This course has already ended.

Third round tasks as a `ZIP file.`

This section focuses on one of the most important concepts of the C language: **Pointers.** Pointers allow processing of **arrays**, **strings** and various other C-programming variables. After reading through this section, you will understand pointers and operations allowed on pointers, how to deal with array data using pointers, as well as know how to write simple programs in which pointers are used. In future sections, pointers are used in practice constantly, so in this course you will be practicing how to use them in abundance.

Arrays¶

Basics¶

Multiple values of a given data type in consecutive memory slots is called an array. When array is declared, its size is given so that the system to allocate enough memory for the array. In variable declaration, array size is indicated using square brackets.

Below is a simple example of how to use arrays:

```
1  #include <stdio.h>
2  
3  int main(void)
4  {
5      short apples = 10;
6      short slots[4];
7      short oranges = 20;
8  
9      int i;
10     for (i = 0; i < 4; i++) {  /* Initialize the array with numbers */
11         slots[i] = i + 1;
12     }
13 
14     for (i = 0; i < 4; i++) {  /* Print the numbers in the array */
15         printf("array element %d is %d\n", i, *(slots + i));
16     }
17 
18     printf("apples: %d     oranges: %d\n", apples, oranges);
19 }
```

The program defines three variables: **'apples'** and **'oranges'** are normal short integers, but **'slots'** is an array that has four *short* data type elements. The array size must be given as constant. As with other data types in C, the elements of an array are not initialized by default, so their initial values are unknown. After three declarations, variables are placed in memory (roughly) in the following way:

../../_images/array.jpg

In memory, the array takes the given number of consecutive slots, with size depending on the data type used. In case of short integer, each element takes 2 bytes (16 bits), so in total the four-element array uses eight bytes.

Following the variable declarations, array is initialized in a for loop (line 10). This example shows the use of array notation (slots[i]) for accessing a particular array element. The first element is always indexed as 0. When indexing an array, any form of expression can be used (variable, constant, arithmetic operation, ...).

Arrays and pointers have a close relationship, as can be seen in the last part of the above example. The 'slots' array can be represented as a pointer to its first element, and the different elements of the array can be accessed using pointer arithmetics. The **printf** call on line 15 that outputs the contents of the array shows an example of the pointer arithmetic usage. Using slots[i] is equivalent with *(slots + i).

C compiler does not check at compile time whether array index is within the array bounds, and the compiler allows indexes beyond 4 in above example. Such errors can be detected only when running the program, and even then they don't always clearly show up. If the for - loop had been wrong and processed indexes beyond 4, it would overwrite other content in the memory. The following small modification to above code (the for loops erroneusly go through 6 items):

```
#include <stdio.h>

int main(void)
{
    short apples = 10;
    short slots[4];
    short oranges = 20;

    int i;
    for (i = 0; i < 6; i++) {  /* Here we initialize the array */
        slots[i] = i + 1;
    }

    for (i = 0; i < 6; i++) {  /* Output the values in array */
        printf("array element %d is %d\n", i, *(slots + i));
    }

    printf("apples: %d     oranges: %d\n", apples, oranges);
}
```

caused the value of 'apples' to become 5 in my test run. The compiler does not warn anything, and also the program seems to work, if one does not pay careful attention to the variable values. However, the memory neighbouring to the array is silently modified. For their near-invisible nature, these sort of errors have been commonly used in attacks that can turn the programming errors into security vulnerabilities. (Heartbleed is a recent example of a severe security vulnerability caused by buffer over-read bug, although in a bit different context)

Like other kinds of variables, also array can also be initialized with declaration. Here are two ways of initializing an array:

```
short slots[5] = { 3, 7, 2, 123, 45 };
int numbers[] = { 67, 12, 34 };
```

In first of the above lines, the array length is explicitly specified. When defined together with an initialization list, the array length does not need to be separately given, because the length of the initialization list indicates the length of the array. In such case, just empty square brackets are enough, as in the case of 'numbers' above. If initialization list is shorter than the explicitly given array length, the unspecified values are initialized to 0. This is true only if the initialization list contains at least one element.

```
int table[1000] = { 0 };
```

Arrays in function¶

Arrays can be passed as function arguments, but the array operator is usually not used in that case. Arrays are rather passed using a pointer type, where the pointer refers to the first element in array. The length of the array cannot be read from the function parameter, but it needs to be indicated by some other means. Here are some solutions:

- Pass the length as another function parameter which tells directly how many elements are in the array
- Indicate the end of an array by some special value after the last element. In this latter case the allocated length of the array needs to have extra space for the end marker.
- If the data type of the array is integer, its size can be told in the first element of the of the array (data communication protocols are designed like this)
- etc...

If the array size is declared as a part of the array itself, some memory must be allocated for this size declaring "extra" part of the array as well. In C programming, there is no way to get the size of an array during runtime (like there is for Python or Java); the programmer must know how many indexes are in the array at all times.

Array cannot be returned as a function return parameter (at least without dynamic memory that will be introduced in later sections). If function should return an array, the space for the array needs to be allocated either outside the function, or it needs to be allocated dynamically (this will be covered in the next modules).

