Fundamentals of Programming 2

Pointers and Dynamically Allocated Variables

Arkadiusz Chrobot

Department of Information Systems

February 26, 2024

Outline

Pointers To Pointers

Recall that a pointer is a variable that can store *an address* of another variable. In the C language, it is possible to define a pointer that points to another pointer. Such a pointer is called *a pointer to a pointer*[1]. Its declaration follows the schema given below:

```
data_type **pointer_to_pointer;
```

The double asterisk means that it is a pointer that stores an address of another pointer. If there was another asterisk, then it would mean that this is a pointer to a pointer to a pointer. The number of asterisks in the pointer declaration determines its *level* or *the level of indirection*. The C language standard specifies that compilers should be able to handle at least up to 12 levels of pointers, but in most programs at most two or three levels are used.
The pointer to a pointer can be declared as a global or local variable or a function parameter.

[1]Some sources refer to it as a *double pointer*, but the name is quite confusing.

Pointers To Pointers

Example

```c
#include<stdio.h>

void display(int **pointer)
{
  printf("Address in the pointer to pointer: %p\n",pointer);
  printf("Address in the pointer : %p\n",*pointer);
  printf("Value in the variable: %d\n",**pointer);
}
```

Pointers To Pointers

Example

```c
int main(void)
{
  int first_variable = 5;
  int second_variable = 6;
  int *first_pointer = &first_variable;
  int *second_pointer = &second_variable;
  int **pointer_to_pointer = &first_pointer;
  display(pointer_to_pointer);
  pointer_to_pointer = &second_pointer;
  display(pointer_to_pointer);
  return 0;
}
```

Pointers To Pointers

Example — Comment

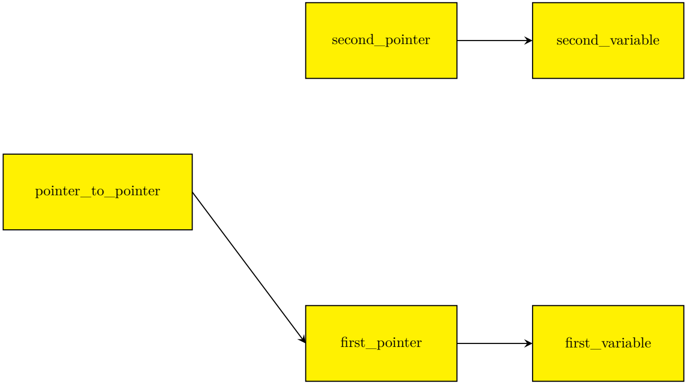
The program from the previous slides demonstrates how the pointer to a pointer works. The `display()` function has a parameter that is a pointer to a pointer, and accepts arguments of the same type. First, it prints the address stored in the parameter. This is an address of another pointer. Then it prints the address stored in the other pointer. To get this, it dereferences the parameter only once. Finally, the function dereferences the parameter twice, to print the value of the variable pointed by a pointer that in turn is pointed by the parameter.

In the `main()` function, two variables of the `int` type are declared and two pointers of the same type. The variables are initialized with numbers 5 and 6 respectively, and the pointers with the addresses of these variables. Finally, a pointer to a pointer is declared that initially points to the first pointer, and the `display()` function is invoked with the former pointer as an argument.

Pointers To Pointers

Example — Comment

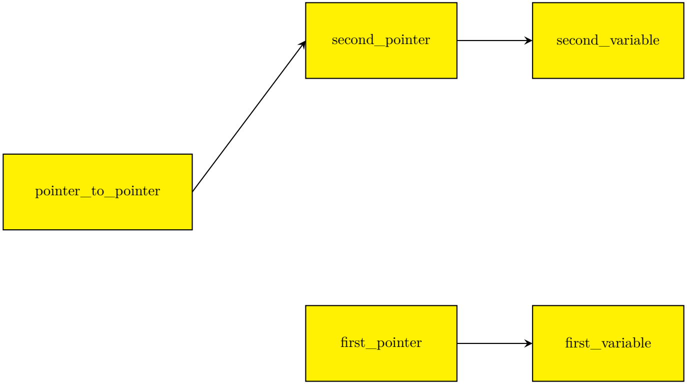
After that, the pointer to a pointer is redirected to the second pointer, and the `display()` function is called once more. This time it prints different data. The figure illustrates how the program works:



Pointers To Pointers

Example — Comment

After that, the pointer to a pointer is redirected to the second pointer, and the `display()` function is called once more. This time it prints different data. The figure illustrates how the program works:



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 these 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 put in these parentheses, as it is shown in the next program.

