Fundamentals of Programming 2

Pointers and Dynamically Allocated Variables

Arkadiusz Chrobot

Department of Computer Science

February 24, 2020

Outline

Pointers — Short Revision

A *pointer variable* or a *pointer*, for short, does not store directly any meaningful information, instead it stores an address of a single memory cell or the first memory cell from a group of adjacent memory cells that store data processed by a program. Such a variable is declared in the following way:

```
data_type *pointer_name;
```

In this case the data type describes not the type of information stored in the pointer (it is always an address), but the type of of data stored in a *pointed variable*, which is the aforementioned memory cell or group of memory cells. Hence, the declaration: `int *integer_pointer;` should be read, as *a pointer to a variable of the `int` type*, but usually the more convenient (although quite confusing) form is used: *pointer of the `int` type*. It is also possible to declare a pointer that points to data of an unspecified type, using the following pattern:

```
void *pointer_name;
```

The type of the data can be defined/modified with the use of casting.

Pointers — Short Revision

Pointers Values

Pointers always store only *addresses of memory cells*. A pointer that points to nothing is called an *empty pointer* and it has a value described by the `NULL` constant. The ISO C99 allows using the `0` number instead of the aforementioned constant. Global pointers are empty pointers by default. Local pointers have an unspecified default value. In Computer Science jargon such pointers are called *wild pointers*. Using them without proper initialization is very dangerous, because it may cause damage to information stored in any of program's memory cells and result in aborting the program. The `printf()` function and its derivatives allow the programmer to display or write to a file the content of a pointer with the use of the `%p` formatting string. It is not allowed to use this string with the `scanf()` function and its derivatives.

Pointers — Short Revision

Data Access

If a pointers points to data of a specified type, then it is possible to get the data by *dereferencing* the pointer variable. In the C language the `*` ("start" or "asterisk") symbol is used as the dereference operator. It is the same symbol which is used in pointer declaration. To dereference a declared and initialized pointer it is necessary to put the symbol before its name. The pointer can be initialized by assigning it an address of a local or global variable with the use of the `&` (ampersand) operator, which is called the *address operator*. This operator ought to be placed before the name of the variable which address it should return. If a pointer points to a structure or an union then accessing its members can be done twofolds:

```
structure_name->field_name
```

or:

```
(*structure_name).field_name
```

It is recommended to use the first notation, because its shorter and more legible.

Pointers — Short Revision

Pointer Arithmetic

Pointers, or more specifically the addresses stored inside of them can be compared with use of the same operators that are used for comparing, for example, the integer numbers. Additionally, the C language allows using the *pointer arithmetic* which means that it makes it possible to add to a pointer or subtract from a pointer an integer number. Moreover, two pointers can be subtracted from each other, but they cannot be added. Also, the increment and decrement operators may be applied to the pointers. The pointer arithmetic can be used, for example, with arrays. However, it should be applied cautiously, and whenever it is possible to avoid it, the arithmetics should be not used.

Pointers — Short Revision

Using Pointers in Functions

The pointers may be applied as parameters of functions. The arguments passed by such parameters are stated to be passed by a pointer or by an address. Pointer parameters are used in the function body just like regular pointers. The name of the argument passed by a pointer parameter has to be prefixed by the address operator (`&`). There are two exceptions from the rule. The first one are pointers and the second one are the linear arrays which names in the C language are (almost) equivalent to the pointers. The argument data type has to be the same as the data type of the pointer parameter, with the exception of the `void` pointer parameter. Changing data pointed by the pointer parameters results in the change of the value of arguments passed by them — the arguments are pointed variables.

Functions may return the values of pointers (addresses). In that case the data type of function's returned value should be declared as a pointer type. Returning an address of any local variable is dangerous and discouraged.

