NTNU | Norwegian University of Science and Technology

C "crash course"

TDT4186 – Operating Systems
TDT4205 – Compiler Construction

Spring semester 2021

Michael Engel

C Crash Course

- Idea:
  - Set a common knowledge foundation for all students
  - Show important differences between Java and C programming and/or refresh your knowledge of C programming
- Why do we use C?
  - Provides abstractions **of the machine** on a high level, not abstractions of your problems (but see [1])
    - Useful for us, we want to build our own abstractions
- What's the problem with C?
  - Compiler construction revolved around string processing (scanning) and data structures (lists, trees, hash tables) – things not easy to use or unavailable in plain C

[1] David Chisnall, "C is not a low-level language", ACM Queue Volume 16, issue 2 (2018) https://queue.acm.org/detail.cfm?id=3212479

What you should be familiar with

- Using a plain text editor
- Storing the editor file on a system running some sort of Unix
  - this includes WSL on Windows 10 (the “Windows Subsystem for Linux” [2]), and Mac OS X and, of course, Linux, *BSD, …
- Logging in to some Unix-like system and use a standard Unix shell such as bash
- Creating, editing and using makefiles and make

- This is easy to learn
  - but maybe feels quite strange for students coming from Windows/ Java and only used to IDEs and automatic project management

[2] https://docs.microsoft.com/en-us/windows/wsl/install-win10

Getting started

- If you don't have a convenient system handy, you can use the system at **login.stud.ntnu.no** (I haven’t tried this myself so far…)
- You can have SSH shells from windows, tiny program download from http://www.putty.org/
- You can transfer files through SAMBA (“map network drive”), or edit them directly through the shell ('nano' is a pretty humane screen-editor available on login.stud, documentation at http://www.nano-editor.org/)
- None of this is particularly hard, but it isn't perfectly intuitive to everyone the first time
- If you can't find your way, ask. Installing a 100 megabytes of colorful buttons will not solve the problem

  If you **can** find your way, feel free to use whatever IDE you know and love, but don't rely on it being there

Programming Paradigms

- Imperative programming
  - Program = sequence of commands
- Procedural programming
  - Special case of imperative programming
  - Program = set of procedures (functions) operating on common data
- Object oriented (OO) programming
  - Encapsulation of code and data in objects
  - Program: set of object interacting via interfaces

  → usually: “OO language” = imperative programming + OO extension

- *Very* different: declarative programming (functional, rule oriented...)

Programming Paradigms (2)

- A programming language can be suitable for a given paradigm.
- However, the language does not *enforce* the use of this paradigm or *preclude* the use of other paradigms
- Examples
  - Procedural programming in Java (*god object, big hairy object*)
  - Non-procedural imperative programming in C (*god function*)
  - Object oriented programming in C

Java vs C

- hello_world.java

```
class Hello {
    public static void main(String argv[]) {
        System.out.println("Hello world!");
    }
};
```

- hello_world.c

```
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

- `printf()` is not part of the language, rather of the *standard library*
- `main()` is not part of a class (since C does not support classes)
- `main()` requires a return parameter: program exit code

Structure of C programs

```
#include <stdio.h>

int counter;

int gcd(int a, int b) {
    counter++;
    if (a==0) return b;
    if (b==0) return a;
    return gcd(b,a%b);
}

int main(int argc, char **argv) {
    int eastwood=10164;
    char ly=240;
    printf("result: %d\n", gcd(ly,eastwood) );
    printf("function calls needed: %d\n", counter);
    return 0;
}
```

global variable definitions and functions

local variables at the beginning of a function or block

- main() function
  - Entrance point for C programs, can be passed parameters
  - Return value gives exit code of the program (in Unix shell: “$?”)

Output using printf

```c
#include <stdio.h>

int main() {
    int eastwood = 4711;
    printf(“A number: %d\nNow in hexadecimal: %x”, -815, eastwood);
    return 0;
}
```

- Make printf „known“ to the compiler:
  #include <stdio.h>
- First parameter: format string:
  “A number: %d\nNow in hexadecimal: %x“
- Contains placeholders for additional parameters
  - decimal (signed): %d (unsigned: %u)
  - hexadecimal: %x
  - many more listed in the printf man page

Functions

- “Classless methods”
- Elementary building blocks which enable *modularized* imperative programs
  - Functions reduce the complexity by partitioning complex problems into manageable parts
  - Reusable program components
  - Hiding of implementation details
- Functions vs. methods
  - Functions are declared and defined in the global scope
  - ...are not part of a class
  - ...do not provide this

Function Declarations and Definitions

- Functions should be *declared* before they are used (called):

