1. Pointer to void. Standard functions of memory area processing.

- What void * is used for in C (processing memory areas and passing anything to a function)
- memcpy, memmove, memset, memcmp

Uses:

- is useful for **referencing an arbitrary memory location**, regardless of the object placed there;

  void *memcpy(void *dst, const void *src, size_t n);

  For example, we implement a function that *copies one memory location to another*. We are only interested in **where this memory is located** and the size of the stored data.

- allows passing a **pointer to an object of any type** into a function

```
void qsort(void *first, size_t number, size_t size, int
(*comparator)(const void *, const void *));
```

Usage Features:

- **assignment of a pointer of void** type to **a pointer of any other type is** allowed without explicit conversion of the pointer type.

  ```
  double a = 1.0;
  double *p_a = &a;
  void *p_aa = p_a; p_a
  = p _ a a ;
  ```

- a pointer **cannot be dereferenced** because the size of the type the pointer points to is unknown.

  ```
  printf("%lf\n", *p_aa); //ERROR
  ```

- **address arithmetic is not applicable to** pointers of void type (for the same reasons as the point above)

  ```
  p_aa++; //ERROR
  ```

Memory area processing functions:

*all functions are defined in *string.h*

function prototype	description
void *memcpy(void *restrict dst, const void *restrict src, size_t count)	Copies count bytes of the memory block referenced by `src` to the second memory block referenced by the `dst` pointer Behavior **is undefined:** - memory **blocks overlap** - invalid signs Returns a pointer to dst
void *memmove(void *dst, const void *src, size_t count)	Copies count bytes of the memory block referenced by `src` to the second memory block referenced by the `dst` pointer. The copying is done through an intermediate buffer, which **allows the function to be used when memory areas overlap** Behavior **is undefined:** -invalid pointers

	Returns a pointer to dst
int memcmp(const void *s_1, const void *s_2, size_t count)	Compares the first count bytes of the s_1 pointer memory block with the first count bytes of the s_2 pointer memory block Behavior **is undefined:** - invalid signs Returns: - **0:** Contents of both memory blocks **equals** - **> 0: The** first memory block is **larger,** than the other - **< 0: The** first memory block is **smaller,** than the other
void *memset(void *dst, int c, size_t n)	Fills count byte at address dst. The code of the character to be filled is c Returns a pointer to dst

2. Dynamic memory allocation functions.

- malloc, calloc, realloc, free: basic algorithm of use, peculiarities of work
- Whether and why to use explicit type conversion
- Allocating 0 bytes of memory
- Processing of NULL returned by the function

Features:

- all functions are located in *stdlib.h*
- functions do not create a variable, they only allocate a memory location.

  As a result, the functions **return the** address of the location of this area in computer memory, that is, the **pointer**
- return pointer type - **void**
- in case of an error during memory allocation return the value **NULL**
- after use it is necessary to **free** the memory with **free**

function prototype	operating principle
void *malloc(size_t size);	- allocates the block of memory specified in bytes - memory block is **not initialized**

	-the sizeof operation is used to calculate the required memory size
void *calloc(size_t nmemb, size_t size);	- allocates a block of memory for the array from nmeb elements, each of which size - **initializes** memory block **0**
void free(void *ptr);	- **frees the** previously allocated block of memory pointed to by the pointer - if *ptr == NULL, nothing happens - if the pointer **was not retrieved** using malloc, calloc, realloc, the **behavior is undefined**
void *realloc(void *ptr, size_t size);	● ptr == NULL && size != 0: memory allocation as malloc ● ptr != NULL && size == 0: release memory as free ● ptr != NULL && size != 0: memory reallocation. Worst case: - select a new area - copy data from old to new - clear the old area

+/- explicit lead-in:

+	-
- **C++** compilation compiler - malloc had a prototype char before ANSI *malloc - additional argument checking	- from ANSI onwards, the conversion is not necessary - can hide an error if you don't connect *stdlib.h* - if the pointer type is changed, the type in the conversion will have to be changed as well

What happens if you allocate 0 bytes of memory?

The result of calling the malloc, calloc and gealloc functions when the requested block size is 0 depends on the compiler implementation:

- will return a null pointer;
- will return a *"normal"* pointer, but it cannot be used for dereferencing;

Therefore, before calling these functions, you must make sure that the block size is not **0!**

P. S. In case of **realloc**, when called from 0 memory block, it will work as **free**

Handling of NULL returned by the function:

- Debugger: **valgrind, Dr. Memory**
- Error return
- **Segmentation** error **(segfault)**
- **Abort** is an idea of Kernighan and Ritchie
- Recovery - **xmalloc** from git

3. Allocating memory for a dynamic array. Typical errors when working with dynamic memory

- function returning a dynamic array as a result and via a parameter

	as a return value	as a function parameter
Prototype	**int *create_array(FILE *f, size_t *n);**	**int create_array(FILE *f, size_t *n, int **arr);**
Challenge	**int *arr;** **size_t n;** **arr = create_array(f, &n);**	**int *arr, rc;** **size_t n;** **rc = create_array(f, &n, &arr);**

Typical errors when working with dynamic memory:

- **incorrect calculation** of memory allocation

struct date *p = NULL	**int n = 5;**

```
p = malloc(sizeof(p));
if (p)
{
        p->day = d a y ;
        …
```

```
int *arr = NULL;

arr = malloc(n);
if (arr)
{
     for (int i = 0; i < n; i++)
            arr[i] = i;
            …
```

- **no check** after memory allocation
- memory leaks

```
int *p = NULL;

p = malloc(sizeof(int)); // as it may be
NULL

if (p)

{

        *p = 5;

        p = malloc(sizeof(int));

        …
```

- **wild pointer:** using an uninitialized pointer

```
int *p;

if (scanf("%d", p) == 1)

{

        …
```

- **dangling pointer:** use pointer immediately after release

```
free(p);

*p = 7;

printf("%d\n", *p);
```

- **changing the pointer** returned by the memory allocation function

```
int *ptr = malloc(sizeof(int));
ptr++;
printf("%d\n", *ptr);
```

- **double freeing**
- **releasing unallocated** memory or **non-dynamic** memory
- dynamic array overrun

4. Pointers to a function. Function qsort.

- What they are used for
  - callback
    - for data processing
    - for signaling (event programming)
  - transition table
  - dynamic linking
- How pointers to a function are described, initialized, how a function call is made on a pointer, specifics
- qsort

What they are used for:

- **callback function** - passing executable code as one of the parameters of another code. A callback allows a function to execute the code that is specified in the arguments to the function when it is called

```
void populate_array(int *array, size_t size, int (*get_next_value)(void))
{
        for (size_t i=0; i<array; i++)
                array[i] = get_next_value();
}
int get_next_value(void)
{
        return rand();
}
```

```
int main(void)

{

        int array[10];

        populate_array(array, 10, get_next_value);

        ---

}
```

- **jump table** is a method of transferring program control (branching) to another part of the program (or to another program that may have been dynamically loaded) using a **jump** instruction table.

```
typedef void (*Handler)(void); // Pointer to the handler function

void func3 (void) { printf("3\n" ); } // functions
void func2 (void) { printf( "2\n" ); }
void func1 (void) { printf( "1\n" ); }
void func0 (void) { printf( "0\n" ); }

Handler jump_table[4] = {func0, func1, func2, func3};

int main (int argc, char **argv) {
        int value;

        /* Convert first argument to 0-3 integer (Hash) */ value
        = atoi(argv[1]) % 4;
        if (value < 0) {
             value *= -1;
        }

        // Call the corresponding function
        jump_table[value]();
}
```

- **dynamic blinding (blinding)** - connection of a precompiled and stored as a separate so called "**blinder"** to a program.
  "dynamic library file" of the module, executed with a special command during the main program.

Description of the pointer to the function:

<return type>(*<name>)(<argument type>).

typedef <return type>(*<name_t>)(<argument type>)

- Declaring a pointer to a function

  double trapezium(double a, double b, int n, double (*func)(double))

- Getting the address of the function

  result = trapezium(1.0, 2.0, 2, &sin /* sin */)

- Function call by pointer

  y = func(x) // y = (*func)(x)

Usage Features:

- the expression from the function name is implicitly **converted to a pointer to the function**

  ```
  int add(int a, int b);

  ...

  int (*p1)(int, int) = add;
  ```

- **The "&"** operation for a function **returns a pointer to the function**, but this is an unnecessary operation.

  ```
  int (*p1)(int, int) = &add /* add */;
  ```

- the **"*"** operation for a function pointer **returns the function itself**, which is implicitly converted to a function pointer

  ```
  int (*p3)(int, int) = *add; int

  (*p4)(int, int) = *****add;
  ```

- function pointers **can be compared**

  ```
  if (p1 == add)

      printf("p1 points to add\n");
  ```

- a pointer to a function can be the type of the function return value
- when using address arithmetic, the pointer may point to a memory location other than the function.

qsort:

```
void qsort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void * ));
```

- ***base** - pointer to the first element of the sorted array
- **nmemb** - number of elements in the array
- **size - size** of one element
- **compare** - compare function

Suppose we need to order an array of integers:

```
int compare(const void *p, const void *q)
{
        return *(int *)p - *(int *)q;
}
…
int a[10];
…
qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]), compare);
```

qsort sorts the array pointed to by the base parameter using *quicksort, Hoare's sorting algorithm*

5. The make utility. Purpose. Simple build script

- What it's for
- Ideas at the heart of the make utility: what inputs are needed for make to perform its functions.
- Varieties of make
- Details about the rules (components, what is used for what, types of rules, what components may be missing)
- A simple build script and how make works on it

Make is a utility **designed** to automate the conversion of files from one form to another.

The conversion rules are specified **in a script** named makefile, which must be located in the root of the project working directory.

The make utility uses **information from the make file and the time each file** was **last modified** to decide which files need to be updated

Principle of operation:

It is necessary to create a so-called **project build script** (make-file)**.** It describes

- **file relationship**
- contains **commands to update each file**

The script itself consists of a set of rules, which in turn are described:

- *goals* (what you need to do)
- *dependencies* (what you need to build from)
- *in commands* (the way you have to assemble)

```
target: dependencies
        team 1
        …
        command n
{sources} ---> [broadcast] ---> {objects}
{objects} ---> [linking] ---> {executable files}
```

Varieties of make:

- **GNU make**
- **BSD make (pmake)** - roughly corresponds to GNU make in terms of functionality
- **nmake (Microsoft make)** - works under Windows, little functionality, only basic principles of make.