Pointers — Short Revision

Example

```
#include <stdio.h>

int main(void)
{
  int *pointer = NULL;
  int variable = 3;
  pointer = &variable;
  printf("The value of the pointed variable: %d\n",*pointer);
  printf("The address stored in the pointer: %p\n",pointer);
  variable++;
  printf("The value of the pointed variable: %d\n",*pointer);
  *pointer+=1;
  printf("The value of the pointed variable: %d\n",variable);
  return 0;
}
```

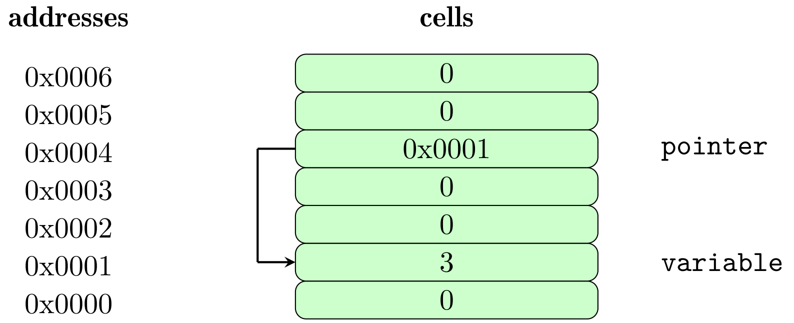
Pointers — Short Revision

Comment

By tracing the program from the previous slide it is possible to find out that the value of the pointed variable can be changed with the use of the pointer, and also the modified valued can be read using the same pointer. Please note the difference between reading the value of the pointer (the address it stores) and reading the value of the variable pointed by the pointer.

Pointers — Short Revision

To better understand how the pointers work, let's take a look at a very simplified model of the computer memory, in which each variable has the size of a single cell. The variables from the example program could be arranged in the memory in the following way:



Pointers — Short Revision

The picture from the previous slide is for reference only. The addresses of cells in the drawing are not real. They are simplified in order to make the depiction of memory easy to understand. In reality the variable may be arranged differently than they are in the picture. There is also one more simplification made in the drawing — it is assumed that every cell in the memory is capacious enough to store an address (to serve as a pointer) and a value of the `int` type. Real cells usually have the size of a single byte and the size of pointers and `int` type variables depends on the architecture of the computer that runs the program. Most of contemporary computers have a 64-bit architecture and thus the size of both data types is eight bytes. For the 32-bit computers the size is four bytes. That means that the address as well as the value are stored in more than one memory cell. The addresses are written using the hexadecimal numbers. One hexadecimal digit is equivalent to four bit, so the addresses in the picture are 16-bits wide.

Function Pointers

Functions just like data are stored in cells of the RAM. Hence, just like regular data functions may be pointed by pointers and also can be invoked. Even those functions that take arguments. Function pointers have to have a specific data type. For example if a function doesn't return any value (or in other words: returns `void`) and doesn't take any arguments, then the function pointer should be declared as follows:

```
void (*function_pointer)(void);
```

Please notice how the parentheses are used. Without them the declaration would describe a prototype (header) of a function that takes no arguments and returns a pointer of an unspecified type. The pointer to a function that takes two arguments of `int` type and returns a value of the same type could be declared in the following way:

```
int (*another_function_pointer)(int, int);
```

Function Pointers

The declarations of function pointers can be even more complicated. The topic will be discussed at the end of the lecture. It is possible to create structures and unions with fields that are function pointers or arrays with elements that are such pointers. In the next slides two example programs are presented that show how to use the function pointers declared in the previous slide.

Function Pointer

Example — Simple Function Pointer

```c
#include<stdio.h>

void say_hello(void)
{
    puts("Hello there!");
}

int main(void)
{
    void (*function_pointer)(void) = 0;
    function_pointer = say_hello;
    function_pointer();
    return 0;
}
```

Function Pointers

Example — Comment

In the program from the previous slide, the `say_hello()` function is declared that takes no arguments and returns no value. In the `main()` function of the program, a pointer to the aforementioned function is declared. It is a local pointer, thus it is initialized with `0` in the place of its declaration. Lack of initialization for such a pointer is not signaled by the compiler, but it may have dangerous consequences during the program run. Thus its initialization is always recommended. In the next line of the `main()` function a curious assignment is made. Literally it can be interpreted as assigning the function name to a pointer. In reality, the name of a function in the C language is (almost) equivalent to a pointer, just like in the case of an array. So, in that line the address of the `say_hello()` function is assigned to the function pointer. The code of the program can be made a little shorter by replacing the two described lines of the `main()` function with the following one:

```
void (*function_pointer)(void) = say_hello;
```

Function Pointers

Examples — Comment (Continued)

The statement in the next line looks like an invocation of the function, but with the pointer name used instead of the function name. Indeed, in the line the function is called, but indirectly, with the use of the pointer variable that points to the function. The parentheses `()` placed behind the pointer are the *function call operator*. It orders the computer to activate the function. If the function needed arguments then their list would be embraced by those parentheses, what is shown in the next program.