```
void bar(int);              /* declaration */

void foo(int b) {
    if (b<0) return;
    bar(b-1);
}

void bar(int a) {            /* definition */
    if (a<0) return;
    foo(a-1);
}
```

- *Forward declaration* tells the compiler that bar exists when it compiles foo
  - Otherwise, the compiler assumes that bar‘s return value is of type int (*implicit declaration*) and disables type checking of parameters
    → bad style, causes compiler warning
  - Historical background: one-pass compilers

Function – Swapping Variable Contents

- Java
  - simple data types: call by value
  - Object types: call by reference

- C
  - (technically) only call by value
  - (call by reference is possible by using pointers)

What is the output of this program?

```
#include <stdio.h>

foo(int a) {
    a++;
}

int main() {
    int a=5;
    foo(a);
    printf("%d",a);
    return 0;
}
```

Control Structures

- In C: same as in in Java
- `if (condition) {...} else {...}`
- `while (condition) {...}`
- `do {...} while (condition);`
- `for(...; condition;...) {...}`
- `switch (...) {case ...: ...}`
- `continue; break;`
- Only difference :
  condition is integer number (not a boolean)

Standard Types

- Simple data types similar to Java
  - `char` character (ASCII code), 8 bits
  - `int` integer number, 32 bits *
  - `float` floating point number (32 bits)
  - `double` double precision floating point number (64 bits)
  - `void` without a value
- Additional *modifiers*:
  - `signed`, `unsigned`, `short` and `long`
- Type `boolean` did not exist in old C (only in C99: `bool`)
  - Boolean expressions evaluate to 0 (false) or 1 (true)
  - Integers can be used in place of Boolean variables

```
print("%d", 4711 > 42 ); /* prints 1 */
while (1) {}             /* endless loop */
```

* depends on the architecture (length of a machine word)!

Structures (structs)

- There are no classes in C
- However, there are *complex* data types (structs)
- “classes without methods”

```
struct student {
    int student_id;
    int age;
    char name[64];
};