All varieties are based on the same principles, but differ in syntax, so **they are not compatible** among themselves

About the rules (in detail):

A simple make-file consists of rules:

```
target: dependencies
        team 1
        …
        command n
```

Typically, **the target** represents the name of the file we are collecting

For example. executable or object-based.

A phony target is a target that is not really a file name. It is just the name of some command sequence (e.g. all, clean).

```
(siren wailing in distance)
    rm *.o *.exe
```

Dependencies are the files from which the target is formed. A rule can also have no dependencies:

For example, clean

```
clean : (no dependencies)
    rm *.o *.exe
```

A command is an action performed by a utility. A target can have several commands

A **rule** describes when and how to update the files specified in it as a create or update target.

```
(siren wailing in distance)
    rm *.o *.exe
    echo "dir is cleaned!"
```

A rule can also **describe** how an action should be performed

A simple script and an example of make working on it:

```
greeting.exe : hello.o bye.o main.o
        gcc -o greeting.exe hello.o bye.o main.o
```

```
test_greeting.exe : hello.o bye.o test.o
	gcc -o test_greeting.exe hello.o bye.o test.o

hello.o : hello.c hello.h
	gcc -std=c99 -Wall -Werror -pedantic -c hello.c

bye.o : bye.c bye.h.
	gcc -std=c99 -Wall -Werror -pedantic -c bye.c

main.o : main.c hello.h bye.h
	gcc -std=c99 -Wall -Werror -pedantic -c main.c

test.o : test.c hello.h bye.h
	gcc -std=c99 -Wall -Werror -pedantic -c test.c

clean :
	rm *.o *.exe
```

First launch:

1. **make** reads the build script and starts executing the first rule

```
greeting.exe : hello.o bye.o main.o
	gcc -o greeting.exe hello.o bye.o main.o
```

2. To fulfill this rule, you must first **process the dependencies**
3. **make is** looking for a rule to create a hello.o file. The hello**.o** file does not exist, but the hello.**c** and hello**.h** files do. Therefore, the rule to create hello**.o** can be executed

```
hello.o : hello.c hello.h
```

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

4. **The bye.o and main.o** dependencies are handled similarly.
5. All dependencies are obtained, now the rule for building **greeting.exe** can be executed

```
gcc -o greeting.exe hello.o bye.o main.o
```

hello.c has been modified

Second start: 1 - 3 by analogy

4. **The hello.o, hello.c, and hello.h** files exist, but the hello.**o** change time is less than the time it takes to modify hello.**c**. You will have to recreate the **hello.o** file
5. **The bye.o and main.o** dependencies are handled similarly, but these files were modified later than the corresponding C files, i.e. nothing needs to be done.
6. All dependencies are obtained. The time to modify **greeting.exe is** less than the time to modify **hello.o. We** will have to recreate greeting**.exe**

```
gcc -o greeting.exe hello.o bye.o main.o
```

6. The make utility. Purpose. Variables. Template rules

- What it's for
- Ideas at the heart of the make utility: what inputs are needed for make to perform its functions.
- Varieties of make
- Briefly about the rules
- Description of variables, what they are used for, examples
- Implicit variables and rules, what are they, why
- Automatic variables and template rules

Make is a utility **designed** to automate the conversion of files from one form to another.

The conversion rules are specified **in a script** named makefile, which must be located in the root of the project working directory.

The make utility uses **information from the make file and the time each file** was **last modified** to decide which files need to be updated

Principle of operation:

It is necessary to create a so-called **project build script** (make-file)**.** It describes

- **file relationship**
- contains **commands to update each file**

About the rules (briefly):

The script itself consists of a set of rules, which in turn are described:

- *goals* (what you need to do)
- *dependencies* (what you need to build from)
- *in commands* (the way you have to assemble)

```
target: dependencies
        team 1
        …
        command n
{sources} ---> [broadcast] ---> {objects}
{objects} ---> [linking] ---> {executable files}
```

Varieties of make:

- **GNU make**
- **BSD make (pmake)** - roughly corresponds to GNU make in terms of functionality
- **nmake (Microsoft make)** - works under Windows, little functionality, only basic principles of make.

All varieties are based on the same principles, but differ in syntax, so **they are not compatible with** each other

About the variables:

- Lines starting with **#** - *comments*
- Variable *definition*: **VAR_NAME := value**
- *Getting the value of a* variable: **$(VAR_NAME)**

Example:

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror
OBJS := hello.o bye.o
app.exe: $(OBJS) main.o
	$(CC) -o app.exe $(OBJS) main.o

hello.o : hello.c hello.h
	$(CC) $(CFLAGS) -c hello.c

bye.o : bye.c bye.h.
	$(CC) $(CFLAGS) -c bye.c

main.o : main.c bye.h hello.h
	$(CC) $(CFLAGS) -c main.c

clean:
	rm *.o *.exe
```

Implicit rules and variables:

Implicit rules (implicit rules) indicate make to some *"standard"* methods of file processing, so that the user can use them without having to go through the **detailed description of the processing method** every time.

- **-p** - **shows** implicit rules and variables
- **-r** - **prohibits the** use of implicit rules

```
OBJS := hello.o bye.o


app.exe: $(OBJS) main.o
```

```
	$(CC) -o app.exe $(OBJS) main.o


clean:

	rm *.o *.exe
```

For example, there is an implicit rule for compiling C source files. The question of running one or another rule is decided based on the names of the files to be processed. As a rule, for example, when compiling C programs, the "input" file with the extension `.c' is converted into a file with the extension `.o'. Thus, if there is such a combination of file extensions, make can apply an implicit rule to them to compile C programs.

Automatic Variables:

- are variables with special names that **automatically take certain values** before executing the commands described in the rule.

- **$^ - the** entire list of dependencies
- **$@ -** target name
- **$< is the** first dependency

```
CC := gcc

CFLAGS := -std=c99 -Wall -Wpedantic -Werror

OBJS := hello.o bye.o

app.exe: $(OBJS) main.o

	$(CC) -o $@ $^


hello.o : hello.c hello.h

	$(CC) $(CFLAGS) -c $<


bye.o : bye.c bye.h.

	$(CC) $(CFLAGS) -c $<
```

```
main.o : main.c bye.h hello.h
	$(CC) $(CFLAGS) -c $<


clean:
	rm *.o *.exe
```

Template rules:

Pattern rules allow you to specify a rule for converting one file to another **based on the dependencies between their names.**

```
%.expand_files_targets: %.expand_files_targets
	command 1
	…
	command n
```

Example:

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror
OBJS := hello.o bye.o


app.exe: $(OBJS) main.o
	$(CC) -o $@ $^


%.o : %.c *h
	$(CC) $(CFLAGS) -c $<


clean:
	rm *.o *.exe
```

! % is exactly one, any non-empty string !

The `%' symbol in the **dependency** template rule means the **same basis** that corresponds to the `%' symbol in the **target** name. In order for a template rule to be applied to a file under consideration, the name of that file must match the target template, and the dependency templates must produce the names of files that exist or can be obtained. These files will become dependencies of the target file under consideration.

7. The make utility. Purpose. Conditional constructs. Dependency analysis

- What it's for
- Ideas at the heart of the make utility: what inputs are needed for make to perform its functions.
- Varieties of make
- Briefly about the rules
- Conditional constructions
    - explicitly conditional designs
    - target-dependent variables
- Dependency analysis
    - Manual
    - "All .c files depend on all .h files."
    - Involvement of the compiler

Make is a utility **designed** to automate the conversion of files from one form to another.

The conversion rules are specified **in a script** named makefile, which must be located in the root of the project working directory.

The make utility uses **information from the make file and the time each file** was **last modified** to decide which files need to be updated

Principle of operation:

It is necessary to create a so-called **project build script** (make-file)**.** It describes

- **file relationship**
- contains **commands to update each file**

About the rules (briefly):

The script itself consists of a set of rules, which in turn are described:

- *goals* (what you need to do)
- *dependencies* (what you need to build from)

- *in commands* (the way you have to assemble)

```
target: dependencies
        team 1
        …
        command n
{sources} ---> [broadcast] ---> {objects}
{objects} ---> [linking] ---> {executable files}
```

Varieties of make:

- **GNU make**
- **BSD make (pmake)** - roughly corresponds to GNU make in terms of functionality
- **nmake (Microsoft make)** - works under Windows, little functionality, only basic principles of make.

All varieties are based on the same principles, but differ in syntax, so **they are not compatible with** each other

Conditional construction:

The conditional construct causes make to process or ignore part of the make file depending on the value of certain variables.

There are three directives used in this conditional construct: **ifeq, else,** and **endif.**

conditional-directive
snippet-for-executed-conditions
ifeq
	conditional-directive
	snippet-for-executed-conditions
	else
	fragment-for-unfulfilled-conditions
	endif

```
ifeq ($(mode), release)
        CFLAGS += -g0
```

```
endif


ifeq ($(mode), debug):
        CFLAGS += -DNDEBUG -g3
endif
```

Target-dependent variables:

```
debug : CFLAGS += -g3
release : CFLAGS += -DNDEBUG -g0
```

Dependency Analysis:

1. Manual enumeration of *.h files

   This is a bad approach, because it is impossible to trace the whole chain of dependencies of header files. Besides, you may forget to specify the necessary files.

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror OBJS
:= hello.o bye.o
app.exe: $(OBJS) main.o
        $(CC) -o $@ $^


hello.o : hello.c hello.h
        $(CC) $(CFLAGS) -c $<


bye.o : bye.c bye.h.
        $(CC) $(CFLAGS) -c $<


main.o : main.c bye.h hello.h
```

```
	$(CC) $(CFLAGS) -c $<


clean:

	rm *.o *.exe
```

2. Connecting all *.h files

```
CC := gcc

CFLAGS := -std=c99 -Wall -Wpedantic -Werror OBJS

:= hello.o bye.o


app.exe: $(OBJS) main.o

	$(CC) -o $@ $^


%.o : %.c *h

	$(CC) $(CFLAGS) -c $<


clean:

	rm *.o *.exe
```

3. Automatic generation of dependencies

For each source file **`name.c'**, there is a makefile `name**.d'** that lists the list of files on which the object file **`name.o'** depends. With this approach, new dependency lists can only be built for those source files that have actually been modified.