Function Pointers

Example — More Advanced Function Pointer

```
#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 two 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, these 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 these 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.

Function Pointers

Usage In The C Standard Library

Function pointers are quite often used in functions available in the C standard library. For example, the `qsort()` function that sorts an array using the Quick Sort algorithm (a very efficient algorithm for sorting arrays) as one of its parameters has a function pointer. The argument passed by this pointer is an address of a function that compares values of two elements of the array. The program, in the next slides, shows how to use the `qsort()` function to sort an array of integers.

Function Pointers

The qsort() Function Usage

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

#define LENGTH 20

void populate(int array[], unsigned int length)
{
    srand(time(0));
    for(int i=0; i<length; i++)
        array[i]=-10+rand()%21;
}
```

Function Pointers

The `qsort()` Function Usage — Comment

In the program are included three header files. The first one is the `stdio.h`, because the program displays messages on the screen using the `printf()` function. It also uses the PRNG, that's why the two other header files are included. There is however an additional reason for including the `stdlib.h` file — the `qsort()` function is declared there. The function `populate()` fills an array, passed as its first argument, with randomly chosen integers ranging from -10 to 10 (inclusive). The array's number of elements is passed as the second argument of the function.

Function Pointers

The `qsort()` Function Usage

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

int compare(const void *first, const void *second)
{
    return *(int *)first - *(int *)second;
}
```

Function Pointers

The `qsort()` Function Usage — Comment

The `print()` function displays the content of an array of integers, passed as its first argument. The array's number of elements is passed as the second argument of the function.
The `compare()` function is invoked by the `qsort()` function to compare values of two elements of the array. It has to return a negative number, if the value of the first element is less than the value of the second element. In case the value of the first element is greater than the value of the second, it has to return a positive number. If the values of both elements are equal, the function should return zero.
In order to be able to sort an array of any type, the `qsort()` function gets the array by a parameter that is a pointer of the `void *` type. It also passes to the `compare()` function two pointers of the same type, that point to elements of the array that values need to be compared.

Function Pointers

The `qsort()` Function Usage — Comment

The `compare()` function first casts the pointer parameters to the `int *` type, then it dereferences the pointers and subtracts the two numbers stored in elements of the array pointed by these pointers. The result is the number expected by the `qsort()` function and thus it is returned.

Function Pointers

The qsort() Function Usage

```
int main(void)
{
    int numbers[LENGTH];
    populate(numbers,LENGTH);
    print(numbers,LENGTH);
    qsort(numbers,LENGTH,sizeof(numbers[0]),compare);
    print(numbers,LENGTH);
    return 0;
}
```

Function Pointers

The `qsort()` Function Usage — Comment

In the `main()` function, an array of 20 elements is created and then populated with random numbers. Next, the content of the array is printed on the screen and the `qsort()` function is invoked in order to sort the array. The function takes as the first argument the array, as the second the number of elements in the array, and as a third the size of a single array's element. The last argument of this function is the address of the `compare()` function, or in other words, the pointer to that function. As it was mentioned before, in the C language, the name of the function is also a pointer to the function. After the array is sorted by `qsort()` its content is once again displayed.
Functions, which addresses are passed to other functions, are sometimes referred to as *callback* functions or simply *callbacks*. For example, the `qsort()` function invokes `compare()` when it is itself invoked.

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. These variables are created in an 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. These 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 other subroutines, which mark the allocated memory as free, i.e. ready to be used for other 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, that 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 in the group of cells that belong to the allocated area. This 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 functions 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 *dangling 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 fails. The returned address may be different from the passed address, in case the function has to overcome obstacles in resizing the area by copying 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 shrunk, 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 has already been introduced. It is the `memset()` function that assignd a specified value to 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 that now stores the value. The second useful function is the `memcpy()`, which copyies 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 copyied. 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 the operations, 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 program that uses a dynamically allocated variable of the `int` type doesn't show the full potential of these variables. The next program creates a resizeable array of integers. The number of elements in the array is not known before running of the program. It is incremented during the runtime to store new values.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

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

Dynamically Allocated Variables

Example — Dynamically Allocated Array

In the program are included three header files: `stdio.h`, `stdlib.h` and `time.h`. The program not only displays messages on the screen and uses a dynamically allocated array, but also applies the PRNG.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