void rejuvenate(struct student
s) {
   s.age = 0;
}

void foo() {
    struct student s1;
    s1.age = 20;
    rejuvenate(s1);
}
```

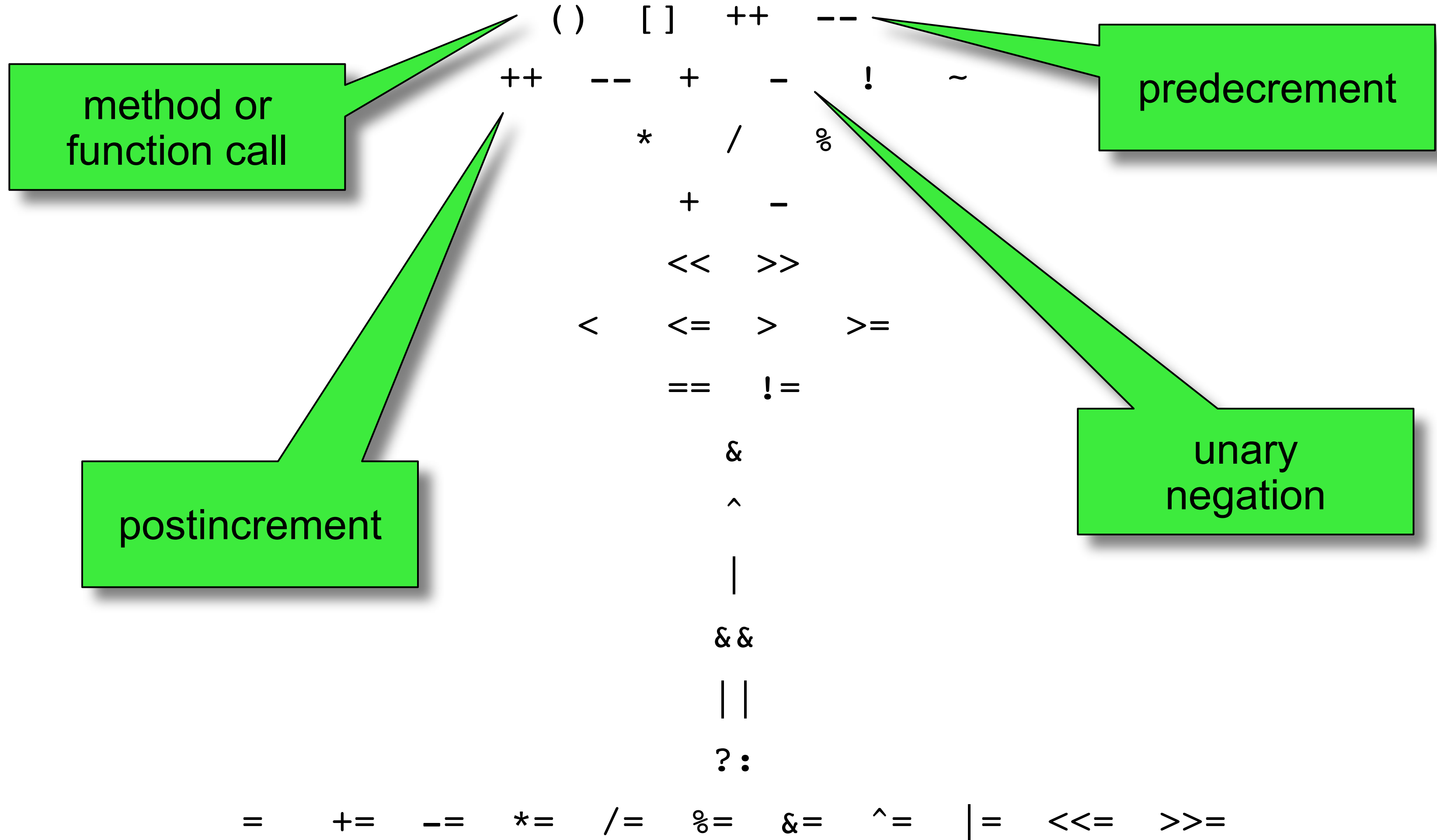
Why does this instruction not cause an exception ?

What is the age of s1 here?

- struct parameters are also passed by value!

Operators (1)

- Mostly identical in C and Java



Operators (2)

- There are a number of differences

```
                 .   ()   []   ++   --   ->
        (t)   ++   --   +   -   !   ~   *   &   sizeof
                        *   /   %
                          +   -
                          <<   >>
                       <   <=   >   >=
                          ==   !=
                             &
                             ^
                             |
                             &&
                             ||
                             ?:
   =   +=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
```

also in Java, but with different semantics!

only in C

Operators (3)

- Access to members of an object
  - Access to method or member variable of an objects in Java
  - C does not provide objects with related methods
- `(t)` cast to type *t*
  - Automatically checked for validity in in Java
  - in C: value in memory gets interpreted as Type *t*

```
int eastwood=0xD431;
char ly = (char) eastwood;    /* ly contains 49 now, why? */
```

- `&` address operator
  - Returns the address of a variable in memory
- `*`, `->` and `sizeof` are discussed later

Variables

- Always have to be initialized (as in Java)
  - Otherwise, their value is *undefined*
  - Initialization can be combined with declaration
- Can have *global* or *local* **scope**

```
int counter=0;

int gcd(int a, int b) {
    counter++;
    if (a==0) return b;
    if (b==0) return a;
    return gcd(b,a%b);
}

int main() {
   int eastwood=10164;
   char ly=240;
   printf("result: %d\n", gcd(ly,eastwood) );
   printf("function calls needed: %d\n", counter);
   return 0;
}
```

Global Variables

- Defined *outside* of functions
- Accessible in the program below the line of their definition
- Can be *overlayed* by local variables

- **Problems**
  - Missing context: relation between data objects and code using these objects is not visible
  - Functions can change variables at any time without the function‘s caller noticing/expection it (*side effects*)
  - More difficult program maintenance

→ **avoid global variables whenever possible!**

Local Variables

- Declared *inside* and *at the start* of a function or blocks
- Are not accessible outside of that function or block
- *Overlays (covers)* all previous definitions of an object with the same name; these are not accessible inside of the block!

- Which value is returned by main() in the example?

