

JTSK-320111

Programming in C I

C-Lab I

Lecture 5 & 6

Dr. Kinga Lipskoch

Fall 2017

This Week's Agenda

- ▶ Pointers and arrays
- ▶ Dynamic memory allocation
- ▶ Multi-dimensional arrays
- ▶ Recursive functions
- ▶ Dealing with larger projects
- ▶ Handling files
- ▶ Revision

Passing Arrays to Functions

- ▶ An array does not store its size
- ▶ This has to be provided as a parameter, or by making assumptions on the contents of the array (like for strings)
- ▶ When an array is passed to a function, a copy of the address of the first element is given
- ▶ Modifications to the elements are seen outside
- ▶ Modifications to the array are not seen outside
- ▶ Can you explain why?

Passing Arrays to Functions: Example

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void strange_function(int v[], int dim) {
4     int i;
5     for (i = 0; i < dim; i++)
6         v[i] = 287;
7     // v = (int *) malloc(sizeof(int) * 1000);
8 }
9 int main() {
10    int array[] = {1, 2, 9, 16};
11    int *p = &array[0];
12    strange_function(array, 4);
13    printf("%d %x %x\n", array[0], p, array);
14 }
```

Dynamic Memory Allocation

- ▶ What if we do not know the dimension of the array while coding?
- ▶ Dynamic memory allocation allows you to solve this problem
 - ▶ And many others
 - ▶ But can also cause a lot of troubles if you misuse it

Pointers and Arrays

There is a strong relation between pointers and arrays

- ▶ Indeed an array is nothing but a pointer to the first element in the sequence
- ▶ We are looking at this in detail

Specifying the Dimension on the Fly

To specify the dimension on the fly you can use the `malloc()` function defined in the header file `stdlib.h`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int *dyn_array, how_many, i;
5     printf("How many elements? ");
6     scanf("%d", &how_many);
7     dyn_array =
8         (int*) malloc(sizeof(int) * how_many);
9     if (dyn_array == NULL)
10        exit(1);
11    for (i = 0 ; i < how_many; i++) {
12        printf("\nInput number %d:", i);
13        scanf("%d", &dyn_array[i]);
14    } return 0;
15 }
```

The malloc() Function (1)

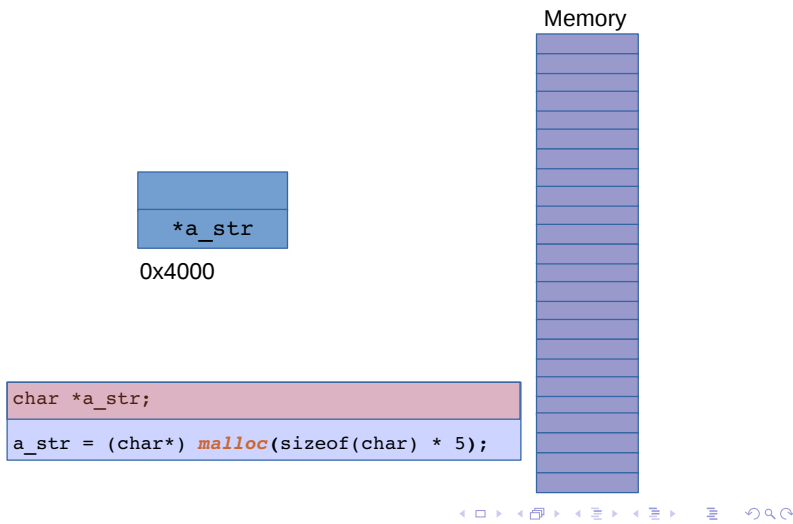
- ▶ `void * malloc(unsigned int);`
- ▶ `malloc` reserves a chunk of memory
- ▶ The parameter specifies how many bytes are requested
- ▶ `malloc` returns a pointer to the first byte of such a sequence
- ▶ The returned pointer must be forced (cast) to the required type

The malloc() Function (2)

