Structuring data

Week 4

Arrays of numbers

This far we have been considering arrays of characters, that is, strings.

But we can store to an array elements of any type.

```
dataType arrayName[arraySize];
```

Declare an array, `mark`, of floating-point type, whose size is 5.

float mark[5];

It is possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

File: Lecture4 > `Example1.c`

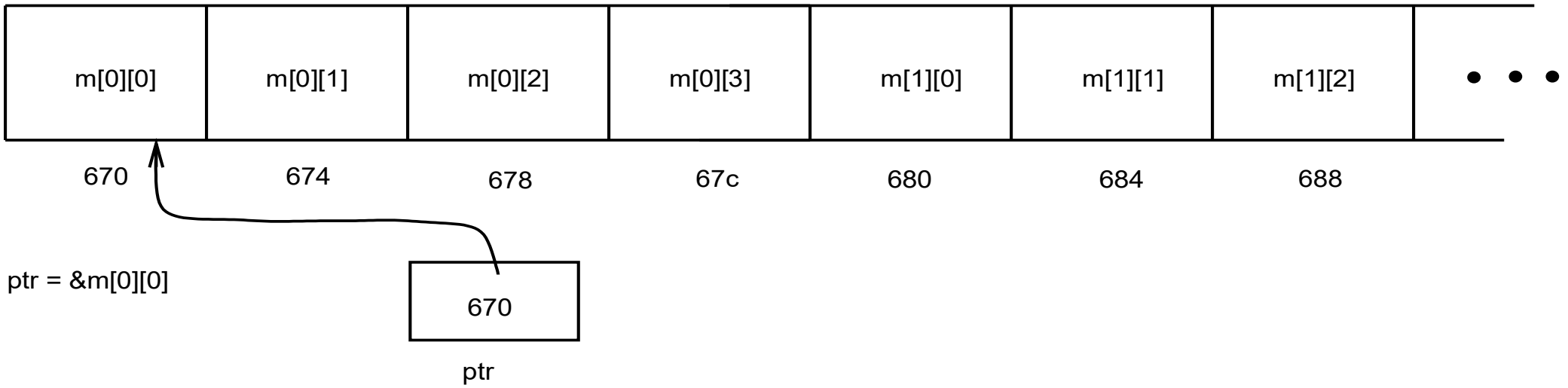
2-dimensional array = matrix

m[0][0]	m[0][1]	m[0][2]	m[0][3]
m[1][0]	m[1][1]	m[1][2]	m[1][3]
m[2][0]	m[2][1]	m[2][2]	m[2][3]

File: `Example2.c`

2-dimensional array = matrix

However the actual representation of this array in memory would be something like this:



Pointers

- Every variable is a memory location and every memory location has its **address**. This address can be accessed using ampersand (&) operator.

- A pointer is a variable whose value is the ***address*** of another variable, i.e., direct address of the memory location.

- Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

```
type *ptr_name;
```

Pointers

```
// Declarations
    int nr;
    int *p_nr;

    nr = 2;
    p_nr = &nr;
    printf("%d, %d \n", nr, *p_nr);
    return 0;
```

Pointer arithmetic

A **pointer** is an address, which is a numeric value.

We can perform arithmetic operations on a pointer just as you can on a numeric value.

There are four arithmetic operators that can be used on pointers:

```
++, --, +, -
```

Let `ptr` be an integer pointer which points to the address 1000.

Assuming 32-bit integers, after `ptr++`

`ptr`	will point to the location 1004 (4 bytes forward)
`*ptr`	the content of the location to what `ptr` is pointing

Records

Arrays allow to define type of variables that can hold several data items of the same kind. **Structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a **record**. Suppose you want to keep track of your ***books*** in a library. The following information relates to one book:

- Title
- Author
- Publisher
- Year

Records

```
struct Books {
    char  title[50];
    char  author[50];
    char  publisher[100];
    int   year;
};
```

To access members of a structure, we use the dot-operator (`.`)

File: `Example3.c`

typedef

The C language contains the `typedef` keyword to allow users to provide alternative names for user-defined (e.g struct) data types.

This keyword ***adds a new name*** for some existing data type but does not create a new type.