```
int *add_element(int *array, int index, int value)
{
   static unsigned int length;
   const unsigned int DELTA = 10;
   if(index>=length) {
      length += DELTA;
      int *new_array = realloc(array, length * sizeof(int));
      if(new_array)
         array = new_array;
   }
   if(array)
      array[index]=value;
   return array;
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Array

The `add_element()` function is responsible for adding a new value to the array. It takes three arguments: the array (more specific — a pointer to the array), the index of the element and the value itself. First, it checks if the index is equal or greater than the current number of elements in the array. This number is stored in a local variable declared as `static`, which means that it is not destroyed after the function terminates, but exists for the entire runtime of the program. Its initial value is 0. If the condition is met, the function increments the number of elements by DELTA, which is defined as a constant of the value 10, and allocates memory for the array using the `realloc()` function. The address returned by `realloc()` is stored in the `new_array` pointer. Because the allocation may fail, the `add_element()` function first checks if the `new_array` pointer is not `NULL` before assigning its content to the `array` pointer. Finally, the function verifies if the `array` pointer is not empty (the first allocaton also may have fail). If it is valid, then the function stores the value in an element specified by the index and returns the address of the array.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

Please notice, that it is possible to return an address of a variable from a function. It only requires, aside from using a valid return statement, declaring the return type of the function as a pointer of a specified type. It is strictly forbidden to return an address of an automatic local variable, no matter if the address was obtained with the use of the address operator (`&`) or by other means. Such a variable ceases to exists when the function terminates, so that address becomes immediately invalid. The only local variable which address may be safely returned is a variable declared with the use of the `static` keyword. The `add_element()` function returns the address of a dynamically allocated variable (an array), which is only stored in a local pointer. It means that it can be safely returned from a function.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

```c
void print_array(const int *array,
                 const unsigned int number_of_values)
{
    if(array) {
        printf("The number of values in the array is %u\n",
        number_of_values);
        for (int i = 0; i < number_of_values; i++)
            printf("%d ", array[i]);
        puts("");
    }
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Array

The `print_array()` function displays the content of the array. It takes as arguments the pointer to the array and the actual number of values stored inside the array. Please notice, that the array may have more elements than it stores values, because each time it is resized ten new elements are added. Also notice, that the `print_array()` function verifies if the pointer to the array is valid before using it. It may have happened, that the `add_element()` function entirely failed to create the array, so the checking is necessary. Besides the values themselves, the function also prints the number of values stored in the array.

Dynamically Allocated Variables

Example — Dynamically Allocated Array

```c
int main(void)
{
    int *array = NULL;
    srand(time(0));
    unsigned int number_of_values = 1000 + rand()%1001;
    for(int i=0; i<number_of_values; i++)
        array = add_element(array,i, -5+rand()%26);
    print_array(array, number_of_values);
    free(array);
    return 0;
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Array

In the `main()` function, the pointer to the array is initialized with `NULL`, which means that the array initially doesn't exist. Moreover, the program chooses (pseudo)randomly the number of values that it will store in the array. The number may range from 1000 to 2000. The array is created and populated in the `for` loop. Each time the `add_element()` function determines that there are no more elements in the array to store another value, it creates 10 more. The number was selected arbitrarily. The greater it is, the more elements could be left unused. The less it is, the more often the program allocates memory, which is a time-consuming operation. If the program was to be used for something serious, then perhaps a different number should be picked. After the array is filled with numbers, the `print_array()` function prints its content and the program releases the memory with the help of the `free()` function.