```
int a=0, b=1;

void bar(int b) {
    a=b;
}

void foo(int a, int b)
{
    {
        int b=a;
        int a=a+b;
    }
    bar(a);
}

int main() {
    int b=a;
    {
        int a=2;
        foo(a,b);
    }
    return a;
}
```

What Do We Know So Far...

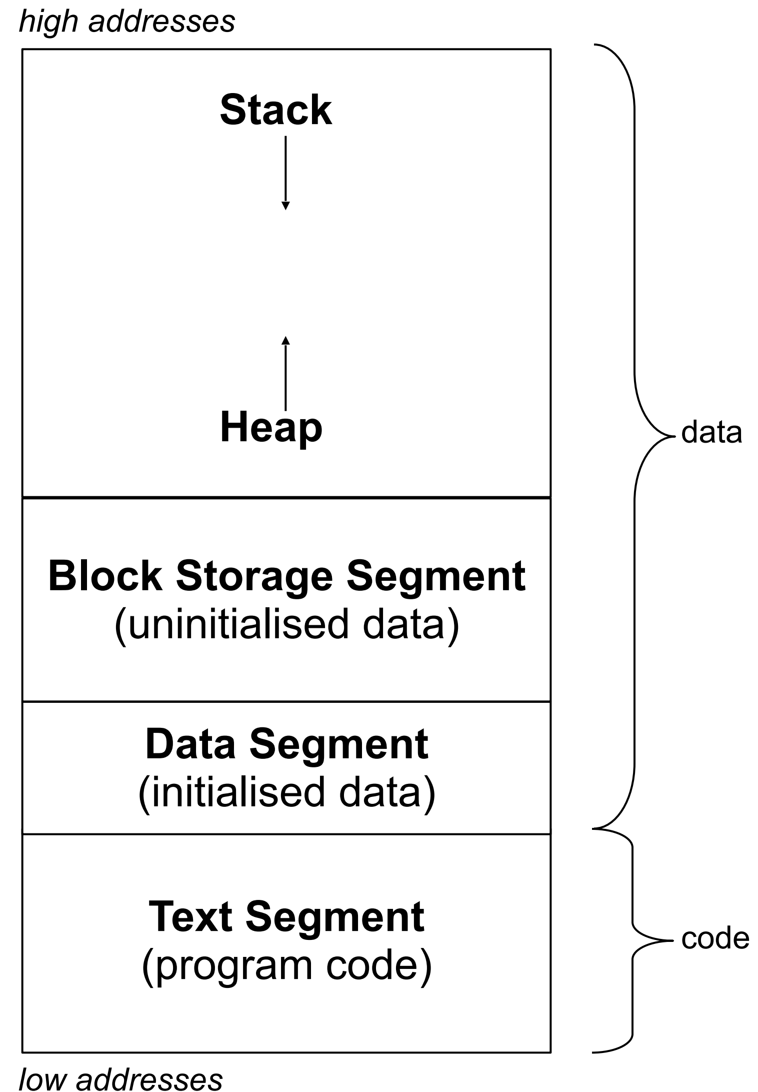
- Things *not* available in C:
  - classes
  - Exceptions
  - `public`, `private` and `protected` qualifiers
  - `new` and garbage collection
  - `import`
  - Single line comments using `//` => valid in C99
- Other things available in C (so far):
  - functions
  - global variables
  - *#include*

C keywords

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Program Memory Layout

- **Global variables** are stored in the BSS or data segment
- **Local variables** are stored on the stack
  - Created when related function is entered
  - Part of a function‘s *stack frame*
  - Removed from stack when function returns
- Heap: *dynamic* variables
  - in Java: created using `new`
  - in C: later... (keywords: pointer, `malloc()`)



Memory Layout (2)

```
#include <stdio.h>
int global_uninitialized;
int global_initialized=4711;

void uboot() {}

int main() {
     int local;
     printf("%p\n%p\n%p\n%p\n",
          &global_uninitialized,
          &global_initialized,
          &uboot,
          &local);
    return 0;
}
```

