Fundamentals of Programming 1

Multidimensional Arrays

Arkadiusz Chrobot

Department of Computer Science

January 3, 2021

Outline

Multidimensional Arrays

A multidimensional array is an array that requires more than one index for locating an element. Most of the contemporary programming languages allow for creating and using such arrays. In the C language, declaration of the multidimensional array is similar to the declaration of a linear array, but it requires additional pairs of brackets. In general, there have to be as many pairs of brackets as the array has dimensions. Each of the pairs should contain the number of elements in the given dimension. The two-dimensional arrays (arrays of the arrays) are usually used for modeling matrices. The three dimensional arrays (arrays of the arrays of the arrays) can be imagined as having a cubic or cuboid shape. Arrays of a higher number of dimensions may also be applied.

Multidimensional Arrays

Initialization of Multidimensional Arrays

The multidimensional arrays may be created as already initialized variables. In that case the number of elements for the first dimension may be omitted (the first pair of brackets may be left empty). Nevertheless it must be defined for the rest of the dimensions. The initial values for the multidimensional array are best described as an array of other arrays. In the simplest case of a two dimensional array, the initial value is an array of linear arrays, just like in the following example:

```
int first_matrix[2][3] = {{1,2,3},{4,5,6}};
int second_matrix[][2] = {{1,2},{3,4}};
```

In the first line the two dimensional array is initialized with the array of two elements which are, in turn, arrays of three elements of the `int` type. In the second line the two dimensional array is initialized with the array of two elements which are arrays of two elements of the `int` type. Please observe, that in the latter case the number of elements for the first dimension is not given.

Indexing of Elements

To access a single element in a multidimensional array it is necessary to provide all of its indices. The element has as many indices as the array has dimensions. Each of the indices should be embraced by a separate pair of brackets. The values of the indices have to be natural numbers, regardless if the indices are expressed as expressions, variables, constants or literals. None of the indices may be greater than or equal to the number of elements in a given dimension. The examples show expressions that reference a single element in a two, three and four dimensional array of elements of the `double` type.

```
double value = matrix[1][0];
value = cube[3][5][7];
value = hypercube[1][3][2][3];
```

Passing by Parameters

The multidimensional array may be passed to a function by a parameter. The parameter may be declared in the parameter list of the function in the same way as the array. When invoking the function the parameter is substituted by an argument which in this case is the array. It is also permissible to skip the number of elements of the first dimension in the declaration of the parameter. For the rest of the dimensions the number has to be defined. The first dimension in the parameter declaration can also be replaced by a pointer. The next three slides show functions that have the described parameters used for passing a two dimensional array.

Passing by Parameters

The First Way

```
void print_matrix(int matrix[ROWS][COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++) {
        for(j=0;j<COLUMNS;j++)
            printf("%4d",matrix[i][j]);
        puts("");
    }

}
```

Passing by Parameters

The First Way — a Comment

The function, which source code is presented in the previous slide, prints the values from a two dimensional array of elements of the `int` type on the screen. The number of elements in each of the dimensions is defined by the `ROWS` and `COLUMNS` constants, what makes changing them easy. It suffices to modify the values of the constants at the place where they are defined. Please notice, that accessing each element of the array requires using two loops, one nested into another. Please also observe the invocation of the `puts()` function. In this example the two dimensional array models a matrix. The aforementioned calling of the `puts()` function serves the purpose of moving the cursor at the beginning of the next line on the screen so the next row of the matrix can be displayed there. This causes the matrix to look on the screen similarly to its mathematical notation. The `printf()` function reserves, according to the formatting string, four places on the screen for each value of the array.

Passing by Parameters

The Second Way

```
void fill_matrix(int matrix[][COLUMNS])
{
    int i,j;
    srand(time(0));
    for(i=0;i<ROWS;i++)
        for(j=0;j<COLUMNS;j++)
            matrix[i][j] = rand()%10;
}
```

Passing by Parameters

The Second Way — a Comment

The function presented in the previous slide fills the two dimensional array (a matrix) with randomly chosen natural numbers ranging from 0 to 9. In the declaration of the function's parameter the number of elements for the first dimension of matrix is omitted. However, the `ROWS` constant can still be used in the function's body, as it is demonstrated in the external loop header.