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

It is possible to create and use a dynamically allocated matrix (two dimensional array) with the help of a pointer to a pointer. The next program allows the user to determine how many rows and columns such a matrix should have and then it creates the matrix and fills with pseudorandomly chosen integer numbers ranging from -10 to 10.

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

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

int **create(const int rows, const int columns)
{
    int **matrix = (int **)calloc(rows,sizeof(int *));
    if(matrix) {
        for(int i=0; i<rows; i++)
            matrix[i] = (int *)calloc(columns,sizeof(int));
    }
    return matrix;
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

The `create()` function is responsible for allocating the memory for the matrix. First, it creates an array of pointers of the `int *` type. The number of the elements in the array is specified by the value of the `rows` parameter. If the operation is successful, then in the `for` loop the memory for arrays of integers is allocated. Addresses of these arrays are stored in the elements of the array of pointers. The number of elements in each of the arrays of integers is determined by the value of the `columns` parameter. Finally, the address of the array of pointers is returned by the function. Please notice, that the address is stored in a local pointer to a pointer called `matrix` and that the function also returns a pointer to a pointer.

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

```
void fill(int **matrix, const int rows, const int columns)
{
    for(int i=0; i<rows; i++)
        for(int j=0; j<columns; j++)
            matrix[i][j] = -10+rand()%21;
}

void print(int **matrix, const int rows, const int columns)
{
    for(int i=0; i<rows; i++) {
        for (int j = 0; j < columns; j++)
            printf("%4d", matrix[i][j]);
        puts("");
    }
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

The `fill()` function populates a matrix with the randomly selected integers ranging from -10 to 10 (inclusive) and the `print()` function prints its content. Please notice, that the functions are very similar to the functions that have been defined in the previous semester for identical operations on “regular” matrices or, to describe them correctly, statically allocated matrices. The difference is that the functions from the previous slide get the matrix by a parameter that is a pointer to a pointer, and that the number of rows and columns is specified by parameters of that names.

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

```
void release(int **matrix, const int rows)
{
    for(int i=0; i<rows; i++)
        free(matrix[i]);
    free(matrix);
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

The `release()` function is responsible for freeing the matrix. First, it deallocates in the `for` loop the memory allocates for each of the arrays of integers. Next, it frees the memory allocated for the array of pointers.

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

```
int main(void)
{
    srand(time(0));
    puts("Please specify the number of rows and columns.");
    puts("Rows?"); int rows = 0;
    scanf("%d",&rows);
    puts("Columns?"); int columns=0;
    scanf("%d",&columns);
    int **matrix = create(rows,columns);
    if(matrix) {
        fill(matrix,rows,columns);
        print(matrix, rows, columns);
        release(matrix, rows);
    }
    return 0;
}
```

Dynamically Allocated Variables

Example — Dynamically Allocated Matrix

In the `main()` function, the program asks the user about the number of rows and columns that the matrix should have. Then the array is created with the help of the `create()` function. Next, the program verifies if the matrix has been successfully created. If so, it populates it with numbers using the `fill()` function, then prints its content on the screen and frees the matrix with the help of the `release()` function. If the `create()` function failed to create the matrix, none of these operations would be performed.
The program can be improved by modifying the `create()` functio in such a way, that it checks if each of the integer arrays has been properly created.

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. These structures will be presented soon.

How To Read Complicated C Declarations? [2]

Looking at the examples presented in the lecture it is easy to discover that the variables in the C language may have 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.

[2]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 too 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

```c
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 an unspecified number of arguments and return a value of the int type.

Questions

?

The End

Thank You For Your Attention!