```
$ gcc mem.c -o mem.elf
$ nm mem.elf -r -n
080495a8 A _end
080495a4 B global_uninitialized
080495a0 b completed.5843
080495a0 A _edata
080495a0 A __bss_start
0804959c D global_initialized
08049598 d p.5841
08049594 D __dso_handle
08049590 W data_start
08049590 D __data_start
08049578 d _GLOBAL_OFFSET_TABLE_
080494a4 d _DYNAMIC
080494a0 d __JCR_LIST__
080494a0 d __JCR_END__
0804949c d __DTOR_END__
08049498 d __DTOR_LIST__
08049494 d __CTOR_END__
08049490 d __init_array_start
08049490 d __init_array_end
08049490 d __CTOR_LIST__
0804848c r __FRAME_END__
0804847c R _IO_stdin_used
08048478 R _fp_hw
0804845c T _fini
08048430 t __do_global_ctors_aux
0804842a T __i686.get_pc_thunk.bx
080483d0 T __libc_csu_init
080483c0 T __libc_csu_fini
08048379 T main
08048374 T uboot
08048350 t frame_dummy
08048320 t __do_global_dtors_aux
080482f0 T _start
08048278 T _init
         U printf@@GLIBC_2.0
```

- Why is variable `local` not shown in the list on the right?

Memory Layout (3)

```
#include <stdio.h>
int global_uninitialized;
int global_initialized=4711;

void uboot() {}

int main() {
    int local;
    printf("%p\n%p\n%p\n%p\n",
        &global_uninitialized,
        &global_initialized,
        &uboot,
        &local);
    return 0;
}
```

```
$ ./mem.elf
0x80495a4
0x804959c
0x8048374
0xffe62a80
```

```
$ gcc mem.c -o mem.elf
$ nm mem.elf -r -n
080495a8 A _end
080495a4 B global_uninitialized
080495a0 b completed.5843
080495a0 A _edata
080495a0 A __bss_start
0804959c D global_initialized
08049598 d p.5841
08049594 D __dso_handle
08049590 W data_start
08049590 D __data_start
08049578 d _GLOBAL_OFFSET_TABLE_
080494a4 d _DYNAMIC
080494a0 d __JCR_LIST__
080494a0 d __JCR_END__
0804949c d __DTOR_END__
08049498 d __DTOR_LIST__
08049494 d __CTOR_END__
08049490 d __init_array_start
08049490 d __init_array_end
08049490 d __CTOR_LIST__
0804848c r __FRAME_END__
0804847c R _IO_stdin_used
08048478 R _fp_hw
0804845c T _fini
08048430 t __do_global_ctors_aux
0804842a T __i686.get_pc_thunk.bx
080483d0 T __libc_csu_init
080483c0 T __libc_csu_fini
08048379 T main
08048374 T uboot
08048350 t frame_dummy
08048320 t __do_global_dtors_aux
080482f0 T _start
08048278 T _init
         U printf@@GLIBC_2.0
```

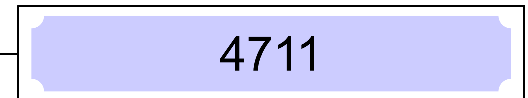
Pointer (variables)

- Variables have an address in memory
- The address can in turn be stored in another variable:

```
int eastwood = 4711;
int *p;
p = &eastwood; /* p "points" to "eastwood" now */
```

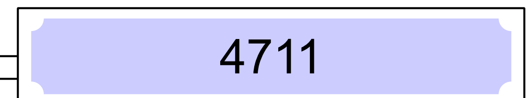
- **Variable:** name for a data object

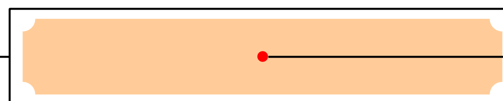
eastwood



- **Pointer variable (pointer):**
  name for a *reference to* a data object

eastwood



p 

Pointers (2)

- A pointer variable (***pointer***) stored the address of another variable: it *points* to that variable
- This address enables **indirect** access to the variable
- Commonly used in C code:
  - Functions can be are enabled to modify their parameters (resp. the objects these parameters *point to*), this is the way to implement *call by reference* in C!
  - Dynamic memory allocation and management
  - More efficient programs
- Disadvantages…
  - Program structure is less clear (which function can access which variables?)
  - Most common source of errors in C programs

Pointers (3)

- Syntax to create a pointer variable:
  Type *Name;
- Example:

```
int eastwood = 4711;
int *p;
int x;

p = &eastwood; /* p "points" to eastwood now */

x = *p; /* copies the object p points to into x */
```

- Address operator **&**
  &x returns the address of variable x
- Dereference operator *
  ***x** enables access to the content of the variable x points to

Pointers as Function Parameters

- Parameters are always passed *by value* in C (i.e., copied)
- A **function can never modify the value** of one of its parameters **in the** context of the **calling function**!
- Pointers are also passed *by value*
  ⇒ function is passed a **copy** of the address
- Using this pointer copy, the function can access the associated variable using the *-operator („dereference“) and change it this way:
  ⇒ *call by reference*

```
void inc(int *x) {
    (*x)++;
}
int main(void) {
    int foobie = 42;
    inc(&foobie);
...
```

Pointers as Function Parameters (2)

- Example:
  swap variable
  values

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void)
{
    int olerant = 42, ernational = 4711;

    swap(&olerant, &ernational);

    printf(“%d %d\n”, olerant, ernational);
    return 0;
}
```

- swap() takes addresses of two variables
- Use of *-operators to access the referenced variables

Pointers to Structures

- Analogous to “pointers to variables”
- Example: linked list

```
#include <stdio.h> /* definition of NULL */

struct listelement {
    char ly;
    struct listelement *next;
} a, b, c;

a.next = &b;
b.next = &c;
c.next = NULL; /* = 0, read the docs! */
```

- **NULL:** ”special” address value, here used to indicate list termination

Pointers to Structures (2)

- Acess to structure elements using a pointer
- Well-known procedure:
  - * operator references the structure
  - . operator used to access the element
  - Operator precedence at work: we have to use brackets here!

```
struct listelement {
    char ly;
    struct listelement *next;
} a, *p;

p = &a;
(*p).ly = 'a';
```

- identical, but (syntactically) nicer (more easily readable!):
  -> -operator

```
p->ly = 'a';
```

Pointers (4)

- Also possible, but not as common: **pointers to pointers** (to pointers to...)

```
int x, *ptr,
**ptr_ptr;
ptr_ptr = &ptr;
ptr = &x;
**ptr_ptr = 4711;
```

- Pointers to functions

```
int (*func)();
func = &myfunction;
(*func)();
```

  - E.g. used to pass a function as a parameter to another function
  - Example: library function **qsort** takes a parameter which is a pointer to a comparison function to compare tuples of elements

Typedef

- Definition of a **new name** for an existing type
- Syntax: like variable declaration, put **typedef** in front:

```
typedef int Length; /* this line does not use any
memory! */

Length len, max_length;
void set_length(Length l) { ... }
```

- **Abstraction:** the actual type is hidden and can be exchanged easily (well-known examples: pid_t, FILE)
- **Documentation:** simple names easier to read/understand than complex pointer to a structure

```
typedef struct listelement *LEPtr;
LEPtr elem1, elem2;
LEPtr find_elem(LEPtr firstelem, int searchnum)
{ ... }
```

Arrays

- Similar to Java, but...
  - Dimensions can only be constants! (except for C99...)
  - Uninitialized **global** Arrays filled with 0s (*not guaranteed for embedded!*)
  - Contents of uninitialized **local** arrays is undefined
  - When using initializers, missing values at the end are filled with zeros

```
int primes[100] = {2, 3, 5, 7, 11, 13, 17};
/* primes[7] to primes[99] are set to 0! */

/* automatic dimensioning */
int even[] = {2, 4, 6, 8, 10, 12};
```

  - No *bounds checks* when accessing arrays!
    ⇒ Effects when reading/writing outside of the array bound range from „nothing happens“ to program crashes to completely undefined behavior!
  - Real **multi-dimensional arrays** are also possible

Multi-Dimensional Arrays

- Definition and initialization

```
int calendar[12][31]; /* 12 “rows”, 31 “columns” */
int lecture_limits[][5] =
      { { 27, 27, 22, 27, 27 },   /* odd weeks */
        { 27, 27, 22, 27, 27 } }; /* even weeks */
```

- Default values defined as for one-dimensional arrays
- Access identical to Java:

```
lecture_limits[1][3]--; /* one less chair */
```

Arrays (2)

- Arrays of pointers

```
int *quark[10];
```

- Arrays of structures

```
struct listelement my_elements[50] =
 { { 12, NULL }, { 80, NULL } };
```