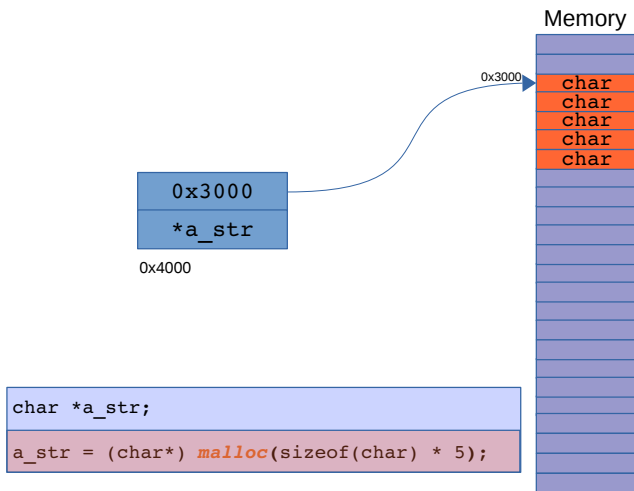
```
1 pointer    = (cast) malloc(number of bytes);
2
3
4 char* a_str;
5 a_str = (char*) malloc(sizeof(char) * how_many);
```

- ▶ malloc returns a `void *` pointer (i.e., a generic pointer) and this is assigned to a non `void *` pointer
- ▶ If you omit the casting you will get a warning concerning a possible incorrect assignment

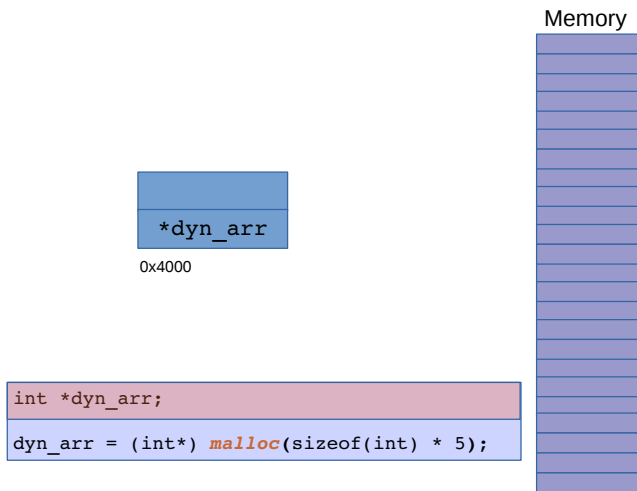
Dynamically Allocating Space for an Array of `char`



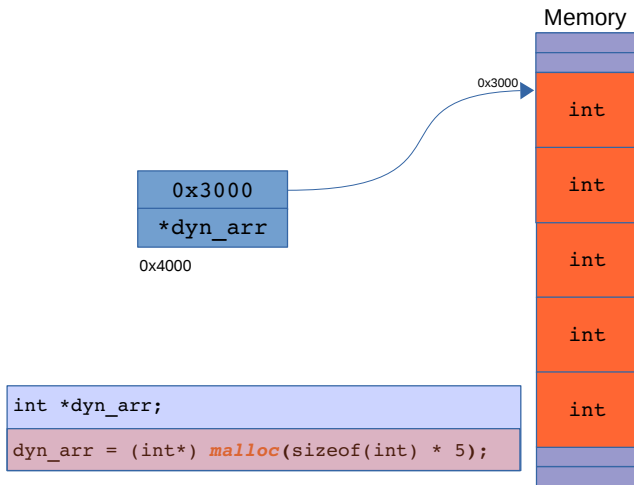
Dynamically Allocating Space for an Array of `char`



Dynamically Allocating Space for an Array of `int`



Dynamically Allocating Space for an Array of `int`



malloc() and free()

- ▶ All the memory you reserve via `malloc`, must be released by using the `free` function
- ▶ If you keep reserving memory without freeing, you will run out of memory

```
1  float *ptr;  
2  int  number;  
3  ...  
4  ptr = (float*) malloc(sizeof(float) *  
        number);  
5  ...  
6  free(ptr);
```

Rules for `malloc()` and `free()`

- ▶ The following points are up to you (the compiler does not perform any control)
 1. Always check if `malloc` returned a valid pointer (i.e., not `NULL`)
 2. Free allocated memory just once
 3. Free only dynamically allocated memory
- ▶ Not following these rules will cause endless troubles
- ▶ `sizeof()` is compile time operator, it does not work on allocated memory

