

# C

## Summer course 2009 slide-guide

Short notes about the lecture slides, by Nerya Or.

Lectures+slides were given by Moti Freiman and Ofir Pele.

In this document are some of the things which in my opinion are important in the lecture slides. (+Extra tips in the end).

The idea is that this document is a sort of “table of contents” for the actual slides, so it’s best to print the slides as well.

Some of the contents may be inaccurate/wrong, so please don’t blame me if something goes bad...

This document is not officially endorsed by the course staff, use at your own responsibility.

Good luck in the exam!

Version 1.01

### 1 Class/Tirgul 1

#### 1.1 Class 1

-Nothing of particular interest...

#### 1.2 Tirgul 1

Pg. 2:

`sizeof(char)==1`, by definition.

Everything else is machine-dependant and can’t be assumed.

Pg. 3:

Short-circuit `&&`, `||` are used.

For example, “`if(i==1 && x.isValid())...`” - the second part might not be evaluated in case the first is ‘true’.

## 2 Class/Tirgul 2

### 2.1 Class 2

Slide 25:

It's impossible to do the following, since an array is basically a (const) pointer:

```
int a[4];  
  
int b[4];  
  
a = b; //illegal.  
  
if(a==b) {...} //illegal (Actually it compares pointers - addresses).
```

Slide 26:

Various ways to initialize an array...

Slide 35:

“Arrays are essentially constant pointers” - demonstration of pointer vs. array syntax, difference.

Slide 36: Important -

```
int *p;  
int a[4];  
  
sizeof(p)==sizeof(void*) , and sizeof(a)==4*sizeof(int).
```

However, if we would **pass an array into a function** (thus going out of its original declaration scope), we would get that `sizeof(a)==sizeof(void*)` as well, since it's passed as a pointer to `a[0]`. (not mentioned in this slide...)

**The exact definition:** `sizeof` within the scope of the array declaration returns the actual number of bytes requires to store the entire array (e.g. number of items in the array, multiply by the required size of each item), but when passed as an argument, it will give the size of a pointer.

### 2.2 Tirgul 2

Slide 6: **The rule of what const protects -**

Const protects his left side, **unless** there is nothing to his left, and **only then** it protects his right side.

Slide 9: Example of what const protects (content/pointer):

```
int arr[] = {1,2,3};  
int const * p = arr;  
p[1] = 1; //illegal!  
*(p+1) = 1; //illegal!  
p = NULL; //ok
```

and also:

```
int arr[] = {1,2,3};
int * const const_p = arr;
const_p[1] = 0; //ok
const_p = NULL /illegal!
```

Slide 22-24: String literals

String literals are in the code-part of the memory, so they can't be changed!

```
char* msg = "text";
```

```
msg[0] = 'w'; //seg fault!!!
```

It's a good idea to make it const, to protect us: `char const * msg = "text";`

Note the difference:

```
char* msg = "text" →msg is a pointer that points to memory in the code part.
```

```
char msg2[] = "text" →msg2 is an array of chars that are on the stack. (And can be modified).
```

## 3 Class/Tirgul 3

### 3.1 Class 3

Slides 11-18: Arrays in struct copying

We note that if we have an array inside a struct (declared for example: `"double _arr[SIZE]"`), then when C copies that struct by-value, the entire array is copied completely. (Not as a pointer, or anything).

But!!!

If we had a pointer in a struct, then only the pointer (==the address it points at) is copied to the other array, and this can cause some problems...

Possible solution: Write a 'clone' function that copies data from one struct pointer to another.

Slides 20 - 21: Arrays and structs as arguments

- When an array is passed as an argument to a function, the address of the 1st element is passed.
- Structs are passed by-value, exactly as the basic types.

Slides 24 - 25: A bit about bit fields

"~" is the bitwise-complement operator. It turns  $1 \leftrightarrow 0$ . There's also bitwise `|`, `&`, `^` (OR,AND,XOR).

Bit field syntax is on slide 24.