Passing by Parameters

The Third Way

```
void print_matrix(int (*matrix)[COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++) {
        for(j=0;j<COLUMNS;j++)
            printf("%4d",matrix[i][j]);
        puts("");
    }
}
```

Passing By Parameters

The Third Way — a Comment

In the definition of the function from the previous slide, a pointer is applied as a parameter for passing a two dimensional array. The function displays on the screen values from the array, just like the first of the functions described in previous slides. Please notice the parentheses used in the parameter declaration. They are necessary to define the parameter as a pointer to an array and not an array of pointers.

Arrays of Strings

Arrays of strings are a special case of two dimensional arrays. It is possible to access a single character in the array by providing two indices: one for the string and one for the character. However, it is also possible to provide only one index for the string. Those arrays are created the same way as regular two dimensional arrays. Only the type of their elements is defined as `char`. The arrays may also be initialized. The example shows the declaration of such an array.

```
char cities[][LENGTH] = {"Brussels","Amsterdam","Antwerp",
                         "Cracow","Vienna","Warsaw"};
```

The `LENGTH` constant defines the number of elements for storing characters. The strings, which are the initial values, do not have to be embraced by curly braces, they may be embraced only by quotation marks. Functions that implement operations on the array are presented in next slides. Those operations are sorting and printing.

Arrays of Strings

Printing the Array of String on the Screen

```
void print_cities(char cities[][LENGTH])
{
    int i;
    for(i=0; i<NUMBER; i++)
        printf("%s ",cities[i]);
    puts("");
}
```

Arrays of Strings

Printing the Array of Strings on the Screen — a Comment

The function from the previous slide prints all strings from the array on the screen. The array is passed by the parameter, in which the number of elements for the first dimension is omitted. The number of strings in the array is given by the `NUMBER` constant, which is used in the `for` loop. To print the values from the array only one loop is needed. The individual strings are displayed on the screen by the `printf()` function with the use of the `"%s"` formatting string.

Arrays of Strings

The swap() Function

```
void swap(char first[], char second[])
{
    char tmp[LENGTH];

    strncpy(tmp,first,LENGTH-1);
    strncpy(first,second,LENGTH-1);
    strncpy(second,tmp,LENGTH-1);
}
```

Arrays of Strings

The `swap()` Function — a Comment

The `swap()` function for the array of strings differs from the same functions for the primitive data types like the `int`. As the parameters are used the arrays of characters. They are substituted by the elements of the array of strings when the function is invoked. The strings stored in the elements are switching their places. To the end the `tmp`[1] variable is created. It is used for storing the first of the strings temporarily. The strings are copied with the use of the `strncpy()` function. The `LENGTH` constant is used in the expression that evaluates to the maximum number of characters that the function can copy.

[1] It is an abbreviation of the "temporary" word.

Arrays of Strings

Sorting of the Array of Strings

```
void sort_cities(char (*cities)[LENGTH])
{
    int i,j;

    for(i=0; i<NUMBER-1; i++) {
        int min = i;
        for(j=i+1; j<NUMBER; j++)
            if(strncmp(cities[min],cities[j],LENGTH-1)>0)
                min = j;
        if(min!=i)
            swap(cities[min],cities[i]);
    }
}
```

Arrays of Strings

Sorting of the Arrays of Strings — a Comment

The function from the previous slide is a modification of the function for sorting a linear array with the use of the selection sort algorithm. In the header, beside the name, the parameter declaration has been changed, so it can pass an array of strings. Also the name of the constant that defines the number of elements in the array has been changed. More importantly, the `strncmp()` is applied in the conditional statement for comparing the string indicated by the `min` index with the one indicated by the `j` index. If the latter is less than the other the function returns a value greater than zero.

Arrays of Strings

Program Arguments

The arrays of strings have a special meaning for the `main()` function. Its parameter list do not have to be empty. It can contain two parameters, which are usually (the names can be changed) called `argc` and `argv`. The first one is of `int` type and the second one is an array of pointers to strings. Both of them are used for passing to the `main()` function the *program arguments*. As an example of such arguments the options of Unix shell commands may be given. Each of the argument is interpreted by the program as a string. If there is more than one argument then they are separated by whitespaces. Those strings (arguments) are stored in the `argv` array and the number of them is stored in the `argc` parameter. It is vital to know, that the value of the `argc` parameter is always at least one. This is because the `argv` array stores in its first element the fully qualified name of the program, i.e. the name of the program's executable file in the form of its full path. The next slide contains a program that prints in separate lines on the screen its arguments.