Function Pointers

Example — More Advanced Function Pointer

```c
#include<stdio.h>

int add_up(int first, int second)
{
    return first+second;
}

int main(void)
{
    int (*another_function_pointer)(int, int) = NULL;
    another_function_pointer = add_up;
    printf("Adding %d to %d gets %d.\n",3,2,
                another_function_pointer(3,2));
    return 0;
}
```

Function Pointers

Example — Comment

In the program from the previous slide a more advanced function is defined that takes to numbers as arguments and returns their sum. A pointer to such a function is declared and initialized in the `main()` function. This time it is assigned the value of the `NULL`, to show that it also can be applied in such a case. In the next line the pointer is assigned the address of the `add_up()` function. Like in the previous example those two lines can be replaced by the following one:

```
    int (*another_function_pointer)(int, int) = add_up;
```

The `add_up()` function has two parameters, thus when it is invoked two arguments have to be passed to it. The same has to be done when it is invoked with the use of the pointer. In the case of the program those arguments are two numbers: `3` and `2`. Please note, that the value returned by the function is an argument of the `printf()` function call, that displays the result on the screen.

Dynamically Allocated Variables

The pointers play one more important role in programming. They allow using dynamically allocated variables. Before this term will be explained, let's review the information about the scope of variables:

- *global variables* — created in the data area of the program's memory; exist through the whole life cycle of the program; initialized by default with the zero value.
- *local variables* — also called *automatic* variables; associated with functions; created on the call stack (an area of program memory) when the function is called; are a part of an activation record (a stack frame); not initialized by default; cease to exist when the function terminates; their scope depends on the place of their declaration.

Dynamically Allocated Variables

The dynamically allocated variables have some properties of both of the kinds described in the previous slide. The programmer decides about their scope and life cycle. Hence the name — they are created and destroyed when the program is running. Those variables are created in a area of the program memory that is called a *heap*, with the use of dedicated subroutines that are standard elements of a programming language. In case of the C language they are functions and are described in the next slides. The subroutines that create dynamically allocated variables allow the programmer to specify the size of the variable or in other words the number of memory cells that constitute the variable, but they do not allow her or him to give it a name. Those subroutines return the address of the new variable, which can be stored in a pointer. Thus the pointer becomes the only link between the variable and the rest of the program. After such an assignment the dynamically allocated variable becomes a pointed variable. Additionally, if the pointer is of a specific type, it also determines the type of the dynamically allocated variable.

Dynamically Allocated Variables

It is possible to point a single dynamically allocated variable with several pointers. This allows for interpreting the same data stored in the variable in different ways. However, this is a complicated case and won't be further discussed in the lecture. The operation of creating a dynamically allocated variable boils down to making a reservation of a continuous area of memory for the variable in the heap and it is called an allocation. It is not a trivial operation and it can fail. The way the allocation is done depends on the computer and operating system internal workings, but the details won't be discussed in the lecture. However, it has to be stated, that all allocated memory on the heap must be freed when the dynamically allocated variables are no longer used or before the program terminates. The operation of freeing the memory is carried out with the use of another subroutines, which mark the allocated memory as free, i.e. ready to be used for another dynamically allocated variables. Freeing memory is also called a deallocation of a variable and is equivalent to destroying it.

Dynamically Allocated Variables

Heap Handling Functions in the C Language

There are four functions in the C language responsible for managing (allocating and freeing) the heap. They are described in tables in this and following slides.