Slide 28: **Extern**

'**extern**' tells the linker to use a global variable from another module. Note that if that variable has been declared as '**static**' in the other module, then we **cannot** import it using '**extern**', since '**static**' makes a variable be local to a module.

Slide 29: Static functions

These functions are local to a specific module, and cannot be used outside of it.

Static variables are also like this, and they hold their value and never 'lose' it.

Slide 35+:

Makefile info (also in Tirgul 3, slides 9-19).

## 3.2 Tirgul 3

Slide 2: argv,argc

Note that it's OK to write "`char* argv[]`" and "`char** argv`", but **not** "`char argv[] []`"!  
(Apparently because we must always specify all-but-the-leftmost sizes of things in [] notation(?)).

Slides 4-7:

File I/O.

Slides 20 - 22:

`goto`

Slides 23+:

`union`. It keeps one of several types in the same position in memory.

The size of a union is the maximum out of all of its members.

(For example a union containing double and int will take up 8 bytes, assuming `sizeof(int)=4`, `sizeof(double)=8`).

It works by storing only the last element we've set, and the other elements in the union are simply ways to cast the contents of the union. (Similar to `reinterpret_cast` in C++).

## 4 Class/Tirgul 4

### 4.1 Class 4

Slides 1 - 15:

General things about memory (stack/static heap).

Static heap == globals area. Changes during runtime, but must be defined at compile time.

Slides 16 - 20:

`malloc`. Usage and explanations. (Recommended: Create a "factory" function to return an initialized pointer to an allocated variable/struct).

Slides 26 - 32:

`free()`.

Note: There's no error checking for the `free()` function!

The behavior for "`free(someRandomPtr)`" is undefined (when we're talking about a pointer that doesn't point at `malloc`'d memory).

Another note: `free(NULL)` does nothing, so it's safe.

Yet another note: `free` doesn't change the given pointer, so it's a good idea to set any `free`'d pointer to `NULL` afterwards.

Also, basic structs are `free`'d properly with a single "`free(structObj)`" command.

If a struct contains any pointer to allocated heap memory that we wish to dispose of, we should create something like a destructor, that frees the heap memory, and then frees the rest of the struct.

## 4.2 Tirlgul 4

Slides 3 - 5: Common bugs to avoid:

**Bug 1:** Allocating heap memory inside a struct, and then using `free()` on the struct, and not freeing the heap memory datamember. Solution: Manually free the heap space, or just create a destructor function.

**Bug 2:** Returning the address (or a reference, in C++!) for a local stack variable of a function. What happens is that the stack variable's lifetime ends as the stack folds down, and we'll get unexpected results.

**Bug 3:** Calling `free()` on something that wasn't allocated by `malloc`. (In the example, it's possible that we didn't call `malloc` if the "if" statement failed).

## 5 Class/Tirlgul 5

### 5.1 Class 5

Slides 8-12: Multi-dimensional arrays.

When sending a multi-dimensional array to a function, only the 1st index can be omitted:

```
void func(int x[5][7]) //ok
void func(int x[][7]) //ok
void func(int x[][]) //error
void func(int* x[]) //error
void func(int** x) //error
```

Also, it's possible to create dynamic sized multi dimensional arrays. See slides 9,10.

From slide 11: Memory in dynamically allocated arrays is **not** continuous (it might be, for each "row" allocated, but that's also the OS's decision when allocating our memory).

Working with (statically allocated) multi dimensional arrays:

Method 1:

If we have an array with R rows and C columns: `int A[R][C];`

then the access to its elements is calculated as follows:

Access to `A[0][0]` is simply at `A`.

Access to `A[i][0]` (i'th row) is at `A+(i*C)`.

Access to `A[i][j]` (i'th row, j'th column) is at `A+(i*C)+j`.

Note: to actually make it work, we'll have to cast `A` to `int*` from `int**`, like so:

```
int A[2][2] = { {1,2},{3,4} };
int i=1, j=1;
int* p = (int*)A+(i*C)+j; //The [i][j] element.
printf("A[%d][%d] = %d\n",i,j,*p); //Prints: "A[1][1] = 4"
```

Method 2:

We can store our array in a single, long array.