Arrays of Strings

Program Arguments

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for(i=0;i<argc;i++)
        printf("%s\n",argv[i]);
    puts("");
    return 0;
}
```

Matrix Operations

The two dimensional arrays are used for representing matrices. In that case the first dimension in the matrix declaration defines the number of rows and the second one the number of columns and thus also the number of items (elements) in each of the rows. Therefore the 2×3 matrix of integer elements could be declared as follows:

```
int matrix[2][3];
```

If the numbers of elements in both dimensions are equal than the matrix is a square matrix. When referencing a single element of the matrix, the first index indicates the row where the element is located and the second one the column. There are many operations defined for the matrices, like addition, subtraction, multiplication, transposition, inversion and calculation of a determinant. The next slides contain functions, with descriptions, that implement the first four of them.

Addition

Two matrices can be added if and only if they have the same numbers of rows and columns. The resulting matrix has the same dimensions as the arguments have. The adding of matrices simply boils down to adding the values of the corresponding elements of both arguments. The operation is illustrated in the next slide. Three items were highlighted in the slide. The items with the green and blue background are those that are added. The resulting item has the red background.

Matrix Operations

Addition

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix}$$

Matrix Operations

Addition — an Implementation

```
void add_matrices(int (*argument_1)[COLUMNS],
                  int (*argument_2)[COLUMNS],
                  int result[ROWS][COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++)
        for(j=0;j<COLUMNS;j++)
            result[i][j] =
                argument_1[i][j] + argument_2[i][j];
}
```

Matrix Operations

Addition — a Comment on the Implementation

Matrices can not be, at least directly, returned by a function. Therefore the `add_matrices()` function has three parameters. The first two of them are used for passing the matrices to the function and the third one is used for passing the resulting matrix outside the function. Adding the matrices requires two loops, one nested in the other. The first one indexes rows in the three matrices and the second one indexes the elements in each row.

Matrix Operations

Subtraction

The subtraction of matrices is similar to addition. The next slide illustrates such an operation.

Matrix Operations

Subtraction

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Matrix Operations

Subtraction — an Implementation

```
void substract_matrices(int (*argument_1)[COLUMNS],
                        int (*argument_2)[COLUMNS],
                        int result[ROWS][COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++)
        for(j=0;j<COLUMNS;j++)
            result[i][j] =
                argument_1[i][j] - argument_2[i][j];
}
```

Subtraction — a Comment on the Implementation

The function that subtracts matrices is defined similarly to the function that implements the addition. The only difference, beside the name, is the subtraction operator used in place of the addition operator.

Matrix Operations

Multiplication

Multiplication of matrices is more complicated operation than the two previously described. In contrast to the numbers multiplication the matrix multiplication is not commutative, i.e. the order of the arguments of multiplication is important. Moreover, the operation is only possible if the first argument has as many columns as the second argument has rows. Calculating the value of a single element of the resulting matrix consists in adding the products of all items in a specific row of the first argument and the corresponding column of the second argument. The next slide contains an example. The value of the item with the red background is calculated by multiplying the row with the green background from the first matrix and the column with the blue background from the second matrix, i.e. by evaluating the following expression: $1 * 1 + 2 * 3 + 3 * 5 = 22$.

Matrix Operations

Multiplication

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}$$

Matrix Operations

Multiplication — Implementation

```
void multiply_matrices(int argument_1[2][3],
                       int argument_2[3][2],
                       int result[2][2])
{
    int i,j,k;
    for(i=0;i<2;i++)
        for(j=0;j<2;j++)
            for(k=0;k<3;k++)
                result[i][j] +=
                    argument_1[i][k]*argument_2[k][j];
}
```

Multiplication — a Comment to the Implementation

The parameters of the function play the same role as in the previously defined functions for addition and subtraction. The function uses the simplest matrix multiplication algorithm. It requires using three loops. The first one indicates a row in the first argument, the second one indicates the column in the second argument and the third one indexes consecutive elements of this row and this column. Also the loop counters of the two first loops indicate the element, in the resulting matrix, in which the result is stored. The element is used during the calculation for storing the partial sums of the items products, so its initial value has to be zero. Therefore the third argument of the function has to be a zero matrix. It is enough to declare the matrix as a global variable.

Matrix Operations

Matrix Multiplication Effectiveness

The order of the loops in the function can be changed. It occurs that it has an impact on the performance of the function. Let's denote the loops by the names of their loop counters. The most effective order of the loops is (ikj), but it also depends on the configuration of the computer that executes the function. There are more efficient algorithms of the matrix multiplication, but they are more complicated, harder to implement correctly and the increased performance is noticeable only for very big matrices.

Matrix Operations

Transposition

In the simplest terms matrix transposition involves swapping the columns and rows, so a $\mathbb{A}$ matrix of 2×3 dimensions becomes a matrix $\mathbb{A}^T$ of the 3×2 dimensions. The result of matrix transposition is called a *transpose*. The next slide shows an example of such an operation.

Matrix Operations

Transposition

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Matrix Operations

Transposition — Implementation