Review: Pointers, Arrays, Values

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int length[2] = {7, 9};
4 int *ptr1, *ptr2; int n1, n2;
5 int main() {
6     ptr1 = &length[0];
7     // &length[0] is pointer to first elem
8     ptr2 = length;
9     // length is pointer to first elem therefore
10    // same as above
11    n1 = length[0];
12    // length[0] is value
13    n2 = *ptr2;
14    // *ptr2 is value therefore same as above
15    printf("ptr1: %p, ptr2: %p\n", ptr1, ptr2);
16    printf("n1: %d, n2: %d\n", n1, n2);
17    return 0;
18 }
```



Multi-dimensional Arrays

- ▶ It is possible to define multi-dimensional arrays
 - ▶ Mostly used are bidimensional arrays, i.e., tables or matrices
- ▶ As for arrays, to access an element it is necessary to provide an index for each dimension
 - ▶ Think of matrices in mathematics

Multi-dimensional Arrays in C

- ▶ It is necessary to specify the size of each dimension
 - ▶ Dimensions must be constants
 - ▶ In each dimension the first element is at position 0

```
1 int matrix[10][20];    /* 10 rows, 20 cols */
2 float cube[5][5][5];  /* 125 elements */
```

- ▶ Every index goes between brackets

```
1 matrix[0][0] = 5;
```

Multi-dimensional Arrays in C: Example

```
1 #include <stdio.h>
2 int main() {
3     int table[50][50];
4     int i, j, row, col;
5     scanf("%d", &row);
6     scanf("%d", &col);
7     for (i = 0; i < row; i++)
8         for (j = 0; j < col; j++)
9             table[i][j] = i * j;
10    for (i = 0; i < row; i++)
11    {
12        for (j = 0; j < col; j++)
13            printf("%d ", table[i][j]);
14        printf("\n");
15    }
16    return 0;
17 }
```

The main Function (1)

- ▶ Can return an `int` to the operating system
 - ▶ Program exit code (can be omitted)
 - ▶ print exit code in shell: `$> echo $?`
- ▶ Can accept two parameters:
 - ▶ An integer (usually called `argc`)
 - ▶ A vector of strings (usually called `argv`)
 - ▶ `argc` specifies how many strings contains `argv`

The main Function (2)

```
1 #include <stdio.h>
2 int main(int argc, char *argv[]) {
3     int i;
4     for (i = 1; i < argc; i++)
5         printf("%d %s\n", i, argv[i]);
6     return 0;
7 }
```

- ▶ Compile it and call the executable paramscounter
- ▶ Execute it as follows:
\$> ./paramscounter first what this
- ▶ It will print first, what and this, one word per line
- ▶ Note that argc is always greater or equal than one
- ▶ The first parameter is the program's name

The `const` Keyword

- ▶ The modifier `const` can be applied to variable declarations
- ▶ It states that the variable cannot be changed
 - ▶ i.e., it is not a variable but a constant
- ▶ When applied to arrays it means that the elements cannot be changed

const Examples

```
1  const double e = 2.71828182845905;
2  const char str[] = "Hello world";
3  e = 3;           /* error */
4  str[0] = 'h';   /* error */
```

- ▶ You can also use `#define` of the preprocessor
- ▶ But defines do not have type checking, while constants do

More `const` Examples

- ▶ `const char *text = "Hello";`
 - ▶ Does not mean that the variable `text` is constant
 - ▶ The data pointed to by `text` is a constant
 - ▶ While the data cannot be changed, the pointer can be changed
- ▶ `char *const name = "Test";`
 - ▶ `name` is a constant pointer
 - ▶ While the pointer is constant, the data the pointer points to may be changed
- ▶ `const char *const title = "Title";`
 - ▶ Neither the pointer nor the data may be changed

Recursive Functions (1)

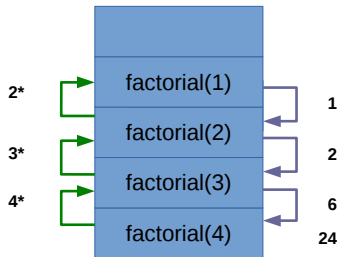
- ▶ Can a function call other functions?
 - ▶ Yes, indeed function calls appear only inside other functions (and everything starts with the execution of `main`)
- ▶ Can a function call itself?
 - ▶ Yes, but in this case special care should be taken
- ▶ A function which calls itself is called a **recursive function**
- ▶ Function *A* calls function *A*
- ▶ At a certain point function *B* calls *A*
 - ▶ *A* calls *A* then *A* calls *A* then *A* calls *A* ...
- ▶ When coding recursive functions attention should be paid to avoid endless recursive calls

Recursive Functions (2)

- ▶ Recursion theory can be studied for a longer time: here we will just scratch its surface from a basic coding standpoint
- ▶ Every recursive function must contain some code which allows it to terminate without entering the recursive step
 - ▶ Usually called **inductive base** or **base case**
- ▶ When recursion is executed, the new call should be driven "towards the inductive case"

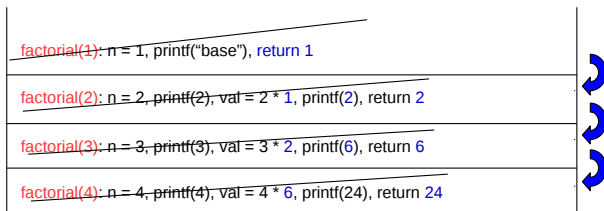
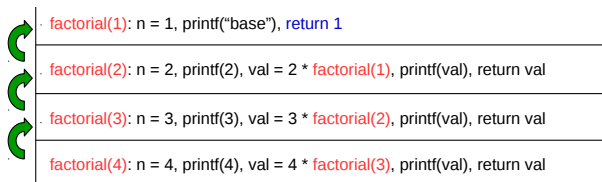
Stack of Calls: Example (1)