```
CC := gcc

CFLAGS := -std=c99 -Wall -Wpedantic -Werror OBJS

:= hello.o bye.o

SRCS := $(wildcard *.c) # all *.c files
```

```
app.exe: $(OBJS) main.o
	$(CC) -o $@ $^

%.o : %.c
	$(CC) $(CFLAGS) -c $<

%.d : %.c
	$(CC) -M $< > $@

include $(SRCS: .c = .d)

clean:
	rm *.o *.exe
```

The point of automatic dependency generation is to get a universal make-file that can be used in different projects

-M: For each source file, the preprocessor will output a list of dependencies to the standard output as a rule for the make program. The dependency list includes the source file itself, as well as all files included with the ***#include <filename>*** **and** ***#include*** directives
"file_name". After the preprocessor is started, the compiler stops and no object files are generated.

Once the dependency files have been generated, we need to **make** them available to the **make** utility. This can be accomplished by using the **include** directive.

8. Dynamic matrices. Representation as a one-dimensional array and as an array of pointers to rows. Analysis of advantages and disadvantages.

- Compare the submissions with each other; ideally write a spreadsheet with comparison criteria.

You can take(imho idk) as criteria:

- relative complexity of initial initialization
- relative complexity of memory allocation, freeing memory
- use as a one-dimensional array
- array outlier tracking
- swap strings between each other via a pointer (I don't know if it's a stretch, but let it be)

One-dimensional array

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

int main()
{
    size_t n = 10, m = 10;
    double *mat = malloc(n * m * sizeof(double));
    if (!mat)
        return NULL;

    for (size_t i = 0; i < n; ++i)
    {
        for (size_t j = 0; j < m; ++j)
        {
            mat[i * m + j] = 0;
        }
    }

    free(mat);

    return 0;
}
```

Advantages	Disadvantages
• Easy to isolate and memory release • Possibility to use as one-dimensional array	• The debugger may not be able to trace beyond the array boundaries • Complex indexing

Array of pointers to rows (columns)

```
double **allocate_matrix(size_t n, size_t m)
{
    double **data = calloc(n, sizeof(double *));
    if (!data)
        return NULL;

    for (size_t i = 0; i < n; ++i)
```

```
    {
        data[i] = malloc(m * sizeof(double));
        if (!data[i])
        {
            free_matrix(data, n); // it is possible to do without calloc and
            instead of
n pass i
            return NULL;
        }
    }
}

void free_matrix(double **data, size_t n)
{
    for (size_t i = 0; i < n; ++i)
        free(data[i]);
    free(data);
}
```

Advantages	Disadvantages
• The debugger can trace beyond the array boundaries • Possibility to exchange strings via pointers	• Complexity of memory allocation and freeing • Memory for the matrix does not lie in one area

Separate allocation for pointer array

```
double **allocate_matrix(size_t n, size_t m)
{
    double **ptrs, *data;
    ptrs = malloc(n * sizeof(double *));
    if (!ptrs)
        return NULL;

    data = malloc(n * m * sizeof(double));
    if (!data)
    {
        free(ptrs);
        return NULL;
    }

    for (size_t i = 0; i < n; ++i)
        ptrs[i] = data + i * m;

    return ptrs;
}

void free_matrix(double **ptrs)
{
```

```
    free(ptrs[0]);
    free(ptrs)
}
```

Advantages	Disadvantages
• Easy memory allocation and release • Possibility to exchange strings via pointers • Can be used as a one-dimensional array	• Complexity of initial initialization • The debugger cannot trace beyond the array boundaries

Pointer array and data array in one area

```
double **allocate_matrix(size_t n, size_t m)
{
    double **data = malloc(sizeof(double *) * n + \
                                        sizeof(double) * n * m);
    if (!data)
        return NULL;

    for (size_t i = 0; i < n; ++i)
    {
        data[i] = (double *)((char *)data + n * sizeof(double *) + \
                                                i * m * sizeof(double));
    }
    return data;
}

free(mat);
```

Advantages	Disadvantages
• Easy memory allocation and release • Can be used as a one-dimensional array • String rearrangement via pointer exchange	• Complexity of initial initialization • The debugger cannot trace beyond the array boundaries

9. -//-
10. -//-
11. -//-
12. -//-
13. -//-

14. Reading complex announcements

[]	Array type
[N]	Array of N elements of type
(type)	A function that takes an argument of type type and returns
*****	Pointer to

Decoding from within. Start from identifier Preference [], ()
over *
*name[] is an array of type …
*name() is a function that takes …
Cannot create an array of functions `int a [10](int)`
A function cannot return a function
`int g(int)(int);`
The function cannot return an array
`int f(int)[]`

15. Strings in dynamic memory. POSIX functions and GNU extensions that return such strings

- Strings in dynamic memory are no different from strings in static memory
- Pay attention to the features of line selection:
  `(size + 1) * sizeof(char)`, since the character size may vary due to wide characters in different OS.
- Feature Test Macro

You should specify the gnu99 standard when compiling, using linux.
char *strdup(const char *) - returns a copy of the specified string, POSIX standard
char *strndup(const char *, size_t n) - analog of strdup, but copies no more than n elements For getline you should use Feature Test Macro - allows the library to conform to several standards. _GNU_SOURCE/_POSIX_C_S_SOURCE/in stdio.h
ssize_t getline(char **lineptr, size_t *n, FILE *stream) - reading a line. Returns the number of characters read (-1 in case of error). *lineptr - either NULL or din. string. POSIX standard.
The sprintf function is neither part of the standard library nor part of POSIX, it is an extension of the GNU library.
int snprintf(char *restrict buf, size_t num, const char *restrict format, ...); - output format string into a dynamic string. If you pass NULL and 0, the function will return the estimated length of the string. Maximum num - 1 character is stored.
int asprintf(char **strp, const char *fmt, ...) - independently allocates memory for the string in strp. Returns the number of characters written.

Property testing macros allow the programmer to control which definitions will be available from system header files when the program is compiled.

16. Peculiarities of using structures with pointer fields

- The = operation for structures of the same type is essentially a bitwise copying of the memory area of one variable into the memory area of another. What bad or good can be caused by the presence of a pointer field in a structure
- Surface and deep copying
- Recursive memory release for structure with dynamic fields

C defines an assignment operation for structures of the same type. It is actually equivalent to copying one memory location of a variable to another.
However, in case structures contain pointers, you should be careful - because only the pointer itself is copied, not its contents. This phenomenon is called ***surface copying.***
This phenomenon can lead to double-release errors and loss of memory area of one structure in case of assignment.
In case of ***deep copying,*** not only the structure fields but also their contents are copied.
This implementation requires separate functions for memory allocation and release.
To free memory from a structure containing pointers to dynamically allocated memory, you must first free the addresses on the pointers and then the structure itself.

17. Variable size structures

- What's that
- Examples where such structures may be encountered in practice
- Implementation within the framework of Ci
- Flexible Array Member
- FAM to c99
- Desirable: allocating memory for such a structure, saving to file, reading from file.

```
struct s
{
    int n;
    double d[];
}
```

Variable-size structures are structures that can change the size of the memory area they occupy.

Example : TLV encoding - Type Len Value. Used in TCP/IP protocol family, PC/SC specification (smart cards, usb keys), ASN.1 notation in encoding (certificates, signed message, secure message).

Flexible array member field Flexible array member field implementation:

1. Such a field should be the last field in the structure;
2. You can't create an array of such structures;
3. Such a structure cannot NOT be the last field in another structure;
4. The sizeof operation does not take into account the size of the field;
5. In the case of accessing an array that has no elements, undefined behavior.

To create such a structure, we need to allocate memory for the structure and an array of n elements.

DO C99:

```
struct s
{
    int n;
    double d[1];
};

struct s* create_s(int n, const double *d)
{
    assert(n >= 0);

    struct s *elem = calloc(sizeof(struct s) +
                            (n > 1 ? (n - 1) * sizeof(double) : 0), 1);
    if (elem)
    {
        elem->n = n;
        memmove(elem->d, d, n * sizeof(double));
    }

    return elem;
}
```

"unwarranted chumminess with the C impleme
(c) Dennis Ritchie

AFTER C99:

```
struct s *arr_create(int n, const double *d)
{
   struct s *answer = NULL;
    if (n > 1)
        answer = calloc(sizeof(struct s) + n * sizeof(double)); if
    (answer)
    {
        answer->n = n;
        memmove(answer->d, d, n * sizeof(double));
    }

    return answer;
}
```

Before C99, the array size '1' was specified in the flexible array field. When creating the array, the required array length was checked - if it was greater than 1, memory was allocated using calloc.
https://wiki.c2.com/?StructHack

The advantage is data atomicity, data economy, one-time memory allocation/release and easy copying of structures.
To work with the file you will need fwrite fread. To read - first read the constant part (the length will be known) - read the remaining data/create a new structure and read the whole file.

Up to C99.	After C99.

struct s
{
    int n;
    double d[1];
};
 	
void print_s(const struct s *elem)
{
    printf("n = %d\n", elem->n);
 	
    for (int i = 0; i < elem->n; i++)
        printf("%4.2f ", elem->d[i]);
 	
    printf("\n\n\n");
}
	struct s
	{
	    int n;
	    double d[];
	};
	
	void print_s(const struct s *elem)
	{
	    printf("n = %d\n", elem->n);
	
	    for (int i = 0; i < elem->n; i++)
	        printf("%4.2f ", elem->d[i]);
	
	    printf("\n\n\n");
	}

struct s* create_s(int n, const double *d)
{
    assert(n >= 0);
 	
    struct s *elem = calloc(sizeof(struct s) + (n > 1 ? (n - 1) * sizeof(double) : 0), 1);
 	
    if (elem)
    {
        elem->n = n;
        memmove(elem->d, d, n * sizeof(double));
    }
 	
    return elem;
}
	struct s* create_s(int n, const double *d)
	{
	    assert(n >= 0);
	
	    struct s *elem = malloc(sizeof(struct s) + n * sizeof(double));
	    /*
	    // To appease valgrind
	    struct s *elem = calloc(sizeof(struct s) + n * sizeof(double), 1);
	    */
	
	    if (elem)
	    {
	        elem->n = n;
	        memmove(elem->d, d, n * sizeof(double));
	    }
	
	    return elem;
	}

int save_s(FILE *f, const struct s *elem)
{
    int size = sizeof(*elem) + (elem->n > 1 ? (elem->n - 1) * sizeof(double) : 0);
    int rc = fwrite(elem, 1, size, f);
 	
    if (size != rc)
	int save_s(FILE *f, const struct s *elem)
	{
	    int size = sizeof(*elem) + elem->n * sizeof(double);
	    int rc = fwrite(elem, 1, size, f);
	
	    if (size != rc)

```
        return 1;

    return 0;
}
```

```
        return 1;

    return 0;
}
```

```
struct s* read_s(FILE *f)
{
    struct s part_s;
    struct s *elem;
    int size;

    size = fread(&part_s, 1, sizeof(part_s),
f);
    if (size != sizeof(part_s))
        return NULL;

    elem = calloc(sizeof(part_s) + (part_s.n >
1 ? (part_s.n - 1) * sizeof(double) : 0), 1);
    if (!elem)
        return NULL;

    memmove(elem, &part_s, sizeof(part_s));

    if (elem->n > 1)
    {
        size = fread(elem->d + 1,
sizeof(double), elem->n - 1, f);
        if (size != elem->n - 1)
        {
            free(elem);

            return NULL;
        }
    }

    return elem;
}
```

```
struct s* read_s(FILE *f)
{
    struct s part_s;
    struct s *elem;
    int size;

    size = fread(&part_s, 1, sizeof(part_s),
f);
    if (size != sizeof(part_s))
        return NULL;

    elem = malloc(sizeof(part_s) + part_s.n *
sizeof(double));
    /*
    // To appease valgrind
    elem = calloc(sizeof(part_s) + part_s.n *
sizeof(double), 1);
    */
    if (!elem)
        return NULL;

    memmove(elem, &part_s, sizeof(part_s));

    size = fread(elem->d, sizeof(double),
elem->n, f);
    if (size != elem->n)
    {
        free(elem);

        return NULL;
    }

    return elem;
}
```