```
void transpose_matrix(int argument[ROWS][COLUMNS],
                      int result[COLUMNS][ROWS])
{
    int i,j;
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            result[j][i]=argument[i][j];
}
```

Transposition — a Comment to the Implementation

The transposition of a rectangular matrix requires using an additional matrix for storing the transpose. The matrix to be transposed is passed to the function by the first argument, the result is passed outside by the second parameter. Please observe the way the loop counters are used as indices in both matrices. The order of the indices is reversed in the resulting matrix.

Matrix Operations

Transposition of a Square Matrix —Implementation

```
void transpose_square_matrix(int matrix [3][3])
{
    int i,j;
    for(i=0; i<3; i++)
        for(j=0; j<i; j++) {
                int tmp;
                tmp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = tmp;
        }
}
```

Matrix Operations

Transposition of a Square Matrix — a Comment on the Implementation

If the matrix to be transposed is a squared one, the operation can be implemented with the use of only one array of arrays. The square matrix consists of two triangular matrices located on the both sides of its main diagonal. The algorithm exchanges the values of the elements in both triangular matrices. Please notice the header of the inner loop. The upper limit of its loop counter values (the `j` variable) is set by the current value of the loop counter of the external loop (the `i` variable). The body of the inner loop contains statements that swap the values of two of the matrix elements. The statements may be replaced by the invocation of the `swap()` function defined in the previous lectures.