```
1 int factorial(int n) {  
2   if ((n == 0) || (n == 1))  
3     return 1;  
4   else  
5     return n * factorial(n - 1);  
6 }
```



Stack of Calls: Example (2)

From the main: call `factorial(4)`



Tracing the Stack of Calls

```
1 int factorial(int n) {
2     int val;
3     if ((n == 0) || (n == 1)) {
4         printf("base\n");
5         return 1;
6     } else {
7         printf("called with par = %d\n", n);
8         val = n * factorial(n - 1);
9         printf("returning %d\n", val);
10        return val;
11    }
12 }
13 int main() {
14     printf("%d\n", factorial(4));
15     return 0;
16 }
```

One More Example: Fibonacci Numbers

$$F(N) = \begin{cases} 1, & N \leq 1 \\ F(N-1) + F(N-2), & N > 1 \end{cases}$$

```
1 int fibonacci(int n) {
2     if ((n == 0) || (n == 1))
3         return 1;
4     else
5         return fibonacci(n-1) + fibonacci(n-2);
6 }
```

Dealing with Big Projects

- ▶ Functions are a first step to break big programs in small logical units
- ▶ A further step consists in breaking the source into many files
 - ▶ Smaller files are easy to handle
 - ▶ Objects sharing a context can be put together and easily reused
- ▶ C allows to put together separately compiled files to have one executable

Declarations and Definitions

- ▶ **Declaration:** introduces an object. After declaration the object can be used
 - ▶ Example: functions' prototypes
- ▶ **Definition:** specifies the structure of an object
 - ▶ Example: function definition
- ▶ Declarations can appear many times, definitions just once

Building from Multiple Sources

- ▶ C compilers can compile multiple sources files into one executable
- ▶ For every declaration there must be one definition in one of the compiled files
 - ▶ Indeed also libraries play a role
 - ▶ This control is performed by the linker
- ▶ `gcc -o name file1.c file2.c file3.c`

Libraries

- ▶ Libraries are collection of compiled definitions
- ▶ You include header files to get the declarations of objects in libraries
- ▶ At linking time libraries are searched for unresolved declarations
- ▶ Some libraries are included by `gcc` even if you do not specifically ask for them

Linking Math Functions: Example