Function Name	**Description**
`malloc()`	The function takes only one argument, which is an expression defining the size (in bytes) of the memory area which is to be allocated in the heap. The returned value is of the `void *` type and it is the address of the first memory cell from the group of cells that belong to the allocated area. The address is called the address of the dynamically allocated variable (memory area) or a pointer to the dynamically allocated variable (memory area). If the function fails to allocate memory, it returns the `NULL` value. The allocated memory is uninitialized.

Dynamically Allocated Variables

Heap Handling Functions in the C Language

Function Name	**Description**
`calloc()`	This is actually a form of the `malloc()` function, which is designed to simplify the allocation of memory for arrays. It takes two arguments. The first one is the number of elements of the dynamically allocated array and the second one is the size of a single element. The array is initialized with zeros.

Dynamically Allocated Variables

Heap Handling Functions in the C Language

Function Name	**Description**
`free()`	The function is responsible for freeing the memory. It returns no value, but takes the pointer to the memory to be freed as its argument. The memory should be previously allocated by one of three function that can do it, otherwise a serious exception may occur and the program may be aborted. If an empty pointer is passed to the function, it will take no actions. It should be noticed, that the function doesn't zero out the memory area that it deallocates, it just marks it as free. The data stored inside the area still exists, but they mustn't be accessed. The function also doesn't zero out the passed pointer and as long as it is not assigned a new address it mustn't be used. In Computer Science jargon such a pointer is called a *hanging pointer*.

Dynamically Allocated Variables

Heap Handling Functions in the C Language

Function Name	**Description**
`realloc()`	The function modifies the size of the allocated memory area in the heap. It takes two arguments. The first one is the pointer to that area, and the second one is the new size expressed in bytes. The function returns an address of the modified memory area (the value of the `void *` type) or `NULL` if it failed. The returned address may be different from the passed address, in case the function has to overcome obstacles in resizing the area by coping the data stored in it to another memory area. If the memory area is expanded, the data inside it are preserved. However, if the area is shrank, a data loss may occur. If an empty pointer is passed to the function it will behave like the `malloc()` function and if the new size is set to `0` the function will behave like the `free()` function.

Dynamically Allocated Variables

All the functions described in tables are declared in the `stdlib.h` header file. Declarations of two other useful functions are in the `string.h` header file. The first function was already introduced. It is the `memset()` function that stores a specified value in a memory area pointed by a pointer. It takes three arguments. The first one is the pointer (of the `void *` type) to the memory area, the second one is the value (of the `int` type) to be stored in the area and the last argument is the size of the area expressed in bytes. The `memset()` function returns the `void *` pointer to the area which now stores the value. The second useful function is the `memcpy()`, which copies the content of one memory area to another. It takes three arguments. The first one is a pointer to the destination area and the second one is the pointer to the source area. Both of them are of the `void *` type. The last argument is the size of data to be copied. The function returns the pointer (of the `void *` type) to the destination area.

Dynamically Allocated Variables

Example — Dynamically Allocated Variable of `int` Type

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

int main(void)
{
    int *variable = (int *)malloc(sizeof(int));
    if(variable) {
        printf("The address of the dynamically allocated variable:\
                                                    %p\n",variable);
        *variable = 24;
        printf("The value of the dynamically allocated variable:\
                                                   %d\n",*variable);
        free(variable);
        variable=NULL;
    }
    return 0;
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Variables of `int` Type

In the previous slide a simple, not split into functions, program is presented that uses a dynamically allocated variable of the `int` type. The variable is created by invoking the `malloc()` function. The size of the variable is calculated with the use of `sizeof` operator applied to the `int` type. The value (the address) returned by the `malloc()` function is casted to the `int *` type and stored in the pointer named `variable`. After the program checks if the memory allocation was successful, the address stored in that pointer is displayed on the screen and then a number `24` is stored in the dynamically allocated variable. Next, the value of the variable is displayed on the screen. After completing all described actions the program deallocates the dynamically allocated variable using the `free()` function and assigns the `NULL` constant value to the pointer. No operation can be carried out with the use of dynamically allocated variable until the program makes sure that the allocation operation was completed successfully.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

The next slides contain a source code of a program that uses a dynamically allocated array. The memory for that array is allocated with the use of `calloc()` function.