Conway's Game of Life

The Game of Life is a cellular automaton (CA) that was discovered by a British mathematician John Conway. Contrary to its name, it has nothing to do with the entertainment, at least not in a common meaning of the word. The automaton is build from cells, arranged in an infinite square board, that change their states in each step of the automaton activity, according to some simple rules. States of all of the cells form a state of the game, which can evolve in a very complicated way. The automaton is a subject of research for mathematicians, physicists, computer scientists and biologists. From the computer science point of view the automaton is equivalent to the Turing machine. It means that the automaton is a computer capable of processing information. In the lecture other aspect of the Game of Life is discussed. The automaton can be simulated with the use of matrices.

Conway's Game of Life

Cell State Evolution Rules

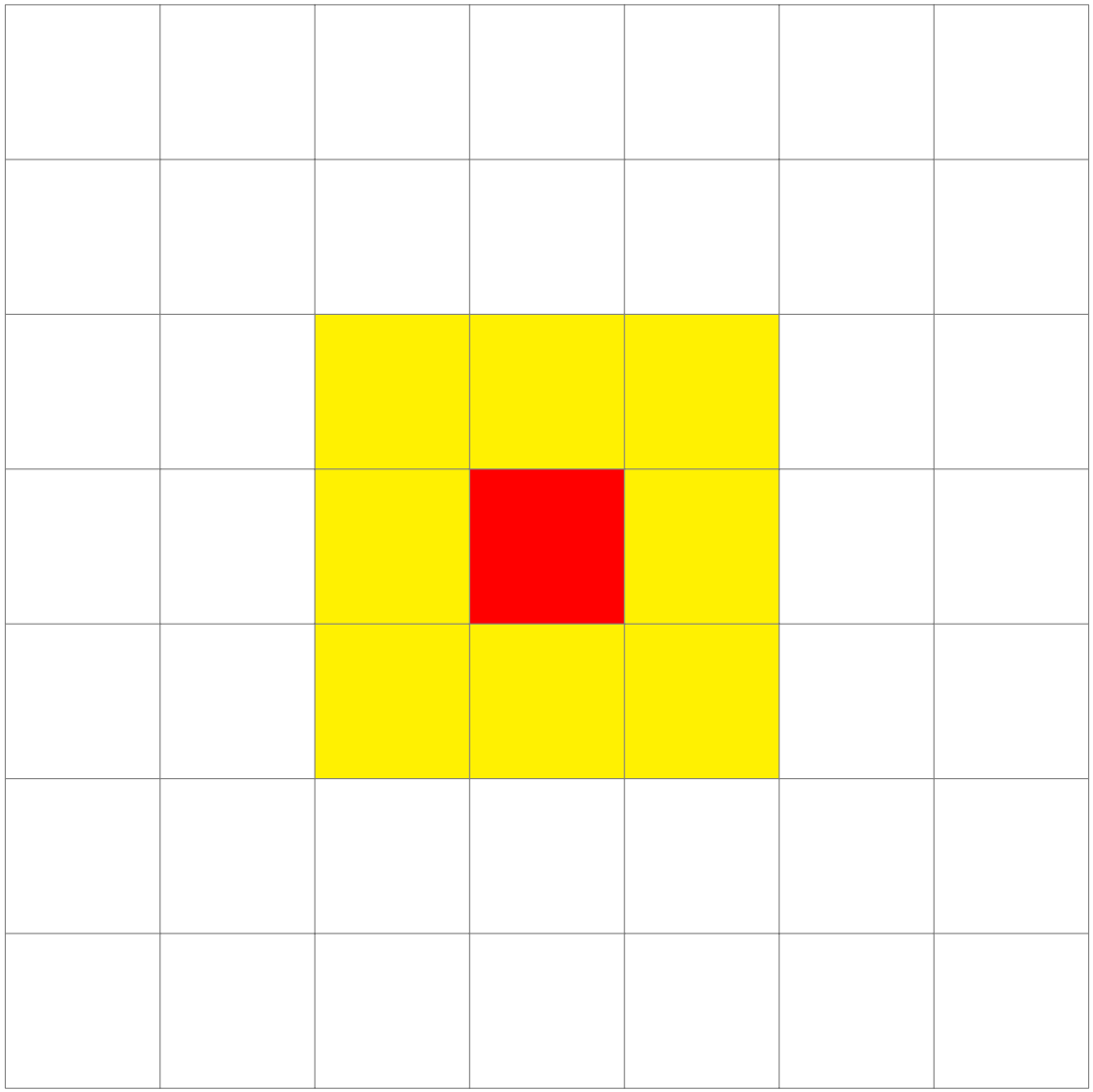
The cells of the automaton are represented by elements of a matrix. Each of the cells in a specific moment of the automaton activity may be in one of two states: dead or alive. The change in the state of the cell in next step of the automaton activity is depended on the states of its neighbours in the current step and undergoes the following rules:

1. each alive cell that has less than two alive neighbours dies,
2. each alive cell that has more than three alive neighbours dies,
3. each alive cell that has two or three alive neighbours stays alive,
4. each dead cell that has three alive neighbours becomes alive.