18. Dynamically expandable array

- What is an array
  -+/- this SD
- Features:
  - Memory is allocated in relatively large blocks
  - Usually such an array is represented by a structure with a slightly larger data set
- Adding, deleting an item
- Features of use
  - When working with such an array, indices are used instead of pointers to elements. Why is this so?

An array is a sequence of elements of the same type and size arranged one after another.
Linearity, Homogeneity, Random access.

The advantages and disadvantages of an array are explained by the memory allocation strategy: memory for all elements is allocated in one block.
"+" Minimal overhead.
"+" Constant access time to the item. "-" Storing
a changing set of values.

```
struct arr_t
{
    int len;                    // Number of elements in the array
    int allocated;          // Number of selected array elements
    int *data;                 // Pointer to array
}
```

Adding an element (you can realloc at once):

```
int arr_add(arr_t *array, int n)
{
    if (!array->data)
    {
        array->data = malloc(INIT_SIZE * sizeof(int));
        if (!array->data)
            return ERR:
        array->allocate = INIT_SIZE;
    }
    else
    {
        if (array->len >= array->allocate)
        {
            int *p = realloc(array->data, STEP * array->allocate * sizeof(int));
            if (!p)
                return ERR;
            array->data = p;
            array->allocate *= STEP;
        }
    }
    array->data[array->len++] = n;

    return OK;
}
```

Deleting an element (can be a loop):

```
int arr_remove(arr_t *array, int ind)
{
    if (!array->data && array->len >= ind)
        return ERR;

    memove(array->data[ind], array->data[ind + 1], (array->len - ind - 1) * sizeof(int));
    array->len--;
    return OK;
}
```

Memory for a dynamic array is allocated in blocks (in some increments) to optimize memory allocation.
For a dynamically expandable array, you should use indexes rather than pointers because the array may change address when added.

In case of a dynamically expandable array there are several disadvantages - manual memory release, storing values of the number of array elements and the size of the area under the array.

19. Linear single-linked list. Adding an element, deleting an element.

- List definition
- List vs array
- Node description
- Removing memory from under the entire list
- If there's time: speculate on possible improvements:
  - Universalization (void *)
  - ...
- Story by topic

A linear singly-linked list is a data structure that stores data and a reference to the next node. The unit of the structure is called a node.
Basic operations: add to start/end, search, insert before/after, delete.

	Array	List
Access/Search	By index/binary	Serial
Add/Delete	Slow due to displacement	Quick and easy
Storage in memory	In one area	In different areas
Size	Indicated at the time of the announcement	Limited by memory

Structure Description:

```
struct node_t
{
    int data;
    struct node_t *next;
}
```

Memory is removed from under the entire list by using an additional pointer to the node preceding this one.
Adding to the beginning

```
struct person_t* list_add_front(struct person_t *head, struct person_t
*pers)
{
    pers->next = head;
    return pers;
}
```

Deletion - search for the required node and store its parent. When found - pointer replacement and memory release.

20. Linear single-linked list. Inserting an element, deleting an element.

- List definition
- List vs array
- Node description
- Removing memory from under the entire list
- If there's time: speculate on possible improvements:
  - Universalization (void *)
  - ...
- Story by topic

See 19

Insert before and after an element (by value)

Insert **before the** value:

```
struct node_t* list_insert_before(struct node_t *head, struct node_t *tmp,
int data)
{
    struct node_t *prev = NULL, *cur = head;

    while (cur && (cur->data != data))
    {
        prev = cur;
        cur = cur->next;
    }

    if (prev && cur)
    {
        prev->next = tmp;
        tmp->next = cur;
    }
    else if (cur == head && head)
    {
       tmp->next = head;
       head = tmp;
    }

    return head;
}
```

Insertion **after the** value:

```
struct node_t* list_insert_after(struct node_t *head, struct node_t *tmp,
int data)
{
    struct node_t *cur = head;

    while (cur && (cur->data != data)) cur
        = cur->next;

    if (cur)
    {
        tmp->next = cur->next;
        cur->next = tmp;
    }

    return head;
}
```

Deleting from underneath the **entire list:**

```
void list_clear(struct node_t **head)
{
    struct node_t *tmp = *head;

    while (*head)
    {
        (*head) = (*head)->next;

        free(tmp);
        tmp = NULL;
        tmp = (*head);
    }
}
```

Universalization:

- **Adding a pointer to the end of the list**
- **List item representation**
  - Universal element *(void*)*
- **Bilinked lists**
  - Requires more resources.
  - Finding the last one and deleting the current one are operations of order O(1)

21. Linear single-connected list. Traversal.

- List definition
- List vs array
- Node description
- Removing memory from under the entire list
- If there's time: speculate on possible improvements:
  - Universalization (void *)
  - ...
- Story by topic

See 19

Feature traversal:

void list_apply(struct person_t *head, void (*f)(struct person*, void*), void *arg)

```
void list_apply(struct person_t *head, void(*f)(struct person *, void *), void *arg)
{
    for(;head;head=head->next)
    {
        f(head, arg);
    }
}
```

22. Binary search tree. Adding an item

- Definition, difference from a regular tree, order relation
- Type Description
- Releasing the memory from under the whole tree
- Story by topic

A tree is a connected acyclic graph.

A DDP is a tree whose vertices are all ordered, each vertex has at most two descendants (let's call them left and right), and all vertices except the root have a parent.

Basic operations: add, search, bypass, delete.

```
struct node_t
{
    int data;
    struct node_t *left, *right;
}
```

Freeing memory from under the entire tree is accomplished using postfix traversal.

Adding an element (if the element already exists, handle this situation yourself however you want):

struct node_t *insert(struct node_t *head, struct node_t *ins);

```
struct node_t *insert(struct node_t *head, struct node_t *ins)
{
	if (!head)
		return ins;

	int cmp = head->data - ins->data;
	if (cmp == 0)
		assert(0);
	else if (cmp <
	0)
		head->left = insert(head->left, ins);
	else
		head->right = insert(head->right, ins);

	return head;
}
```

23. Binary search tree. Searching for an item

- Definition, difference from a regular tree, order relation
- Type Description
- Releasing the memory from under the whole tree
- Story by topic

See 22 U.S.C.

Search using a function (either recursively or with a loop):
struct node_t* find(struct node_t *head, int val);

```
struct node_t *find(struct node_t *head, int val)
{
	int cmp;
	while (head)
	{
		cmp = head->data - val;
		if (cmp == 0)
			return head;
		else if (cmp < 0)
			head = head->left;
		else
			head = head->right;
	}
	return NULL;
}
```

```
struct node_t *find(struct node_t *head, int val)
{
	if (!head)
		return NULL;

	int cmp = head->data - val;
	if (cmp == 0)
		return head;
```

```
    else if (cmp < 0)
        return find(head->left, val);
    else
        return find(head->right, val);
}
```

24. Binary search tree. Traversal

- Definition, difference from a regular tree, order relation
- Type Description
- Releasing the memory from under the whole tree
- Story by topic
- You can talk about the DOT language

See 22 U.S.C.

Feature traversal:

void node_view(struct node_t *head, void (*f)(struct node_t *, void*), void*);

Prefix - f view view view Infix -
view f view view Postfix - view
view view f

```
void node_view(struct node_t *head, void (*f)(struct node_t *, void *), void *param)
{
    if (!head)
        return;

    f(head, param);
    node_view(head->left, f, param);
    node_view(head->right, f, param);
}
```

DOT is a language for describing graphs.

A graph described in the DOT language is typically a text file with a .gv extension in a format understandable to humans and the processing program.

Graphs described in the DOT language are represented graphically using special programs such as **Graphviz**.

25. Binary search tree. Deleting an item

- Definition, difference from a regular tree, order relation
- Type Description
- Releasing the memory from under the whole tree
- Story by topic

See 22 U.S.C.

Deleting an element (you will need to remember the ancestor of the vertex):

node_t *node_del(node_t *head, int del);

Three cases are considered:

1. If the node has no descendants, the ancestor pointer is deleted and changed;
2. If a node has 1 descendant, the ancestor pointer to the descendant is deleted and changed;
3. If a node has 2 descendants, it searches for the minimal one in the right subtree, swaps the values of the nodes to delete and the found one, and then the found one is deleted.