Here is an example of using array with function. It also shows how the sizeof operator can be used to determine the total size of the array in bytes (later section for more information). One way of getting the number of elements in the array is to divide its total size by size of one element, as done in this example. The example shows, once more, how pointer type parameter is used to access the array inside the function.

```
1  #include <stdio.h>
2  
3  void show_table(short *a, size_t n)
4  {
5      int i;
6      for (i = 0; i < n; i++) {
7          // print the table using pointer arithmetics:
8          printf("%d ", *(a + i));
9  
10         // Also this would produce same result:
11         printf("%d ", a[i]);
12 
13         // We could also do this for same effect, but pointer 'a' is modified.
14         // Therefore, this cannot be used together with indexing, as above.
15         // Modification of pointer 'a' is not visible outside the function.
16         //printf("%d ", *a++);
17     }
18     printf("\n");
19 }
20 
21 int main(void)
22 {
23     short table[] = { 1, 4, 6, 8};
24     printf("size: %lu\n", sizeof(table)); /* print array size for fun */
25 
26     /* below is one way to get the number of elements */
27     // sizeof(table) is 4 * sizeof(short) == 8;
28     show_table(table, sizeof(table)/sizeof(short));
29 
30     // in this case the above would be equivalent to:
31     show_table(table, 4);
32 }
```

Task 3.3: Show Table¶

Change the above program in the following way:

- Remove second (additional) print statement from *show_table* - function (line 8 and 11)
- Change table size to **30**, and data type to **long**
- Beginning element of the array **must be 3**.
- Set the next element of the array such that it is **two times** higher than the preceding array element.

Do not remove show_table function or the array's size when calling the print function from main. However you might need to update the parameters a bit.

When program works, it must print the following.

```
size: (some size here)
3 6 12 24 48 96 ....
```

and so on..

Points 10 / 10 My submissions 1 Deadline Friday, 18 June 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Show Table

Select your files for grading

main.c

Choose File No file chosen

Submit

Task 3.4: Arrays¶

In this task, we continue to learn more about arrays and pointers. Complete functions need to be implemented in this task.

In this task, implement the following two functions.

1) You should implement function `int array_sum(int *array, int count)` that gets a pointer to the beginning of array of integers in consecutive slots in memory, and calculates the sum of the integers. The number of integers to be counted is given in parameter 'count'. The 'sum' is returned as the return value of the function.

For example, the following code:

```
int valarray[] = { 10, 100, 1000 };
int ret = array_sum(valarray, 3);
```

should set ret to 1110.

2) Implement function `unsigned int array_reader(int *vals, int n)` that reads integer values using scanf into pre-allocated array ('vals'). The numbers read from input are separated with whitespace (space, tab, newline,...) that is the default field separator for scanf function, i.e., you should be able to use the basic scanf format string for decimal numbers. Parameter 'n' gives the maximum length of the array, and the maximum number of values to be read.

If user does not give a valid integer (as can be seen from return value of scanf), the array ends, even if the maximum size was not yet reached. The function returns the final size of the array at the end, which can be smaller that the incoming 'n' parameter, if the user finished the input with non-valid integer.

Below is an example how this function can be tested:

```
int array[10];
unsigned int n = array_reader(array, 10);
printf("%d numbers read\n", n);
unsigned int i;
for (i = 0; i < n; i++) {
    printf("%d ", array[i]);
}
```

For example, the following input should cause the first four array elements to become 5, 8, 2, and 7, and then terminate because the fifth field read is not a decimal number:

```
5 8 2 7 -
```

Implement therefore '.c' - file for the above two functions, as well as their main function to test. For both properly operating function you will get 15 points (no warnings must be present).

Points 30 / 30 My submissions 1 Deadline Friday, 18 June 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Arrays

Select your files for grading

source.c

Choose File No file chosen

Submit

Task 3.5: Count chars¶

This task handles an array that consists of characters. The end of the array is marked with value 0.

You will implement two functions, of which both will yield part of the points in the task.

(a): Implement function defined as `unsigned int arraylen(const char *array)`. The function gets argument *array*. You will not know the length of the array at the time of calling the function, but you will know that it ends in value 0. The function should return the number of members in the array, not including the ending 0 value.

(b): Implement function defined as `void countchars(const char *array, unsigned int *counts)`. Function gets argument *array*, that functions as above described. The second parameter *counts* refers to an array with 256 members, to which you should count each of the characters in *array*. Remember that char type consists of 8 bits, so the table can fit all possible char values. For example, if *array* is `[ 'a','b','r','a','c','a','d','a','b','r','a',0 ]`, then `counts['a']` gets 5 as its value, and `counts['b']` gets 2 as its value. Note that for example character 'a' os just one way to represent integer (with decimal representation 97 based on ASCII table). You can assume that the *counts* array is initialized with zeros by the calling function.

You can use the following program to test your functions. Add the function implementations in front of the below code.

```
void printcounts(int min, int max, const unsigned int *counts) {
    for (int i = min; i <= max; i++) {
        printf("%c: %u  ---  ", i, counts[i]);
        if ((i - min + 1) % 6 == 0)
            printf("\n");
    }
}

void printarray(const char *array) {
    printf("{ ");
    while (*array) {
        printf("'%c',", *array);
        array++;
    }
    printf("0 }");
}

int main()
{
     unsigned int counts[256] = { 0 };

     char sample[] = { 'a','b','r','a','c','a','d','a','b','r','a',0 };

     printf("%s, length: %u\n", sample, arraylen(sample));
     countchars(sample, counts);
     printcounts('a', 'z', counts);

    return 0;
}
```

Points 20 / 20 My submissions 2 Deadline Friday, 18 June 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Count chars

Select your files for grading

source.c

Choose File No file chosen

Submit

Task 3.6: Sort¶

Write function `void sort(int *start, int size)` that sorts the integers in the given array into an ascending order (from smallest to largest). You can use the selection sort algorithm: first find the smallest number in array, and swap it with the first element of the array. Then do the same starting from the second element of the array, moving on to third, fourth, etc. until the whole array is processed. Test the function with different arrays of different size.

Points 25 / 25 My submissions 1 Deadline Friday, 18 June 2021, 19:59 To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Sort

Select your files for grading

source.c

Choose File No file chosen

Submit