The next slide contains a picture that illustrates the concept of neighbourhood in the Game of Life.

Conway's Game of Life

Moore Neighbourhood



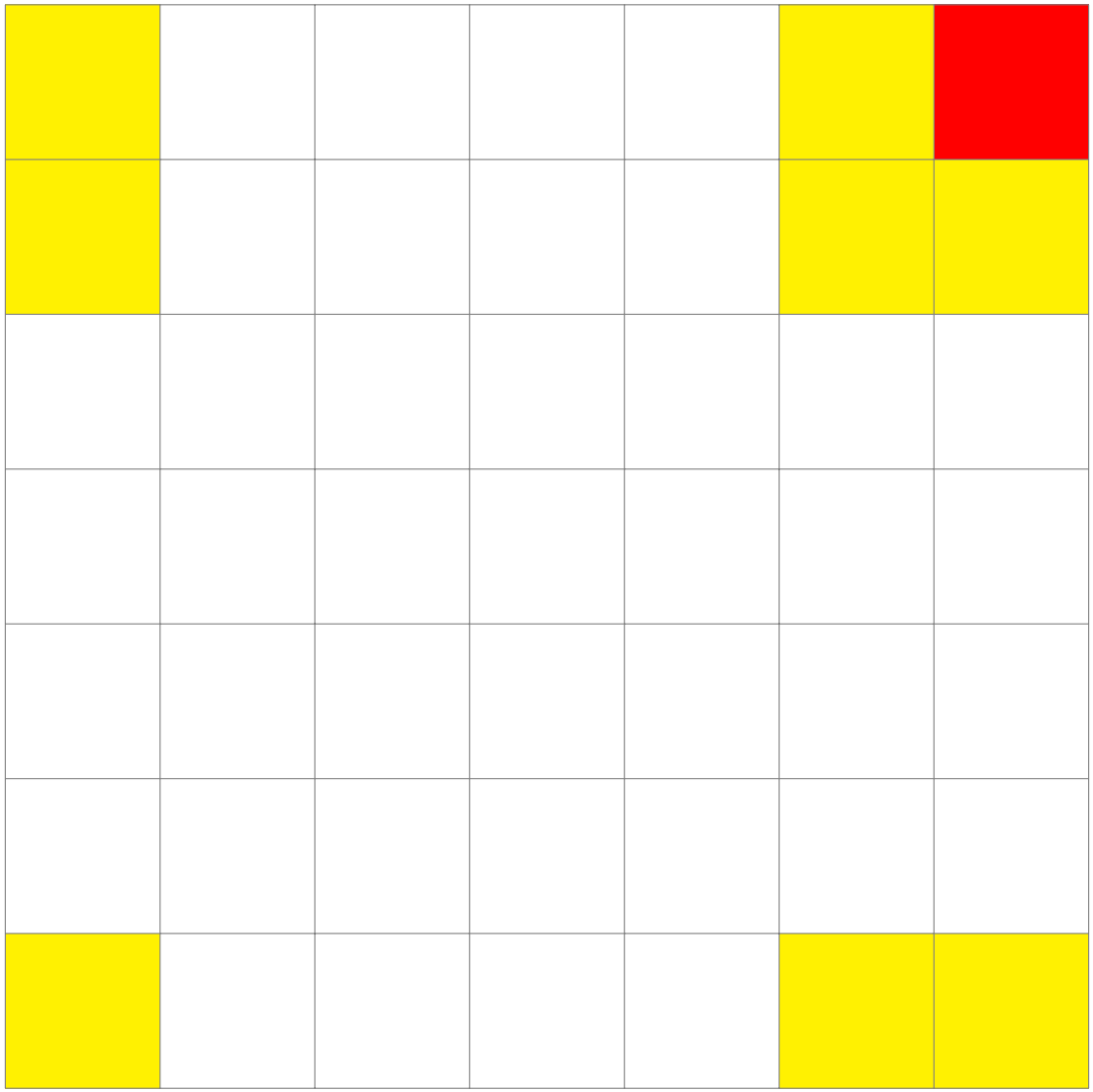
Conway's Game of Life

Coordinates of the Neighbours

The red-marked cell has eight yellow-marked neighbours, as it is shown in the picture. Calculating their coordinates (the row and the column) is not hard. They differ at most by -1 or $+1$ from the coordinates of the red-marked cell. For example, if the red-marked cell has coordinates of (x, y), then its top left neighbour has coordinates of $(x - 1, y - 1)$, assuming that the cell in top left corner of the board has coordinates of $(0, 0)$, the values of indices of rows grow in top-bottom direction and the values of indices of columns grow in left-right direction. There is however one more issue that should be taken into account. The issue is illustrated in the next slide.

Conway's Game of Life

More Neighbourhood



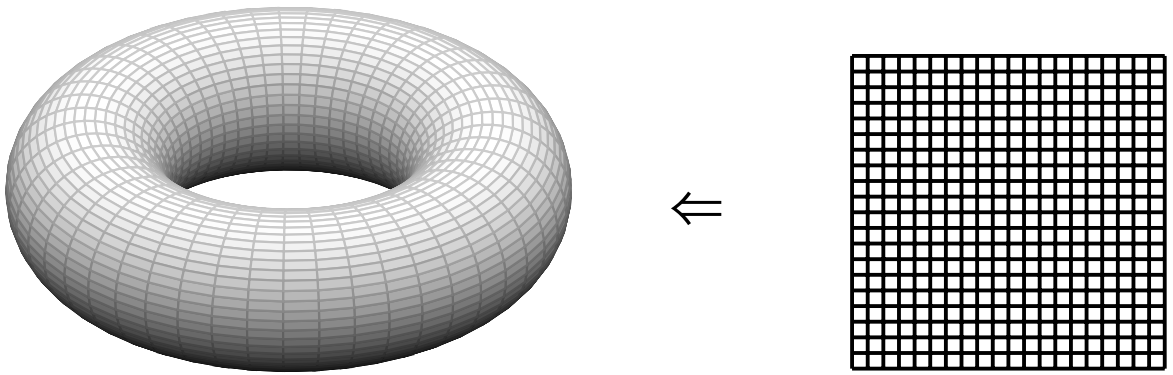
Conway's Game of Life

Coordinates of the Neighbours

The game's board should be infinite. It means that the elements on the edges of the board must also have their neighbours and exactly eight for each of them. This obstacle can be overcome by using the modular arithmetic, what is demonstrated in the source code of a program that simulates the Game of Life. Conceptually, the modular arithmetics converts the square board into a torus, as it is depicted in the next slide.

Conway's Game of Life

The Board



Conway's Game of Life

Implementation

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

#define SIZE 32

enum state {DEAD, ALIVE};

unsigned char board[SIZE][SIZE];
```