Implementation of deletion by INDEX:

```
tree_node_t *find_right_min_tree(tree_node_t *tree)
{
    if (!(tree->left->left))
    {
        tree_node_t *tmp_tree = tree->left;


        tree->left = NULL;

        return tmp_tree;
    }

    tree_node_t *new_tree = find_right_min_tree(tree->left);


    return new_tree;
}


void destroy_node(void *node, void *trash)
{
    if (!trash)
        free(node);
}

tree_node_t *del_tree_node(tree_node_t *tree, int *ind)
{
    if (!tree)
        return NULL;

    if (tree->index == *ind)
    {
```

```
        *ind = -1;

        if (!(tree->left) && !(tree->right))
        {
            destroy_node((void *) tree, NULL);
            return NULL;
        }

        if (tree->left && !(tree->right))
        {
            tree_node_t *tmp_node = tree->left;


            destroy_node((void *) tree, NULL);
            return tmp_node;
        }

        if (tree->right && !(tree->left))
        {
            tree_node_t *tmp_node = tree->right;


            destroy_node((void *) tree, NULL);
            return tmp_node;
        }

        if (!(tree->right->left))
        {
            tree_node_t *tmp_node = tree->right;


            tree->right->left = tree->left;
            destroy_node((void *) tree, NULL);
            return tmp_node;
        }

        tree_node_t *new_tree = find_right_min_tree(tree->right);


        new_tree->left = tree-
        >left; new_tree->right = tree-
        >right;
```

```
        destroy_node((void *) tree, NULL);
        return new_tree;
    }

    if (tree->left && *ind > 0)
        tree->left = del_tree_node(tree->left, ind);
    if (tree->right && *ind > 0)
        tree->right = del_tree_node(tree->right, ind);

    return tree;
}
```

26. The concept of linking

- What is binding
- What types of binding are there
- How object and executable file properties are affected
- Mention static
- How the linker works
- Example, name table, what variables go where **Binding** - defines the area of the program within which the "program object" will be available to other functions.

Types of binding are external, internal and none.
The object file contains machine code and information about variables and functions that are defined or will be needed for operation.
Names with external linking fall into the .symtabl symbol table

Letter	Location
B, b	Uninitialized data section (.bss).
D, d	Initialized data (.data) section.
R, r	Read-only data section (.rodata)
T, t	Code section (.text)

U	The symbol has not been determined, but is expected to appear.

Lowercase letters stand for "local" characters, uppercase letters for external (global) characters.

```c
#include <stdio.h>

int a = 10;           // D
static int b = 5;     // d
int c;                // B
static int d;         // b
extern int e;         // U
const char f = 'e'; // R
void foo_boo(void); // U
void foo(void)        // T
{
    foo_boo();
}
static void boo(void)     // t
{
    e = 5;
}
```

27. Memory classes. General concepts. Memory classes auto and static for variables

- Properties (scope, lifetime, binding) of default variables.
- To some extent, these properties can be controlled using memory classes
- Enumeration of memory classes, peculiarities of their use Managing

lifetime, scope and binding of a variable (up to a certain extent) can be done with the help of so-called memory classes. There are 4 memory classes: auto static extern register

Default:

```
int a ; // File OV, global lifetime, external binding

{
    int b;    // Block OV, local VL, no binding
}
```

auto - applicable only to block variables. It has block OV, local VL, no binding. By default, all variables in a block or function header have this binding.

static - applies to any variable. Changes binding to internal, lifetime to global.
A global variable with memory class static has an internal binding, file scope, and global lifetime. This hides the variable in the file.

28. Memory classes. General concepts. Extern and register memory classes for variables

- Properties (scope, lifetime, binding) of default variables.
- To some extent, these properties can be controlled using memory classes
- Enumeration of memory classes, peculiarities of their use

See id. at 27

extern - applicable to any variables. Changes lifetime to global, binding to parent binding. It helps to "export" variables from other files. In the case when a variable is declared extern first, then static, there will be an error, because in such a case the binding will be changed, which contradicts the standard.
register - applies to local variables. It is a request of the compiler to place this variable in a processor register for quick work with it. The addressing operation cannot be applied to it.

29. Memory classes. General concepts. Memory classes for functions

- Default properties (scope, lifetime, binding) of functions.
- To some extent, these properties can be controlled using memory classes
- Enumeration of memory classes, peculiarities of their use

See id. at 27

The function has a default (be careful with OV and VL, I'm not sure about the correctness of such expressions to the function)))) :)

```
void foo(void);        // File OW, Global OW, External Binding
extern void boo(void); // File OV, Global OV, External Binding
static void too(void); // File OV, Global OV, Internal Binding
```

By default, the function has an **extern** memory class.
Static helps to hide the function and prevent it from being used from other files. (encapsulation, you can reuse the name)

30. Global variables. Journaling.

- Features the use of global variables. +/-. When you can use global variables
- 3 approaches to journaling

+:

1. Accessible from any part of the program
2. Global lifetime

-:

1. When a variable is changed, it is difficult to trace the error
2. Requires validation in all functions where it is used
3. Functions cannot be used in other programs

Logging is the process of recording information about events happening to some object (or within some process) in a **"log" (**e.g., a file).

The file variable for journaling is defined **globally** and declared in all project implementation files. This allows you to call the journaling file functions from anywhere.

For logging you can write your own function, where the variable pointer to the log file can be either a global or a static variable.

1. **<u>Using a global variable</u>**

We create a **global variable** and declare it in the **header file**. Then there are two functions - **file creation and file closing**.

On the plus side: journaling from anywhere in the program.

Cons: if we mess up a global variable, we break journaling

log.h
```c
#ifndef __LOG_H__
#define __LOG_H__

#include <stdio.h>

extern FILE *flog;

int log_init(const char *name);

void log_close(void);

#endif // __LOG_H__
```

log.c
```c
#include "log.h"

FILE *flog;

int log_init(const char *name)
{
    flog = fopen(name, "w");

    return (!flog) ? 1 : 0;
}

void log_close(void)
{
    fclose(flog);
}
```

main.c
```c
#include "log.h"

void func_1(void)
{
    fprintf(flog, "func_1\n");
}

void func_2(void)
{
    fprintf(flog, "func_2\n");
}

int main(void)
{
    if (log_init("test.log"))
    {
        printf("kek\n");
        return -1;
    }

    func_1();
    func_2();

    log_close();

    return 0;
}
```

2. **<u>Concealment</u>**

The variable is still **global. log_get** creates
copy, but not yet protected - you can call and close the file

log.h > ...

```c
#ifndef __LOG_H__
#define __LOG_H__

#include <stdio.h>

int log_init(const char *name);

FILE *log_get(void);

void log_close(void);

#endif // __LOG_H__
```

log.c > ...

```c
#include "log.h"

static FILE *flog;

int log_init(const char *name)
{
    flog = fopen(name, "w");

    return (!flog) ? 1 : 0;
}

FILE *log_get(void)
{
    return flog;
}

void log_close(void)
{
    fclose(flog);
}
```

main.c > main(void)

```c
#include "log.h"

void func_1(void)
{
    fprintf(log_get(), "func_1\n");
}

void func_2(void)
{
    fprintf(log_get(), "func_2\n");
}

int main(void)
{
    if (log_init("test.log"))
    {
        printf("kek\n");
        return -1;
    }

    func_1();
    func_2();

    log_close();

    return 0;
}
```

3. **<u>The ideal</u>** is **not log_get,** but a logging function log_message that only outputs a string.
   Minus: overloads, because you need to form a string first. But it can be implemented with a **variable number of parameters**

log.h > ...

```c
#ifndef __LOG_H__
#define __LOG_H__

#include <stdio.h>
#include <stdarg.h>

int log_init(const char *name);

FILE *log_message(const char *format, ...);

void log_close(void);

#endif // __LOG_H__
```

log.c > ...

```c
#include "log.h"

static FILE *flog;

int log_init(const char *name)
{
    flog = fopen(name, "w");

    return (!flog) ? 1 : 0;
}

FILE *log_message(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    vfprintf(flog, format, args);
    va_end(args);
}

void log_close(void)
{
    fclose(flog);
}
```

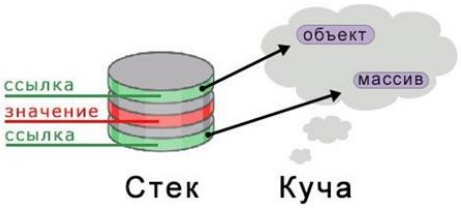
main.c > ...

```c
#include "log.h"

void func_1(void)
{
    log_message("%s", "func_1\n");
}

void func_2(void)
{
    log_message("%s", "func_2\n");
}

int main(void)
{
    if (log_init("test.log"))
    {
        printf("kek\n");
        return -1;
    }

    func_1();
    func_2();

    log_close();

    return 0;
}
```

31. Heap in a C program. Algorithms of malloc and free functions.

- What is a pile, where did the term come from
- Properties of the memory that is allocated in the heap
- Ideally write code with comments, but you can describe the algorithm in detail
- Features of realization
    - equalization
    - fragmentation

Heap is a memory storage located in RAM. It allows dynamic memory allocation and is, in fact, a simple storage for variables.



The first time Donald Knuth used the term Heap was when he said that several authors at the beginning of the
1975 referred to the **Free Memory Pool** as the **Heap. The** title of the paper and the names of the authors remained unknown to anyone, so the following conclusion was reached:

"The term Heap originated as a counterpart to the term Stack."

Memory properties:

- At least the specified number of bytes are allocated (less is not allowed, more is allowed)
- The pointer returned by malloc points to the allocated area
- No other call to malloc can allocate this area, or any part of it, unless it has been freed with free.

TODO code, and maybe in ass its

Memory Block Structure

```
struct block_t
{
    size_t *size;
    int free;
    struct block_t *next
};
```

Malloc Algorithm:

1. View the list of occupied/free memory areas in search of a free area of a suitable size
2. If the area is exactly the size requested, mark the found area as occupied and return a pointer to the beginning of the memory area
3. If the area is large, divide it into parts, one of which will be occupied (selected) and the other will remain free
4. If no region is found, return a null pointer Algorithm free:

1. View a list of occupied/free memory areas in search of the specified area
2. Mark the found area as free
3. If the freed area borders closely on a free area on either side, merge them into a single area of a larger size

merge_blocks accepts nothing as parameters. Implementation

features:

1. Fragmentation is a state in which the available memory is split into small unrelated blocks. In this case, even if the total memory size is sufficient to fulfill the request, memory allocation may fail
2. Alignment

```
union block_t
{
    struct
    {
        size_t size;
        int free;
        union block_t *next;
    } block;
    size_t align_x;
}
```

To store arbitrary objects, the block must be properly aligned.

If an element of the most demanding type can be placed at some address, then any other elements can be placed there as well.

32. Variable size arrays.

c99: variable length arrays - VLA.

```
int a[n] // n is a variable
```

- The length of such an array is calculated at runtime, not at compile time (the standard requires initialization to be performed during the

compilation, while in VLA it occurs at runtime, so you cannot initialize VLA by standard (-pedantic will ban you)).