```
1 #include <math.h>
2 #include <stdio.h>
3
4 int main() {
5     double n;
6     double sn;
7
8     scanf("%lf\n", &n); /* double needs %lf */
9     sn = sqrt(n);
10    /* conversion from double to float ok */
11    printf("Square root of %f is %f\n", n, sn);
12 }
13
14 gcc -lm -o compute compute.c
```

Compilers, Linkers and More

- ▶ Different compilers differ in many details
 - ▶ Libraries names, ways to link against them, types of linking
- ▶ Check your documentation
- ▶ But preprocessing, compilation and linking are common steps

File Handling in C

- ▶ Input and output can come from/go into files
- ▶ C treats files as streams of data
- ▶ A stream is a sequence of bytes (either incoming or outgoing)
- ▶ The language does not provide basic constructs for file handling, but rather the standard library does

Files, C and UNIX

- ▶ The file view of C is influenced by UNIX
- ▶ UNIX sees everything as a file
- ▶ You have already used two files/streams
 - ▶ `stdin` (standard input): associated with the keyboard
 - ▶ `stdout` (standard output): associated with the screen
 - ▶ These files are always tied to your program by the operating system

Working with Files

- ▶ The paradigm is the following:
 - ▶ open the file
 - ▶ read/write from/into file
 - ▶ close the file
- ▶ In C the information concerning a file are stored in a FILE structure (i.e., `struct`) defined in `stdio.h`

File Modes

Streams can be handled in two modes: (only important for MS Windows)

- ▶ Text streams: sequence of characters logically organized in lines. Lines are terminated by a newline ('\n')
 - ▶ Sometimes pre/post processed
 - ▶ Example: text files
- ▶ Binary streams: sequence of raw bytes
 - ▶ Examples: images, mp3, user defined file formats, etc.

Opening a File

- ▶ To open a file the `fopen` function is used
`FILE *fopen(const char * name, const char * mode)`
- ▶ `name`: name of the file (OS level)
- ▶ `mode`: indicates the type of the file and the operations that will be performed

```
FILE *fptr;  
fptr = fopen("myfile.txt", "r");
```

Mode Strings

String	Meaning
"r"	Open for reading, positions at the beginning
"r+"	Open for reading and writing, positions at the beginning
"w"	Open for writing, truncate if exists, positions at the beginning
"w+"	Open for reading and writing, truncate if exists, positions at the beginning
"a"	Open for appending, does not truncate if exists, positions at the end
"a+"	Open for appending and writing, does not truncate if exists, positions at the end

A `b` or `t` can be added to indicate it is a binary/text file

Closing a File

- ▶ `int fclose(FILE *fp);`
- ▶ Forgetting to close a file might result in a loss of data
- ▶ After a file is closed it is not possible anymore to read/write

```
1     FILE *fptr;
2     fptr = fopen("myfile.txt", "r");
3     if (fptr == NULL) {
4         printf("Some error occurred!\n");
5         exit(1);
6     }
7     ...
8     /* do some operations */
9     fclose(fptr);
10    ...
```

Reading/Writing

Prototype	Use
<code>int getc(FILE *fp)</code>	Returns next <code>char</code> from <code>fp</code>
<code>int putc(int c, FILE *fp)</code>	Writes a <code>char</code> to <code>fp</code>
<code>int fscanf(FILE* fp, char * format, ...)</code>	Gets data from <code>fp</code> according to the format string
<code>int fprintf(FILE* fp, char * format, ...)</code>	Outputs data to <code>fp</code> according to the format string

Line Input and Line Output

```
char *fgets(char *line, int max, FILE *fp);
```

- ▶ Already seen with stdin
- ▶ Used for files as well

```
int fputs(char *line, FILE *fp);
```

- ▶ Outputs/writes a string to a file

Files: Example 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char ch;
5     FILE *fp;
6     fp = fopen("file.txt", "r");
7     if (fp == NULL) {
8         printf("Cannot open file!\n");
9         exit(1);
10    }
11    ch = getc(fp);
12    while (ch != EOF) {
13        putchar(ch);
14        ch = getc(fp);
15    }
16    fclose(fp);
17    return 0;
18 }
```



Files: Example 2

```
1 # include <stdio.h>
2 # include <stdlib.h>
3 int main () {
4     char ch;
5     FILE * fp;
6     fp = fopen("file.txt", "r") ;
7     if (fp == NULL) {
8         printf("Cannot open file!\n");
9         exit(1);
10    }
11    while((ch=getc(fp))!=EOF) {
12        putchar(ch);
13    }
14    fclose(fp);
15    return 0;
16 }
```

Files: Example 3

```
1 # include <stdio.h>
2 # include <stdlib.h>
3 int main () {
4     char ch;
5     FILE * fp;
6     fp = fopen("file.txt", "r") ;
7     if (fp == NULL) {
8         printf("Cannot open file!\n");
9         exit(1);
10    }
11    while(!feof(fp)) {
12        ch=getc(fp);
13        if (ch!=EOF)
14            putchar(ch);
15    }
16    fclose(fp);
17    return 0;
18 }
```



