to the function by the `from` parameter. The number of elements of the array is denoted by the `ARRAY_LENGTH` constant.

```
int choose(int from)
{
        return from+rand()%(ARRAY_LENGTH-from);
}
```

Array Initialization

Shuffling — the Implementation

The function defined below shuffles the values in the array. Please notice, that the `choose()` function is invoked in brackets. It means that the value returned by the function is used as the index of the element which value is swapped with the value of the currently visited element of the array. The index of the latter one is in the `i` variable.

```
void shuffle(int array[])
{
        srand(time(0));
        unsigned int i;
        for(i=0;i<ARRAY_LENGTH-1;i++)
                swap(&array[i],&array[choose(i)]);
}
```

Array Coping

Two arrays of the same number and the same type of elements may be copied with the use of a loop. However, the task may be accomplished more efficiently if the `memcpy()` function is used. The function is declared in the `string.h` header file. It takes three arguments. The first one is the the name of the destination array. The second one is the name of the source array. The third one is the size of the copied (source) array. Generally, the sizes of the arrays may differ. In that case the destination array should be bigger than the source array. The value returned by the `memcpy()` is usually ignored. Other helpful function, that is also declared in the `string.h` file, is `memset()`. This function also takes three arguments. The first one is the name of an array. The second one is an `int` number that should be copied to all elements of the array. The last argument is a size of the array. The `memset()` function could be utilized to initiate elements of the array with the same value. It is also a common practice to ignore the value returned by `memset()`.

Printing the Values of Array's Elements

An iterative statement, like the `for` loop, may be used for indexing and successively visiting all elements of an array. Below is defined a function that uses such a loop to display all values stored in an array.

```
void print_array(int array[])
{
    unsigned int i;
    for(i=0; i<100; i++)
        printf("array[%u]: %d ",i,array[i]);
}
```

The values may also be displayed in a simplier manner:

```
void print_array(int array[])
{
    unsigned int i;
    for(i=0; i<100; i++)
        printf(" %d ",array[i]);
}
```

Finding the Minimum Value

The Algorithm

Solving some problems may require finding the smallest value in an unsorted array. The algorithm for that task is quite simple:

1. Store the value of the first element of the array in a separate variable. Assume for now, that it is the minimum value.
2. Visit all other element of the array, starting from the second one. If the currently visited element has a value smaller then the one stored in the variable, then store the value of the element in the variable.
3. If all elements of the array has been visited, the minimal value is in the variable.

Finding the Minimum Value

The Implementation

The function blow implements the algorithm described in the previous slide. It searches for the minimum value in an array of the number of elements given by the constant `NUMBER_OF_ELEMENTS`.

```
int find_min(int *array)
{
    int min;
    unsigned int i;
    min=array[0];
    for(i=1; i<NUMBER_OF_ELEMENTS; i++)
        if(min>array[i])
            min=array[i];
    return min;
}
```

Finding the Maximum Value

The Implementation

The algorithm for finding the maximum value is basically the same as for finding the minimum value. The only difference is the operator used in the conditional statement — less than instead of greater than. The function that implements the algorithm is named accordingly and so is the variable inside the function, that stores the result.

```
int find_max(int *array)
{
    int max;
    unsigned int i;
    max=array[0];
    for(i=1; i<NUMBER_OF_ELEMENTS; i++)
        if(max<array[i])
            max=array[i];
    return max;
}
```

Finding the Extremes

It is easy to spot, that in the case where both values (the minimum and the maximum) are needed it would be better to search for both of them simultaneously than look them up separately. The function below realizes this idea. It passes the minimum and maximum values by the `min` and `max` parameters.

```
void find_exterme_values(int array[], int *min, int *max)
{
    unsigned int i;
    *max = *min = array[0];
    for(i=1; i<NUMBER_OF_ELEMENTS; i++) {
        if(*min>array[i])
            *min=array[i];
        if(*max<array[i])
            *max=array[i];
    }
}
```

Searching For a Specific Value in an Unsorted Array

The Algorithm

The problem of locating a specific value in an array is very common. There are many variants of this issue. Here, we are interested in finding the first occurrence (starting from the first element) of the value in the array. The algorithm is quite simple. It checks all elements of an array one by one. If it finds the value in one of them it stops and returns the index of that element. Otherwise, if it does not find such a value in any of the elements, it returns a number, for example `-1`, that indicates that the array does not contain such a value. Such an algorithm is know as the *linear search algorithm*.

Searching For a Specific Value in an Unsorted Array

The Implementation

Below is a function that implements the linear search algorithm.

```
int find_value_index(int array[], int value)
{
    unsigned int i;
    for(i=0; i<NUMBER_OF_ELEMENTS; i++) {
        if(array[i]==value)
            return i;
    }
    return -1;
}
```

Searching For a Specific Value in an Unsorted Array – a Different Approach

The Algorithm

In the function from the previous slide, in each iteration of the `for` loop two conditions are checked. The first one is in the header of the loop and it tests whether there are any elements of the array left to visit. The second one is in the conditional statement and it checks whether the currently visited element stores the value that the function searches for. It is possible to simplify the loop. To do that an array is needed with one additional element at the end of it. In that element is stored ... the desired value! That enables the reduction of conditions tested in the loop. Now, it is only required to check whether the value is found. After the loop terminates it is necessary to check the value of the variable used for indexing the array. If it indicates the last element in the array, the function should return `-1`, because the value is not among the original values in the array. If not, it specifies the first element of the array containing the desired value. The algorithm is called a *sentinel linear search* or a *guarded linear search*.

Searching For a Specific Value in an Unsorted Array – a Different Approach

The Implementation

The following function implements sentinel linear search:

```
int faster_find_value_index(int *array, int value)
{
    int larger_array[NUMBER_OF_ELEMENTS+1];

    memcpy(larger_array,array,sizeof(array[0])*NUMBER_OF_ELEMENTS);
    larger_array[NUMBER_OF_ELEMENTS]=value;

    unsigned int i = 0;
    while(larger_array[i]!=value)
        i++;
    return (i!=NUMBER_OF_ELEMENTS)?i:-1;
}
```

The original values are copied to the `larger_array` with the use of `memcpy()` function. The array that stores them is passed to the function by a pointer. Thus its size is calculated as the product of the size of its first element and the total number of its elements.

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!