- Memory for array elements is allocated on the stack
- The VLA cannot be initialized when it is defined
- VLAs are multidimensional
- Address arithmetic is valid for the VLA
- VLAs make it easier to describe function headers that process arrays (there is no check for correct arguments)

```
void foo(int n, int m, int a[n][m]) // a[n][m] not mb at the beginning
```

```
void *alloca(size_t size);
```

Allocates a memory area of size on the stack. The memory area is freed automatically when the function that called alloca returns control to the caller. The behavior of the program is undefined in case of stack overflow.

```
void foo(int size) {
    ...
    while(b){
        char tmp[size];
        ...
    }
```

```
void foo(int size) {
    ...
    while(b){
        char* tmp = alloca(size);
        ...
    }
```

If you use VLA in the loop body, the array will be destroyed and recreated every iteration. If you use alloca() in the loop body, the memory for the array will **not be freed** after the iteration! This will happen only after exiting the foo() function.

33. Functions with variable number of parameters

- the idea of realizing such functions
- You can't do that!
- how such functions are implemented using the standard library

```
double avg(int n, ...)
{
    int *p_i = &n;
    double *p_d = (double *)(p_i + 1);
    double sum = 0..;

    if (!n)
        return 0;
    for (int i = 0; i < n; ++i, p_d++) sum
        += *p_d;
    return sum/n;
}
```

```
int main(void)
{
    double res = avg(4, 1.0, 2.0, 3.0, 4.0);
    ...
}
```

It is undesirable to use such an implementation, because the calling convention, although it regulates some things, such as the order of passing arguments (from right to left) or "extension" of some small types, but still leaves a lot of other things to the compiler developers.

The stdarg.h library is used to implement functions with a variable number of parameters

```
#include <stdarg.h>

double avg(int n, ...)
{
    va_list ap;
    double sum = 0, num;
    if (!n)
        return 0..;

    va_start(ap, n);

    for (int i = 0; i < n; ++i)
    {
        num = va_arg(ap, double);
        sum += num;
    }
    va_end(ap);
    return sum/n;
}
...
```

34. Preprocessor. General concepts. The #include directive. Simple macros. Predefined macros.

- What is a preprocessor
- What are the main actions it performs with the program
- Into which groups preprocessor directives can be divided
- Which rules are true for all directives

The **preprocessor** is the first utility a program encounters on its way to obtaining an executable file.

Actions:

1. Deleting comments
2. Enabling #include files
3. Handling conditional compilation directives and macro definitions (i.e. text substitution)

Preprocessor Directives:

1. Macro definitions #define, #undef
2. File inclusions #include
3. Conditional compilation #if, #ifdef, #else, #endif, etc.
4. The others are #pragma, #error, #line, etc.

General rules:

1. Directives begin with # and end with \n.
2. \ is used for multi-line directives
3. Lexemes can be separated by as many spaces as you like.
4. Can be defined anywhere in the program

Simple macros

Macro format #define replacement list identifier

```
#define PI 3.14
#define ABO "BA"
#define end }
```

If a macro is detected, the preprocessor performs a substitution list. Uses:

1. As names for numeric, character, and string constants;
2. Minor change in language syntax (better not to use it);
3. Type renames (better not to use);
4. Conditional compilation controls.

Predefined macros:

Useful for journaling, generating error messages with the name of file, line when debugging code

- **LINE**_____- current line number (decimal constant)
- **FILE**_____- compile file name

MB are used to provide compile time information

- **DATE**_____- compilation date
- **TIME**_____- compile time

These identifiers cannot be overridden or overridden by the undef directive.

- **func**_____- function name as a string (GCC only, C99 and not macro)

35. Preprocessor. Macros with parameters

- What is a preprocessor
- What are the main actions it performs with the program
- Into which groups preprocessor directives can be divided
- Which rules are true for all directives
- Story by topic
- Macros with parameters vs functions
- Macros with variable number of parameters
- Writing long macros
- Examples

p. 34

Macros vs functions

Advantages	Disadvantages
the program may run a little faster	The compiled code gets bigger
macros are "universal" `#define MAX(x,y) (x)>(y)?(x):(y)`	Argument types are not checked
	You cannot declare a pointer to a macro
	A macro can compute arguments more than once `n = MAX(i++,j);`

Macros with variable number of parameters

```
#define DBGPRINT(fmt, ...) printf(fmt, VA_ARGS )
```

Writing long macros

Method	Realization	+/-

1	`#define ECHO(s) {gets(s);puts(s);}`
	
	`if (echo_flag)`
	`  ECHO(s)`
	`else`
	`  gets(s);`
		Familiarity of syntax for macro description

		When viewing the program itself, it will appear that the rules of the language have not been followed
2	`#define ECHO(s) (gets(s),puts(s))`	There is no question about the staging of ; in the program

	ECHO(s);	Impossibility to use operators (if-else, for, while, ...)
3	#define ECHO(s) \ do \ { \ gets(s); \ puts(s); \ } while(0) \	There is no question about the staging of ; in the program
		The do-while operator is used only for grouping expressions

36. Preprocessor. General concepts. Conditional compilation directives. Directives #error and #pragma

- What is a preprocessor
- What are the main actions it performs with the program
- Into which groups preprocessor directives can be divided
- Which rules are true for all directives
- #ifdef vs #if (check file in the examples for the lecture)
- Story by topic
- Examples

p. 34

if	ifdef

#if value
// code that will be executed if value -- true
#elif value1
// code that will be executed if value1 -- true
#else
// code that will be executed otherwise
#endif
	#ifdef NAME_TOKEN
	// code that will be executed if NAME_TOKEN is defined
	#else
	// code that will be executed if NAME_TOKEN is not defined
	#endif

value - value (of expression) can be nested #elif directives	NAME_TOKEN - symbolic constant or macro defined via #define

#error "Error message"

Allows a message to be displayed in the compilation error list if a corresponding error occurs

```
#ifndef SIZE
#error "SIZE is not defined"
#endif
```

The #pragma directive allows you to get specific behavior from the compiler.

#pragma once is an alternative to include guard.

Additional advantages: it is shorter, and because the compiler itself is responsible for handling #pragma once and the programmer does not need to create new names (e.g., LIST_H), the risk of name collision is eliminated. However, it is not supported by all compilers, although it is widespread.

list.h	table.h	main.c

`#pragma once`
 	
`typedef struct`
`{`
`  int data;`
`  node_t *next;`
`} node_t;`
	`#include "list.h"`	`#include "list.h"`
		`#include "table.h"`

#pragma pack - change alignment of structures

```
// push -- сообщение компилятору сохранить текущее выравнивание
// 1 -- задаем нужное выравнивание
// в данном случае выравнивание отсутствует, поля располагаются вплотную
#pragma pack(push, 1)
struct s
{
    char c;
    int i;
    double d;
};
// восстанавливаем ранее сохраненное выравнивание
#pragma pack(pop)
```

37. Preprocessor. General concepts. Operations # and

- What is a preprocessor
- What are the main actions it performs with the program
- Into which groups preprocessor directives can be divided
- Which rules are true for all directives
- Story by topic
- Examples
- Peculiarities of the use of item

The # operation converts the macro argument to a string literal

```
#define PRINT_INT(x) printf(#x " = %d\n", x)

PRINT_INT(i/j); // printf("i/j" " = %d", i/j);
```

The ## operation combines two tokens into a single token

```
#define GENERAL_MAX(type)           \
type type###_max(type x, type y) \
{                                   \
---
```

Features:

1. Arguments are substituted into the replacement list already "expanded" unless # or ## operations are applied to them.
2. After all arguments have been "expanded" or the # or ## operations have been performed, the result is looked at again by the preprocessor. If the preprocessor result contains the name of the original macro, it is not replaced.

and # immediately turn what was passed into text and into code, even if it is the name of a macro. Thus, the following example will not work as we intend:

```
#include <stdio.h>

#define STR(x) #x

#define NAME Bob

int main(void)
{
    puts(STR(NAME));
    return 0;
}
```

The macro will unfold like this:

```
puts("NAME");
```

To get what we want, we need to remember the above properties, namely: arguments are inserted into the list already expanded if # or ## is not applied to them. I.e. we need to write a small wrapper so that already expanded macros will be in the list by the time the #x operation takes effect

```
#define STR_HELPER(x) #x
#define STR(x) STR_HELPER(x)
```

This will first be revealed as STR_HELPER(Bob) and then as "Bob".

Similarly for ##

```
#include <stdio.h>

#define AVG(x) (((max_##x) - (min_###x)) / (x##_count)))

#define TIME time

int main(void)
{
	double max_v = 5, min_v = 0, v_count = 10;
	double max_time = 10, min_time = 5, time_count = 100;

	printf("avg(v) %f\n", AVG(v));

	printf("avg(TIME) %f\n", AVG(TIME));

	return 0;
}
```

This will make incorrect expressions that have TIME instead of time in them.

```
#include <stdio.h>

#define GLUE_HELPER(x, y) x##y
#define GLUE(x, y) GLUE_HELPER(x, y)

#define AVG(x) (((GLUE(max_, x))) - (GLUE(min_,x))) / (GLUE(x,_count))))

#define TIME time

int main(void)
{
	double max_v = 5, min_v = 0, v_count = 10;
	double max_time = 10, min_time = 5, time_count = 100;

	printf("avg(v) %f\n", AVG(v));

	printf("avg(TIME) %f\n", AVG(TIME));

	return 0;
}
```

Now, this is where it's gonna work right

38. Embedded functions

- c99: inline
- Why did it appear
- Ways of describing embedded functions

inline - a request to the compiler to replace function calls by sequentially inserting the code of the function itself

In c99, `inline` means that the function definition is provided for substitution only and there must be another such definition of the same function somewhere in the program.

```
inline int add(int a, int b) { return a + b; }
```

Prior to the **C99** standard, if a function had a `static` memory class in the header file, it was replaced in the file by its body. This led to the problem of a large increase in the size of the executable file. Therefore, the **`inline`** keyword was added, which already advises the compiler to replace the function call with its body.

Ways of describing:

1. Use the keyword static
   ```
   static inline add(a, b) {return a + b;}
   ```
   Disadvantage: you need to duplicate the function throughout the program if you need to
2. Remove inline
   Disadvantage: The compiler may fail to perform the embedding
3. Add a non-inline definition somewhere in the program.
   Disadvantage: 2 different definitions possible, need to duplicate function.
4. C99: add an implementation file with this declaration
   ```
   extern inline int add(int a, int b);
   ```
   Disadvantage: **ta hz)))))**

39. Libraries. Static libraries.

WHAT IS A LIBRARY?