Conway's Game of Life

Implementation — a Comment

The beginning of the program from the previous slide contains, beside the preprocessor directives that include all necessary header files, a definition of a constant that defines the size of the game's board (32 elements in each dimension), a definition of an enumerated type and a declaration of a matrix. Variables of the enumerated type are not used in the program. Only elements of this type are used as constants that describe the state of a single cell. A dead cell has a value of 0 and the constant for this state is named `DEAD`. An alive cell has a value of 1, and the constant for the state is named `ALIVE`.

Conway's Game of Life

Implementation

```
unsigned int down(unsigned int y)
{
    return (y+1)%SIZE;
}

unsigned int up(unsigned int y)
{
    return (y+(SIZE-1))%SIZE;
}
```

Conway's Game of Life

Implementation — a Comment

The `up()` and `down()` functions are used for finding the vertical coordinate of neighbours of any cell on the board. The `down()` function uses the modulus operator to find the vertical coordinate of the neighbours of cells at "the very bottom" of the board. The `up()` function uses the same operator to find the vertical coordinate of the neighbours of cells at "the very top" of the board. The `down()` function increments the value of the vertical coordinate of a cell in the range from `0` to `SIZE-1`. The `up()` function decrements the same value in the same range.

Conway's Game of Life

Implementation

```
unsigned int right(unsigned int x)
{
    return (x+1)%SIZE;
}

unsigned int left(unsigned int x)
{
    return (x+(SIZE-1))%SIZE;
}
```

Conway's Game of Life

Implementation — a Comment

The horizontal coordinate of the neighbours of any cell may be calculated with the use of the same functions that calculate the vertical coordinate. However, the program is more legible if two separate functions are defined for calculating the horizontal coordinate. Those functions are similar to the ones that find the vertical coordinate.

Conway's Game of Life

Implementation

```
unsigned char count_alive_neighbours(
                            unsigned char board[SIZE][SIZE],
                            unsigned int i, unsigned int j)
{
    return board[i][down(j)]
           + board[i][up(j)]
           + board[left(i)][j]
           + board[right(i)][j]
           + board[right(i)][down(j)]
           + board[right(i)][up(j)]
           + board[left(i)][down(j)]
           + board[left(i)][up(j)];
}
```

Conway's Game of Life

Implementation — a Comment

The `count_alive_neighbours()` function counts the number of alive neighbours of a cell which coordinates are given by `i` and `j`. Since every alive cell has the value of `1` and every dead cell has a value of `0`, it is enough to add values of all neighbouring cells to get the number of the alive ones.

Conway's Game of Life

Implementation

```
void get_next_step(unsigned char board[SIZE][SIZE])
{
    static unsigned char swap[SIZE][SIZE];
    unsigned int i,j;

    for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++) {
            unsigned char state = board[i][j];
            unsigned char alive_neighbours = count_alive_neighbours(board,i,j);
            if(state == ALIVE && alive_neighbours < 2)
                swap[i][j] = DEAD;
            if(state == ALIVE && alive_neighbours > 3)
                swap[i][j] = DEAD;
            if(state == ALIVE && (alive_neighbours == 3 || alive_neighbours == 2))
                swap[i][j] = ALIVE;
            if(state == DEAD && alive_neighbours == 3)
                swap[i][j] = ALIVE;
        }
    memcpy(board,swap,SIZE*SIZE);
}
```

Conway's Game of Life

Implementation — a Comment

The `get_next_step()` function generates the state of the automaton in the next step. The new state is stored in the `swap` matrix. It is created according to the current state that is passed to the function in the form of the `board` matrix. Please observe that the `swap` matrix is a `static` local variable. It is the easiest way of initializing the matrix with zeros. In the two `for` loops the state of each cell is read and the number of its alive neighbours is calculated. Then the resulting state of the cell is computed according to the rules presented before and stored in the `swap` matrix. Finally, after the loops terminate, the content of the `swap` matrix is copied to the `board` matrix with the use of `memcpy()` function.