The most common use is to avoid the need to type the word “struct” all over again.

```
typedef struct Book BOOK;

BOOK Book2 = {"Python Crash Course",
              "Eric Matthes", "No Starch Press", 2015};
```

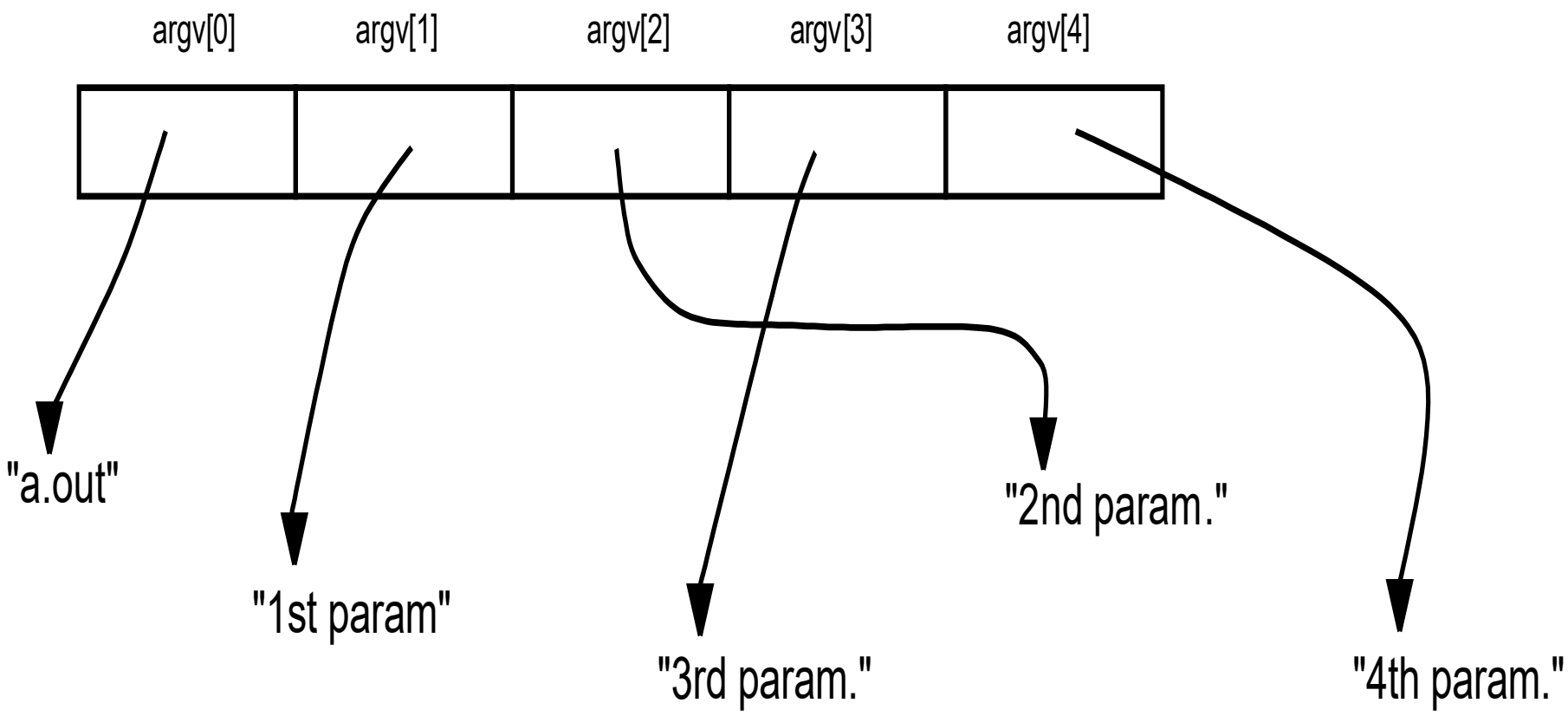
File: `Example4.c`

Command line arguments

- It is possible to pass some values from the command line to your C programs when they are executed.
- These values are called **command line arguments**.
- You can pass any number of arguments and you do not need to define variables for them.
- The command line arguments are handled by `main()` function arguments.
- Typically `argc` refers to the ***number*** of arguments and `argv[]` is an array which points to each argument (string) passed to the program
- Recall that each string in C holds the address of the first element of the array, i.e., it points at the starting memory address.
- It should be noted that `argv[0]` holds the name of the program itself

File: `Example5.c; Example6.c`

Command line arguments



Strings to numbers

The `atoi` function converts a string to integer.

This makes it possible for you to give integers as command line arguments.

File: `Example7.c;`

The `atof` function converts a string to a floating-point number (double).