Library - a collection of subroutines and objects used for software development.

The library includes

– header file;

– compiled file of the library itself:

• libraries change rarely - no reason to recompile every time;

• binary code prevents access to the source code.

Libraries are divided into

– static;

– dynamic.

They are distributed as compiled object files combined into libraries. Libraries may include some functions (e.g., math functions) and/or data types (va_list in stdarg.h).

Each library should have its own header file, which should describe the prototypes (declarations) of all the functions contained in that library. With header files, you "tell" your program code what library functions there are and how to use them.

Static libraries are linked to the program at the time of linking. The library code is placed in an executable file.

	PROS AND CONS OF STATIC LIBRARIES
+	– The executable includes everything you need. – There is no problem with using the wrong version of the library.
-	– "Size" If multiple applications use the same library, the library code is present on the computer in multiple instances – When updating the library, the program must be rebuilt.

STATIC LIBRARY BUILDING

Same for Windows and GNU/Linux

compilation:

gcc -std=c99 -Wall -Werror -c name_lib.c

packaging:

ar rc libname.a name_lib.o

indexing (optional):

ranlib libname.a

Packing - getting an archive from object files.

c - create library file, if it does not exist (create) r - if there is already a file with library and OF, the functions will be be placed if the library contains older versions of them. (replace)

Indexing - If the library consists of a large number of object files and functions, it is recommended to create an index to speed up the linker's work. This is done using the ranlib utility, which modifies the libname.a library file itself

The functions of such libraries are not "formalized" in any special way (static library - archive of object files)

BUILDING APPLICATIONS USING STATIC LIBRARIES

gcc -std=c99 -Wall -Werror main.c libname.a -o app.exe

or

gcc -std=c99 -Wall -Werror main.c -L. -lname -o app.exe

-L - specifies the folders where the linker will look for libraries (analogous to I for headers during preprocessing) . (dot)- current directory

The source code of applications that use such libraries is also not "designed" in any special way

40. Libraries. Dynamic libraries. Dynamic layout

WHAT IS A LIBRARY?

Library - a collection of subroutines and objects used for software development.

The library includes

– header file;

– compiled file of the library itself:

• libraries change rarely - no reason to

recompile every time;

• binary code prevents access to the source code.

Libraries are divided into

– static;

– dynamic.

They are distributed as compiled object files combined into libraries. Libraries may include some functions (e.g., math functions) and/or data types (va_list in stdarg.h).

Each library should have its own header file, which should describe the prototypes (declarations) of all the functions contained in that library. With header files, you "tell" your program code what library functions there are and how to use them.

Dynamic linking is delegating part of the function of loading a library and searching for the required functions in that library to the linker

When using dynamic libraries, subroutines from the library are loaded into the application at runtime. The library code is not placed in the executable file.

	PROS AND CONS OF DYNAMIC LIBRARIES
+	– Multiple programs can "share" a single library. – Smaller application size (compared to static library application). – Upgrading the library does not require recompiling the program. – Can use programs in different languages.
-	– A library on a computer is required. – Library versioning (if the interface has changed)

DYNAMIC LIBRARY BUILDING

Windows	Linux
.dll - dynamic linked library	.so
-shared - builds exactly the dynamic library	
compilation: - gcc -std=c99 -Wall -Werror -c name_lib.c layout: gcc -shared name_lib.o -Wl, --subsystem,windows -o name.dll	compilation: - gcc -std=c99 -Wall -Werror -fPIC -c name_lib.c layout: gcc -o libname.so -shared name_lib.o export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.

DESIGN OF DYNAMIC LIBRARY FUNCTIONS

Windows	Linux
Function headers in .h and .c files are changed. Functions that are available from the library ___declspec(dllexport) Functions that are **not** available from the library - **unchecked** Specifies the calling convention, the default is. _____cdecl. We write it for safety	You don't have to

Example

```
// When building the DLL, define the NAME_EXPORTS macro
#ifdef NAME_EXPORTS
#define NAME_DLL declspec(dllexport)
#else
#define NAME_DLL declspec(dllimport)
#endif

#define NAME_DECL cdecl

NAME_DLL void NAME_DECL foo(type args);
```

DESIGN OF APPLICATION FUNCTIONS WHEN USING DYNAMIC LIBRARIES

Windows	Linux
The linker creates an import table for the functions to be taken from the libraries. The linker must realize that this function is taken from the library. The dllimport attribute is used for this purpose using the construct __declspec as its argument declspec(dllimport)))	You don't have to

PROS AND CONS OF APPLICATION LAYOUT METHODS

	Dynamic loading	Dynamic layout
+	-Functions are used on an as-needed basis	- Linux: no need to modify the library code - The library is connected automatically
-	- Manual control of loading/unloading functions - Changing the program code	- Windows: you have to make changes to the library code - Not all features may be needed

BUILDING APPLICATIONS WITH DYNAMIC LINKING:

gcc -std=c99 -Wall -Werror -c main.c

gcc main.o -L. -larr -o test.exe

41. Libraries. Dynamic Libraries. Dynamic loading

WHAT IS A LIBRARY?

Library - a collection of subroutines and objects used for software development.

The library includes

– header file;

– compiled file of the library itself:

• libraries change rarely - no reason to recompile every time;

• binary code prevents access to the source code.

Libraries are divided into

– static;

– dynamic.

They are distributed as compiled object files combined into libraries. Libraries may include some functions (e.g., math functions) and/or data types (va_list in stdarg.h).

Each library should have its own header file, which should describe the prototypes (declarations) of all the functions contained in that library. With header files, you "tell" your program code what library functions there are and how to use them.

Dynamic loading - search and loading of dynamic library functions independently, without the involvement of the linker, using the interface provided by the OS.

Windows	Linux
windows.h	dlfcn.h
HMODULE LoadLibrary(LPCSTR); - returns library descriptor	void* dlopen(const char *file, int mode);

FARPROC GetProcAddress(HMODULE, LPCSTR); - get function address
 	
FreeLibrary(HMODULE);
	void* dlsym(void *restrict handle, const char *restrict name);
	
	int dlclose(void *handle);

When using dynamic libraries, subroutines from the library are loaded into the application at runtime. The library code is not placed in the executable file.

	PROS AND CONS OF DYNAMIC LIBRARIES
+	– Multiple programs can "share" a single library. – Smaller application size (compared to static library application). – Upgrading the library does not require recompiling the program. – Can use programs in different languages.
-	– A library on a computer is required. – Library versioning (if the interface has changed)

BUILDING DYNAMIC LIBRARIES (DYNAMIC LINKING)

Windows	Linux
.dll - dynamic linked library	.so
-shared - builds exactly the dynamic library	
compilation: -gcc -std=c99 -Wall -Werror -c name_lib.c layout: gcc -shared name_lib.o -Wl, --subsystem,windows -o name.dll	compilation: -gcc -std=c99 -Wall -Werror -fPIC -c name_lib.c layout: gcc -o libname.so -shared name_lib.o export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.

DESIGN OF DYNAMIC LIBRARY FUNCTIONS

Windows	Linux
Function headers in .h and .c files are changed. Functions that are available from the library ____declspec(dllexport) Functions that are **not** available from the library - **unchecked** Specifies the calling convention, the default is. _cdecl. We write it for safety	You don't have to

Example

```
// When building the DLL, define the NAME_EXPORTS macro
#ifdef NAME_EXPORTS
#define NAME_DLL declspec(dllexport)
#else
#define NAME_DLL declspec(dllimport)
#endif

#define NAME_DECL cdecl

NAME_DLL void NAME_DECL foo(type args);
```

DESIGN OF APPLICATION FUNCTIONS WHEN USING DYNAMIC LIBRARIES

Windows	Linux
The linker creates an import table for the functions to be taken from the libraries. The linker must realize that this function is taken from the library. The dllimport attribute is used for this purpose using the construct __declspec as its argument _____declspec(dllimport)))	You don't have to

PROS AND CONS OF APPLICATION LAYOUT METHODS

	Dynamic loading	Dynamic layout
+	-Functions are used on an as-needed basis	-Linux : no need to modify the library code

		-The library is automatically connected
-	- Manual control of loading/unloading functions - Changing the program code	- Windows: you have to make changes to the library code - Not all features may be needed

BUILDING APPLICATIONS AT DYNAMIC LOADING:

gcc -std=c99 -Wall -Werror main.c -o test.exe -ldl

42. Libraries. Dynamic library in C. Python application

- What is a library
- In what form are disseminated and why
- Types of libraries their +/-
- Layout is different in windows and linux
- Story by topic

For everything except the story on the subject see id. at 41.

To load the library into a Python program, you need to create a class object CDLL:

```
import ctypes
lib = ctypes.CDLL('example.dll')
```

There are several classes for working with libraries in the ctypes module:

- CDLL (cdecl and return value int);
- OleDLL (stdcall and return value HRESULT);
- WinDLL (stdcall and return value int).

The class is chosen depending on the calling convention that the library uses.

Once the library is loaded, you must describe the library function headers using the notation and types known to Python.

```
# int add(int, int)
add = lib.add
add.argtypes = (ctypes.c_int, ctypes.c_int)
add.restype = ctypes.c_int
```

In order for the Python interpreter to properly convert arguments, call
add function and return the result of its work, it is necessary to specify the argtypes and restype.

Integers are "immutable" objects in Python. Trying to change them will cause exception. Therefore, for arguments that "use" a pointer, you must use the compatible types described in the ctypes module to create an object and pass exactly that object.

```
def divide(x, y):
  rem = ctypes.c_int()
  quot = _divide(x, y, rem)
  return quot, rem.value
```

The avg function expects to get a pointer to an array. We need to understand which Python data type will be used (list, tuple, etc.) and how it is converted to an array.

```
# void avg(double*, int)
_avg = lib.avg
_avg.argtypes = (ctypes.POINTER(ctypes.c_double), \
                                          ctypes.c_int)
_avg.restype = ctypes.c_double

def avg(nums):
  src_len = len(nums)
  src = (ctypes.c_double * src_len)(*nums)
  return _avg(src, src_len)
```

ctypes: bottom line:

- The main problem of using this module with large libraries is writing a large number of signatures for functions and, depending on the complexity of functions, function-wrappers.
- You need to have a detailed understanding of the inner workings of Python types and how they can be converted to C types.
- Alternative approaches are to use Swig or Cython.

Usually the functions of an expansion module have the following form

```
static PyObject* py_func(PyObject* self, PyObject* args)

{

    ...

}
```

- PyObject is a C data type that represents any Python object.