Revisiting: Casting

- ▶ It is possible to overcome standard conversions (casting)
- ▶ To force to a different data type, put the desired data type before the expression to be converted
(type name) expression
- ▶ Casting is a unary operator with high precedence

Revisiting: Casting – Example

```
1   int a;
2   float f1 = 3.456, f2 = 1.22;
3   /* these operations imply demotions */
4   a = (int) f1 * f2;    /* a is now 3 */
5   a = (int) (f1 * f2); /* a is now 4 */
```

- ▶ You have already used casting when using `malloc`. `malloc` returns a `void *` pointer (i.e., a generic pointer) and this is assigned to a non `void *` pointer
- ▶ If you omit the casting you might get a warning concerning a possible incorrect assignment

Revisiting: String Functions

- ▶ Defined in `string.h`
- ▶ `strlen` determines the length of a string
- ▶ `strcat` concatenates two strings
- ▶ `strcpy` copies one string into another
- ▶ `strcmp` compares two strings
- ▶ `strchr` searches a char in a string
- ▶ `strdup` duplicates a string
- ▶ See man pages (`man 3 string`) or section B3 in the Kernighan & Ritchie book

Revisiting: `void *`

- ▶ `void *` is a generic pointer holding a memory address
 - ▶ `malloc` returns a `void *`, thus the need for a cast
- ▶ Every pointer can be assigned to a `void *` pointer and vice versa, without explicit casts
 - ▶ This can create big problems

Revisiting: Misusing void*

```
1 int main(void) {
2     void * vp;      /* a generic pointer */
3     int * ip;
4     float f = 1.234,
5     float * fp = &f;
6     vp = fp;
7     ip = vp;
8     /* float * assigned to int *
9        via a generic pointer
10        this will not work correctly ...
11    */
12    printf("%d\n", *ip);
13    /* outputs some strange number */
14    return 0;
15 }
```

Examples Revisited (1)

```
1 char a_string[] = "This is a string\0";
2 char *p;
3 int count = 0;
4 int main() {
5     printf("%s\n", a_string);
6     for (p = &a_string[0]; *p != '\0'; p++)
7         count++;
8     printf("The string has %d chars\n", count);
9     p--;
10    printf("Printing the reverse string:\n");
11    while (count > 0) {
12        printf("%c", *p);
13        p--;
14        count--;
15    }
16    return 0;
17 }
```

Examples Revisited (2)

To specify the dimension on the fly you can use the `malloc` function defined in the header file `stdlib.h`

```
1 ...
2 int *dyn_array, how_many, i;
3 printf("How many elements? ");
4 scanf("%d", &how_many);
5 dyn_array = (int*) malloc(sizeof(int) *
6     how_many);
7 if (dyn_array == NULL)
8     exit(1);
9 for (i = 0 ; i < how_many; i++) {
10     printf("\nInput number %d:", i);
11     scanf("%d", &dyn_array[i]);
12 }
13 ...
```

Examples Revisited (3) – Reading from the Keyboard

```
1 #include <stdio.h>
2 int main() {
3     int v;
4     char str[30];
5     char line[80];
6     printf("Enter a string: ");
7     fgets(line, sizeof(line), stdin);
8     sscanf(line, "%s", str); // not really needed
9     // just read str directly
10
11     printf("Enter a number: ");
12     fgets(line, sizeof(line), stdin);
13     sscanf(line, "%d", &v);
14     printf("String: %s\n", str);
15     printf("Number: %d\n", v);
16     return 0;
17 }
```


Conversion Specification for printf()

Conversion	Meaning
<code>%c</code>	single character
<code>%d</code>	signed decimal integer
<code>%f</code>	double (also float)
<code>%e</code>	floating point (exponential notation)
<code>%s</code>	string (pointer needs to be passed)

Conversion Specification for scanf()

Conversion	Meaning
as above	
%f	float (decimal notation)
%lf	double (decimal notation)

Final Exam: Details

- ▶ Information about exact time and location of the final exam will follow
- ▶ Programming exercises to be solved on paper
 - ▶ You have two hours to solve exercises
 - ▶ Similar to the programming assignments
 - ▶ Practice to write your programs on paper
- ▶ You do not need paper, it will be provided
- ▶ You may not use books or other documentation while taking the exam
- ▶ You may not use mobile phones, calculators or any other electronic devices