File: `Example8.c;`

Recursive definitions / Factorial

$0! = 1$ and $n! = (n-1)! \cdot n$ for $n \geq 1$

$5! = 4! \cdot 5$ = **120**

$4! = 3! \cdot 4$ = 24 ↑

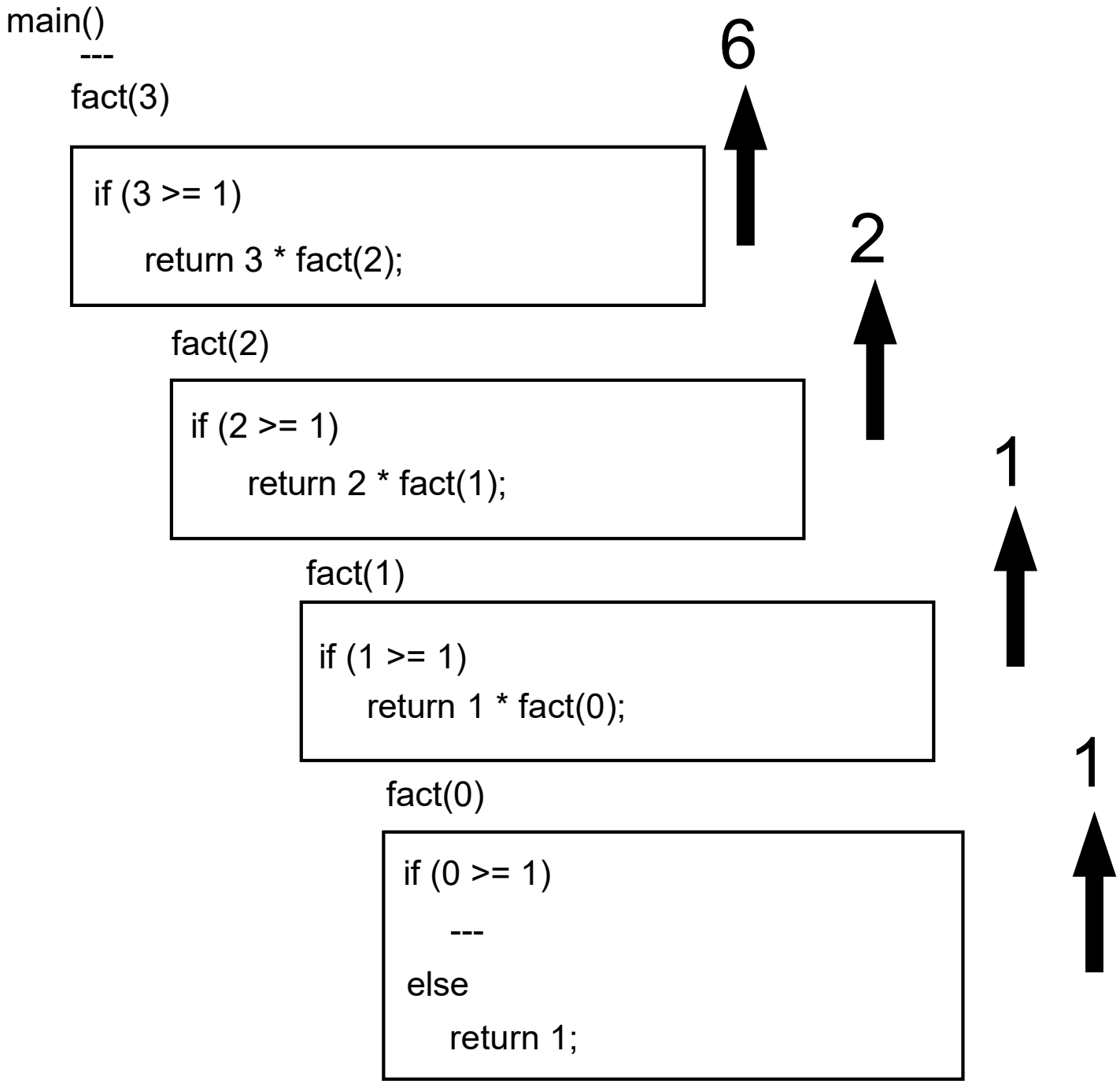
$3! = 2! \cdot 3$ = 6 ↑

$2! = 1! \cdot 2$ = 2 ↑

$1! = 0! \cdot 1$ = 1 ↑

$0! = 1$ ↑

File: `Example9.c`


main()
---
fact(3)
if (3 >= 1)
return 3 * fact(2);
6
fact(2)
if (2 >= 1)
return 2 * fact(1);
2
fact(1)
if (1 >= 1)
return 1 * fact(0);
1
fact(0)
if (0 >= 1)
---
else
return 1;
1

Array of Pointers

As we created an array of integers in the examples above, we can create an array of pointers — an array that stores memory addresses

File: `Example10.c`

Records and binary files

File: `Example11.c`