- The extension module function receives a tuple of such objects (args) and returns a new Python object as the result.
- The self argument is not used in simple functions.
- The PyArg_ParseTuple function is used to convert variables from the Python representation to the C representation.
- As input, this function takes a format string that describes the type of value required, and the addresses of the variables into which it will be placed values.
- During conversion, the PyArg_ParseTuple function performs various checks. If something went wrong, the function returns NULL.

```
int a, b;
if (!PyArg_ParseTuple(args, "ii", &a, &b))
    return NULL;
```

- The Py_BuildValue function is used to create Python objects from C data types. This function also receives a format string describing the desired type.

```
int a, b, c;
if (!PyArg_ParseTuple(args, "ii", &a, &b))
   return NULL;
c = add(a, b);
return Py_BuildValue("i", c);
```

- Near the end of the extension module are the PyMethodDef module method table and the PyModuleDef structure, which describes the module as a whole.
- The PyMethodDef table lists the following
  - C functions;
  - names used in Python;
  - flags used when calling the function,
  - lines of documentation.

- The PyModuleDef structure is used to load a module.
- At the very end of the module is the module initialization function, which is almost always the same except for its name.

The Python script setup.py is used to compile the module. The compilation is performed using the command:

```
python setup.py build_ext --inplace
```

43. Uncertain behavior

Definition:
As a result of calculating the value of an expression, not only new data is generated, but there is also a change in the state of the execution environment.

Species:

- Data Modification.
- Addressing variables declared as volatile.
- Calls a system function that produces side effects (such as file input or output).
- Call functions that perform any of the above actions.

Expressions in SI:

Expressions will be evaluated in almost the same order as they appear in the source code: top to bottom and left to right.

However, the compiler may perform actions in a different sequence for optimization purposes, except in cases where the order of computation is explicitly guaranteed

An example of indeterminate behavior:

```
i = 1;
x[i] = i++ + 1;

// Method 1 (right to left)
(i++ + 1) => 2, i = 2, x[2] = 2

// Method 2 (from left to right)
x[1] = (i++ + 1) = 2, i = 2
```

Waypoints:

Порядок определен у :

- ●Выражений с &&, || .
- ●У выражений в которых используется “,”
- ●В тернарных операторах

Операция	Название	Нотация	Класс	Приоритет	Ассоциат.
!	Не	!x	Префиксные	15	Справа налево
&&	И	x && y	Инфиксные	5	Слева направо
||	Или	x || y		4	Слева направо

при && и || следовательно слева направо!!!

A sequence point is a point in a program at which the programmer knows which expressions (or subexpressions) have already been evaluated and which expressions (or subexpressions) have not yet been evaluated.

Types of waypoints in C99:

- Between the computation of the left and right operands in the operations &&, || and "," .

```
*p++ != 0 && *q++ != 0
```

- Between the computation of the first and second or third operands in a ternary operation.

```
a = (*p++) ? (*p++) : 0;
```

- At the end of the full expression.

```
a = b; if
() switch
() while
()
do{} while()
for ( x; y; z) return
x
```

- Before entering the called function.

  -The order in which the arguments are evaluated is undefined, but this sequence point ensures that all of its side effects are manifested at the time of entry into the function.

- In a declaration with initialization at the time the initializing value computation is completed.

```
int a = (1 + i++);
```

Types of uncertain behavior:

- **Unspecified behavior.**

The standard offers several options to choose from. The compiler can implement any variant. In this case, a correct program is supplied to the compiler input.

For example: all function arguments must be evaluated before the function call, but they can be evaluated in any order.

- **Implementation-defined behavior.**

It looks like unspecified behavior, but the compiler documentation should specify which behavior is implemented.

For example: the result x % y, where x and y are integers and y is negative, can be either positive or negative.

- **Undefined behavior**

This behavior occurs as a consequence of an incorrectly written program or incorrect data. The standard does not guarantee anything, anything can happen.

Why indeterminate behavior is in SI:

- free compiler developers from the necessity to detect errors that are difficult to diagnose.
- avoid favoring one implementation strategy over another.
- mark language areas for language extension (language extension).

Ways to combat it:

- Include all compiler warnings, read them carefully.
- Use the compiler features (-ftrapv).
- Use multiple compilers.
- Use static code analyzers (for example, clang).
- Use tools such as valgrind, Doctor Memory, etc.
- Use assertions.

• -ftrapv • This option generates traps for signed overflow on addition, subtraction, multiplication operations. • This option generates traps for sign overflow in addition, subtraction, multiplication operations.

Examples of uncertain behavior:

- Use of uninitialized variables.
- Overflow of signed integer types.
- Going beyond the array boundaries.
- The use of "wild" pointers.
- ...

See Appendix J standard (pages 490 to 502)

An example of implementation-dependent behavior:

The result x % y, where x and y are integers and y is negative, can be either positive or negative. The size of int can be from 2 to 4 bytes, depending on the implementation or compiler settings.

An example of nonspecific behavior:

All function arguments must be evaluated before the function call, but they can be evaluated in any order.

Rounding result when the value is out of range

Order of two elements compared as equal in an array sorted by qsort function (unstable sorting)

The most dangerous type of undefined behavior is Undefined behavior

44. ATD. The concept of module. Varieties of modules. Abstract object. Stack of integers

A program can be considered as a set of independent modules. There are two parts of it: interface and implementation.
An interface describes the capabilities of a module. Implementation describes how the module does what the interface suggests.
Four varieties of module are distinguished:

1. Abstract data type;
2. Abstract Object;
3. Dataset;
4. Library.

An abstract data type is an interface that defines a data type and operations on that type. It is abstract because the interface hides the implementation details of the type.
An abstract object is a set of functions that handle hidden data.

stack.h	stack.c
... #include <stdbool.h>	#include <stdlib.h> #include <assert.h> #include "stack.h"

void make_empty(void);
bool is_empty(void);
bool is_full(void);
int push(int i);
int pop(int *i);
	struct stack_type
	{
	int* content;
	size_t top;
	size_t size;
	
	};
	
	struct stack_type stack;
	
	... Implementing functions with a global stack variable.....

45. ATD. The concept of module. Varieties of modules. Abstract data type. Stack of integers

see previous paragraph

stack.h	stack.c

…
typedef struct stack_type* stack_t;
 	
stack_t create(void);
void destroy(stack_t s);
void make_empty(stack_t s);
bool is_empty(const stack_t s);
bool is_full(const stack_t s);
int push(stack_t s, int i);
int pop(stack_t s, int *i);
	#include <stdlib.h>
	#include <assert.h>
	#include "stack.h"
	
	struct stack_type
	{
	int* content;
	size_t top;
	size_t size;
	
	};
	
	... Realization of functions with function parameters....

46. Linux kernel lists. Idea. Highlights of the usage.

- The idea of implementing a universal list in the Linux kernel
- How the head of the list is described
- Addition
- Detour
- Deletion
- Liberation

Списки Беркли: идея

Список Беркли – это циклический двусвязный список, в основе которого лежит следующая структура:

```
struct list_head
{
    struct list_head *next, *prev;
};
```

В отличие от обычных списков, где данные содержатся в элементах списка, структура list_head должна быть частью сами данных

```
struct data
{
    int i;
    struct list_head list;
    ...
};
```


Списки Беркли: описание

```
#include "list.h"

struct data
{
    int num;
    struct list_head list;
};
```

Следует отметить следующее:

- Структуру struct list_head можно поместить в любом месте в определении структуры.
- struct list_head может иметь любое имя.
- В структуре может быть несколько полей типа struct list_head.

The Berkeley list is a cyclic doubly-linked list based on this structure:

```
struct list_head
{
    struct list_head *next, *prev;
};
```

list_head is itself part of the data.

MB anywhere in the SD definition. Can have any name

There can be several fields of the struct list_head type in an SD

Ways of describing:

1.
```
LIST_HEAD(num_list); // pass the name of the variable we want to create

#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) \
    struct list_head *name = LIST_HEAD_INIT(name)
```
2.
```
struct data num_list;
INIT_LIST_HEAD(&num_list.list);

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

Addendum:

```
struct data *item;
LIST_HEAD(num_list);

item = malloc(sizeof(*item))
if (!item)
    ...

*item = 5;
INIT_LIST_HEAD(&item->list);
list_add(&item->list, &num_list); // add to the beginning
list_add_tail(&item->list, &num_list); // add to end
```

Detour:

1. The usual workaround, within which data will have to be retrieved using list_entry
```
struct list_head *iter;
```

```
list_for_each(iter, &num_list)
{
    // list_entry(element, structure itself, field name)
    item = list_entry(iter, struct data, list)
    ...
}

list_for_each_prev(iter, list_head) // in the reverse direction
```

2. A normal traversal, within which you will not need to use list_entry

```
list_for_each_entry(item, &num_list, list)
{
    ...
}

list_for_each_entry_prev()
```

3. Bypass with copying

```
list_for_each_safe(iter, iter_safe, &num_list)
{
    ...
}
```

Deleting an item:

```
tmp = list_entry(num_list.next, struct data, list);
list_del(num_list.next);
free(tmp);
```

Releasing the list

```
list_for_each_safe(iter, iter_safe, &num_list)
{
    item = list_entry(iter, struct data, list);
    list_del(iter);
    free(item);
}
```

47. Linux kernel lists. Idea. The main points of realization.

- The idea of implementing a universal list in the Linux kernel
- We focus on the container_of and offsetof macros

The Berkeley list is a cyclic doubly-linked list based on this structure:

```
struct list_head
```

```
{
    struct list_head *next, *prev;
};
```

list_head is itself part of the data.

MB anywhere in the SD definition. Can have any name

There can be several fields of the struct list_head type in an SD

The idea of offsetof

The idea is to find out the offset of a given field. That is, we assume that the address of the structure is located at address zero. Then, the distance to the field of the given structure is easily calculated by converting it to an integer:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
int offset = (int) (&((struct s*)0)->i))
```

```
((struct s*)0) // this line tells the compiler that there is a structure at address 0 and we get a pointer to it.

((struct s*)0)->i // compiler thinks this field is located at address 0 + offset i

&((struct s*)0)->i // calculate offset i in structure s

(size_t) &((struct s*)0)->i
```

The idea of container_of

This macro is used to get a pointer to a structure by a given field of that structure. It helps to access data in, for example, Berkeley lists:

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)

#define container_of(ptr, type, field_name) (\
    (type *) ((char *)(ptr) - offsetof(type, field_name)))
```