Conway's Game of Life

Implementation

```
void seed_board(unsigned char board[SIZE][SIZE])
{
    unsigned int i,j,k;
    srand(time(NULL));
    for(k = 0; k<8; k++) {
        i = rand()%SIZE;
        j = rand()%SIZE;
        board[i][j] = ALIVE;
        int choice = rand()%8;
        switch(choice) {
        case 0 :
            board[i][down(j)] = ALIVE;
        case 1 :
            board[i][up(j)] = ALIVE;
        case 2 :
            board[left(i)][j] = ALIVE;
        case 3 :
            board[right(i)][j] = ALIVE;
        case 4 :
            board[right(i)][down(j)] = ALIVE;
        case 5 :
            board[right(i)][up(j)] = ALIVE;
        case 6 :
            board[left(i)][down(j)] = ALIVE;
        case 7 :
            board[left(i)][up(j)] = ALIVE;
        }
    }
}
```

Conway's Game of Life

Implementation — a Comment

The `seed_board()` function initializes the board before the game starts. It chooses randomly the coordinates of eight cells which become alive and for each of them it initializes alive neighbours. The number of that neighbours is also randomly chosen by the function.

Conway's Game of Life

Implementation

```
void create_blinker(unsigned char board[SIZE][SIZE])
{
    board[SIZE/2-1][SIZE/2-1] = board[SIZE/2][SIZE/2-1]
    = board[SIZE/2+1][SIZE/2-1] = ALIVE;
}
```

Conway's Game of Life

Implementation — a Comment

The `create_blinker()` function also initializes the game's board, but this time deterministically. It creates in the middle of the board a pattern that belongs to the group of so called oscillators and its name is "blinker". The pattern consists of three cells. Two vertical and two horizontal cells alternately become dead or alive, so they look as if they are circulating.

Conway's Game of Life

Implementation

```
void create_ten_in_row(unsigned char board[SIZE][SIZE])
{
    memset((void *)&board[SIZE/2-1][SIZE/2-6],ALIVE,10);
}
```

Conway's Game of Life

Implementation — a Comment

The `create_ten_in_row()` function also initializes the board by placing in the middle of it a pattern that consists of ten alive cells located one by one in a row. The pattern is called a "ten in a row" or a "crocodile". It turns out that the simple pattern evolves in a complicated meaner. To initialize the ten elements of the matrix with the `ALIVE` value the `memset()` function is used. It is possible thanks to the structure of the matrix – each of its elements has a size of one byte. The function assigns the value, passed as its second argument, to the ten subsequent cells, starting with the cell which coordinates are $(\texttt{SIZE}/2 - 1, \texttt{SIZE}/2 - 6)$.

Conway's Game of Life

Implementation

```
void print_board(unsigned char board[SIZE][SIZE])
{
    unsigned int i,j;

    for(i=0; i<SIZE; i++) {
        printf("\n");
        for(j=0; j<SIZE; j++)
            printf("%2d",board[i][j]);
    }
    printf("\n");
}
```

Conway's Game of Life

Implementation — a Comment

The `print_board()` function displays on the screen the current state of the game, in other words the content of the matrix that is passed as its argument. For the value of each cell the `printf()` function reserves two places on the screen. Please observe, that the `printf()` function is also used instead of the `puts()` function to move the cursor to the next line on the screen.

Conway's Game of Life

Implementation

```
int main(int argc, char **argv)
{
    if(argc==2) {
        if(!strcmp(argv[1],"blinker"))
            create_blinker(board);
        else if(!strcmp(argv[1],"ten_in_row"))
            create_ten_in_row(board);
        else
            seed_board(board);
    } else
        seed_board(board);

    do {
        print_board(board);
        get_next_step(board);
    } while(getchar()!='q');
    return 0;
}
```

Conway's Game of Life

Implementation — a Comment

In the beginning the `main()` function initializes the state of the game basing on whether the user passed an argument to the program and the value of the argument. If there is one argument, then the function checks whether its value is `blinker` or `ten_in_row`. According to this value it sets one of the patterns as an initial state of the board. If the program is run without an argument or the argument has an invalid value, the board is initialized (pseudo)randomly. After the initialization is finished the `do...while` loop is performed until the user presses the `q` or the `Enter` key. In the body of the loop the current state of the game is displayed on the screen with the use of the `print_board()` function and the next state is generated with the use of the `get_next_step()` function.

Conway's Game of Life

Implementation — a Comment

The described program displays the state of the game on screen as a series of zeros and ones. The overall effect is not spectacular, but a more fluent animation may be achieved by pressing and holding the `Enter` key. Printing the state of the game is improved in the future lectures.

The Game of Life, depending on the initial state may evolve into four deferent ways:

1. the board becomes empty — all the cells are dead,
2. the state of the game is stable — all the patterns on the board aren't evolving any more,
3. the state of the game is changing periodically,
4. the game evolves infinitely.

A series of interesting patterns may appear on the board during the game.

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!