Dynamically Allocated Variables

Example — Comment

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

#define NUMBER_OF_ELEMENTS 20

void fill_array(int *array)
{
    int i;
    for(i=0;i<NUMBER_OF_ELEMENTS;i++)
        array[i]=i;
}
```

Dynamically Allocated Variables

Example — Comment

The beginning of the source code of the program that demonstrates the use of dynamically allocated array is presented. It's very similar to the beginning of program that uses a regular (i.e. statically allocated) array. A constant is defined that describes the number of the array's elements and a function which assigns the value of indices to the elements of the array. The declaration of the function's parameter may be changed to `int array[]`. In the C language the name of an array is (almost) equivalent to a pointer. That means that the same syntax can be used to handle both statically and dynamically allocated arrays.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

```c
void print_array(int *array)
{
    int i;
    for(i=0;i<NUMBER_OF_ELEMENTS;i++)
        printf("%d ",array[i]);
    puts("");
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Array

In the previous slide a function that displays the array content on the screen is presented. That function can also be applied to a statically allocated array.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

```
int main(void)
{
    int *array_pointer = (int *)calloc(NUMBER_OF_ELEMENTS,
                                                  sizeof(int));
    if(array_pointer) {
        fill_array(array_pointer);
        print_array(array_pointer);
        free(array_pointer);
        array_pointer = NULL;
    }
    return 0;
}
```

Dynamically Allocated Variables

Example — Comment

In the `main()` function a dynamically allocated array is created with the use of the `calloc()` function invocation. As the first argument the constant describing the number of elements of the array is passed to the function, and as the second one the size of a single element. In this case the size of `int` type. After the program makes sure that the allocation was successful it calls the functions that assign values to the elements of the array and print them on the screen. Finally, the array is deallocated and the pointer is assigned a `NULL` value.

Dynamically Allocated Variables

Example — Own Implementation of `calloc()` Function

The next example is a program which behaves in the same way as the previous one, but it uses its own implementation of the `calloc()` function. The function that reimplements the behaviour of `calloc()` is called `allocate_array()`.

Dynamically Allocated Variables

Example — Own Implementation of `calloc()` Function

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

#define NUMBER_OF_ELEMENTS 20

void *allocate_array(unsigned int number_of_elements,
                                 unsigned int element_size)
{
    unsigned long int array_size =
                         element_size * number_of_elements;
    void *array_pointer = malloc(array_size);
    if(array_pointer)
        array_pointer = memset(array_pointer,0,array_size);
    return array_pointer;
}
```

Dynamically Allocated Variables

Example — Comment

The `string.h` header file is included in the program to allow using the `memset()` function. In the program the `allocate_array()` function is defined which is the counterpart of the `calloc()` function. The number of elements of the array and the size of a single element are passed by the parameters to the function. First, the `allocate_array()` function calculates the total size of the array by multiplying the values of both parameters. Next, it allocates the memory for the array by calling the `malloc()` function. After making sure that the allocation was successful i.e. checking if the value of the `array_pointer` variable is different than `0` (`NULL`), the function zeroes out the content of the array by invoking the `memeset()` function. Regardless if the array was created and initialized or not, the function returns the value of `array_pointer` variable. This is consistent with the behaviour (semantics) of the `calloc()` function.

Dynamically Allocated Variables

Example — Comment (Continued)

Please note that the `allocate_array()` returns a value of the same type (`void *`) as the `calloc()` function. Its also worth noticing, that the product calculated in the first line of the function is stored in a variable of twice as capacious type as the type of the multiplication arguments, to avoid an overflow.

Dynamically Allocate Variables

Example — Own Implementation of `calloc()` Function

```c
void fill_array(int *array)
{
    int i;
    for(i=0;i<NUMBER_OF_ELEMENTS;i++)
        array[i]=i;
}

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

Dynamically Allocated Variables

Example — Comment (Continued)

The two functions presented in the previous slide are the same as in the previously described example.