So for an array with R rows and C columns: `int A[R*C];`

Access to `A[i][j]` is via `A[i*C + j]`.

This method is considered more efficient.

Slides 16 - 22:

A bit about **pointers to functions**, and some strange example about the peculiarly named MAP-CAR (??).

(More about pointers to functions in tirlgul 5, slides 8+).

Slides 23 -24:

`qsort`, from the standard library.

Slide 32: (Quoting just because I find it hilarious):

“Don’t attribute bugs to mysterious forces”.

## 5.2 Tirlgul 5

Slide 5-6:

A `void*` pointer cannot be dereferenced, without explicit casting.

Slides 8+:

Pointers to functions. Also, some info about `qsort`.

# 6 Class/Tirlgul 6

## 6.1 Class 6

Slides 3 - 10:

The difference between static libraries (“inserted” into the executable in compilation/linkage), and shared libraries (“helper” files that are used in runtime: dll files in Windows (like DirectX, for example), or .so files in Linux).

Slides 20 - 22:

`errno`, and `perror()`.

Slides 25 - 29:

**Pointers to arrays.** (Important confusing stuff).

There’s a difference between “`char (*arr)[5];`” and “`char *arr[5];`” !!

–Pay close attention to the syntax difference described in slides 27,28.

(Another example - tirlgul 6, slide 9).

Working with pointers to arrays:

From labc2008b, question 1 - the following code will print “7”:

```
#define COLS 3
#define ROWS 3
int GetNthItem (int arr[ROWS][COLS], int n) {
    return *((int*)(arr+n));
}
int main () {
    int arr[ROWS][COLS] = {{1,2,3},{4,5,6},{7,8,9}};
    printf ("%d\n", GetNthItem(arr,2));
    return 0;
}
```

The reason for this is as follows:

The actual type that was sent to `GetNthItem` is `int (*arr)[COLS]`, which is a **pointer to an array** of `int`, of length `COLS`.

Now, of course, given this 'arr' as we passed it into `GetNthItem`,

we have `sizeof(arr)=sizeof(void*)`, but `sizeof(*arr) = COLS*sizeof(int)`.

So by pointer arithmetic, "`arr+n`" actually equals (in single bytes!) to "`arr+n*sizeof(*arr)`" (in single bytes, of course), which is equal to "`arr+n*COLS*sizeof(int)`" (also in single bytes).

(This is the same thing that goes on in slides 25 - 28).

The fix we should do if we want `GetNthItem` to return the Nth item in the array, is to change the return value of the function to `"return *((int*)arr+n);"`.

This will cast `arr` to `int*`, and then the pointer arithmetic advances by units of `sizeof(int)`, which is good for what we need.

Slides 30 - 32:

**A function that returns a pointer to a function.** (+variations with typedef).

## 6.2 Tirlgul 6

Slide 9:

Another example of the "pointer to array" syntax, mentioned in class 6, slides 25-29.

Slide 12:

**Explanation of why we need to specify all but the leftmost number of cells, in a multi-dimensional array.**

Slide 17:

`register.`

Slides 18 - 21:

`volatile.`

## 7 Other notes

Note 1:

The following code is illegal:

```
char str[10] = "hello";
foo(&str);
```

This is because an array doesn't have an address, since it's basically a const pointer. So "`&str`" is meaningless.

Note 2:

```
char str[2] = "hi"; //Very bad!!! Missing room for '\0' in the end.
char str[3] = "hi"; //OK
```

Note 3:

From labc2005b, question 1: (Tricky pointer arithmetic, in my opinion).

```
int main () {
    int arr[] = {0,1,2,3,4,5};
    int * p = arr;
    p = *(p+4) + p; //Note that *(p+4) = 4
    *p += 1;
    printf("%d %d %d %d %d %d\n",arr[0],arr[1], arr[2], arr[3], arr[4], arr[5]);
    return 0;
}
```

Result will be:

0 1 2 3 5 5