Dynamically Allocated Variables

Example — Own Implementation of `calloc()` Function

```c
int main(void)
{
    int *array_pointer = (int *)
             allocate_array(NUMBER_OF_ELEMENTS, sizeof(int));
    if(array_pointer) {
        fill_array(array_pointer);
        print_array(array_pointer);
        free(array_pointer);
        array_pointer = NULL;
    }
    return 0;
}
```

Dynamically Allocated Variables

Example — Comment

In the `main()` function the calling of `calloc()` is replaced by invocation of the `allocate_array()` function. The way of using the functions in both presented programs is the same.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

One of the advantages of the dynamically allocated variables is that their size can be modified *while the program is running*. The next program creates an array with as many elements as the user requests when the program starts. The size of the array is hence described by the value of a variable instead of the value of a constant. The ISO C99 standard allows the programmer to declare a local arrays which size is also described by a value of a variable, but in the newer edition of the standard (ISO C11) that way of declaring an array is not recommended. It is possible that in the future editions of the standard this method of declaring an array will be forbidden. Using dynamically allocated variables for this purpose is generally a better idea. The next example shows how to do it.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

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

void fill_array(int *array, unsigned int number_of_elements)
{
    int i;
    for(i=0;i<number_of_elements;i++)
        array[i]=i;
}

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

Dynamically Allocated Array

Example — Comment

The `NUMBER_OF_ELEMENTS` constant is removed from the program. It is replaced by a variable of the same name, although written in lower cases. The `fill_array` and `print_array()` functions get an additional parameter. It is used for passing the number of elements of the array.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

```
int main(void)
{
    puts("How many elements should the array have?");
    unsigned int number_of_elements = 0;
    scanf("%u",&number_of_elements);
    int *array_pointer =
                (int *)calloc(number_of_elements, sizeof(int));
    if(array_pointer) {
        fill_array(array_pointer,number_of_elements);
        print_array(array_pointer,number_of_elements);
        free(array_pointer);
        array_pointer = NULL;
    }
    return 0;
}
```

Dynamically Allocated Variables

Example — Comment

In the `main()` function the aforementioned `number_of_elements` variable is declared. It is of the `unsigned int` type. In the variable the number of elements, given by the user, is stored. The value is used for allocating a memory for the array. Please notice while running the program, that the number of displayed values of the array elements is equal to the number inputed by the user to the program.

Dynamically Allocated Variables

Summary

The pointers and dynamically allocated variables may be applied for building a more complex and advanced data structures, than the arrays described in the lecture. Those structures will be presented soon.

How To Read Complicated C Declarations? [1]

Looking at the examples presented in the lecture it is easy to discover that the variables in the C language may have a complicated declarations. Function pointers are one of the examples. Fortunately, there is a rule that defines how to read such declarations:

The Rule

Start at the variable name (or innermost construct if no identifier is present). Look right without jumping over a right parenthesis; say what you see. Look left again without jumping over a parenthesis; say what you see. Jump out a level of parentheses if any. Look right; say what you see. Look left; say what you see. Continue in this manner until you say the variable type or return type.

[1] Based on an article by Terence Parr published here: https://parrt.cs.usfca.edu/doc/how-to-read-C-declarations.html

How To Read Complicated C Declarations?

The next slides contain a few examples of declarations with their descriptions. The names of the variables in examples are one letter long, to avoid giving away to soon the meaning of the declarations.

How To Read Complicated C Declarations?

Example no 1

Example

```
int *a[10];
```

How To Read Complicated C Declarations?

Example no 1

Example

```
int *a[10];
```

Answer

The `a` variable is an array of 10 pointers of the `int` type.

How To Read Complicated C Declarations?

Example no 2

Example

```
int (*x) (int *, int *);
```

How To Read Complicated C Declarations?

Example no 2

Example

```
int (*x) (int *, int *);
```

Answer

The `x` variable is a pointer to a function that has two pointer parameters of the `int` type and returns a value of the `int` type.

How To Read Complicated C Declarations?

Example no 3

Example

```
int (*(*v)[])();
```

How To Read Complicated C Declarations?

Example no 3

Example

```
int (*(*v)[])();
```

Answer

The `v` variable is a pointer to an array of pointers to functions that take unspecified number of arguments and return a value of the `int` type.

Questions

?

The End

Thank You For Your Attention!