- Arrays of char
  - Representation of *strings* in C!
  - *C strings* are sequences of **char**s terminated by a 0 character
  - Initialization like normal arrays *or* using double quotes

```
/* three times the same using different syntax */
char text1[] = { 'f', 'o', 'o', 0 };
char text2[] = { 102, 111, 111, 0 };
char text3[] = "foo";
```

  - More on strings later

Programs Consisting of Multiple Files

main.c

```
int main() {
    hello();
    return 0;
}
```

hello.c

```
#include <stdio.h>

void hello() {
    printf("Hello, world!\n");
}
```

- Let's try...

```
$ gcc main.c hello.c -o hello_world.elf
```

- ...this works!
  - Files are compiled *separately*
  - *implicit declaration* of hello()
    (return type **int**, no checking of parameter types) (=> warning!)
  - Compiler notes that hello is an *undefined symbol*
  - Linker finds this symbol in the other compilation unit and patches the call site with the correct function address
- Very questionable approach (why?)

Programs Consisting of Multiple Files (2)

main.c

```
void hello();

int main() {
     hello();
     return 0;
}
```

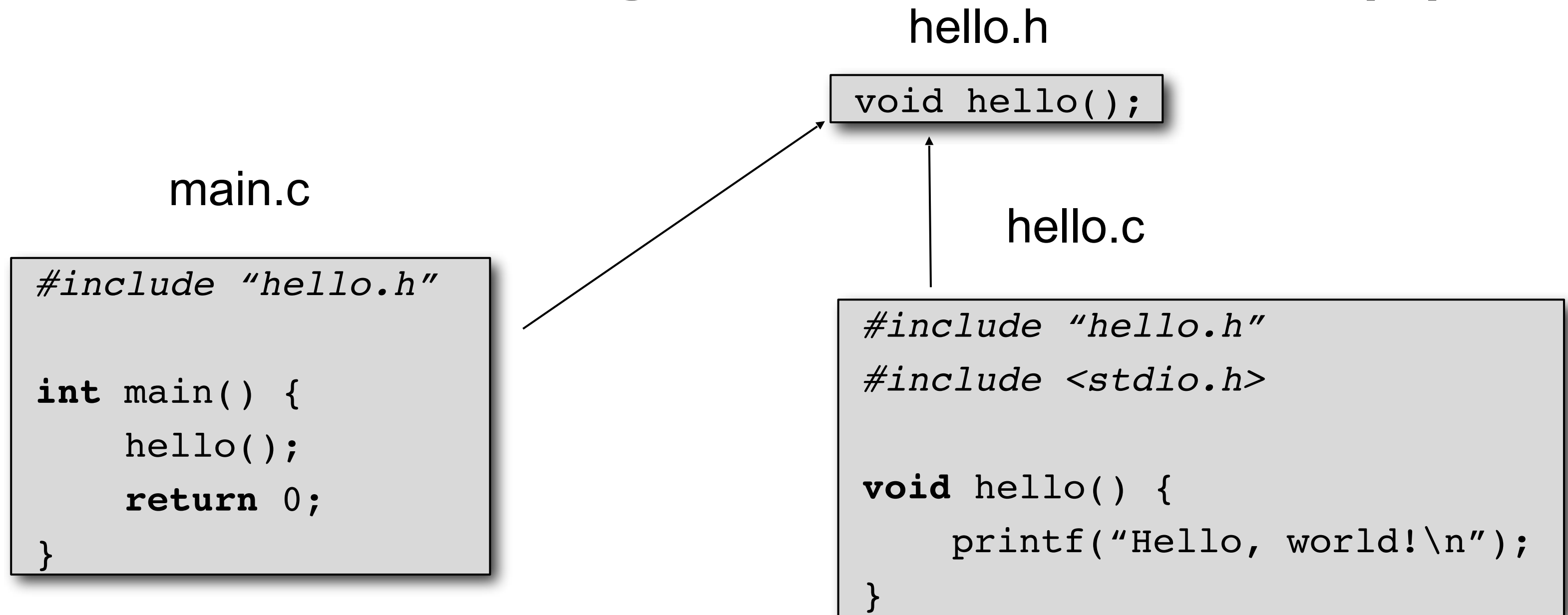
hello.c

```
#include <stdio.h>

void hello() {
     printf(“Hello, world!\n”);
}
```

- This also works when using `-Werror`
- ...still not a *good* solution
  - Declaration of hello() in hello.c might change!
  - Both have to use **an identical declaration** (why?)

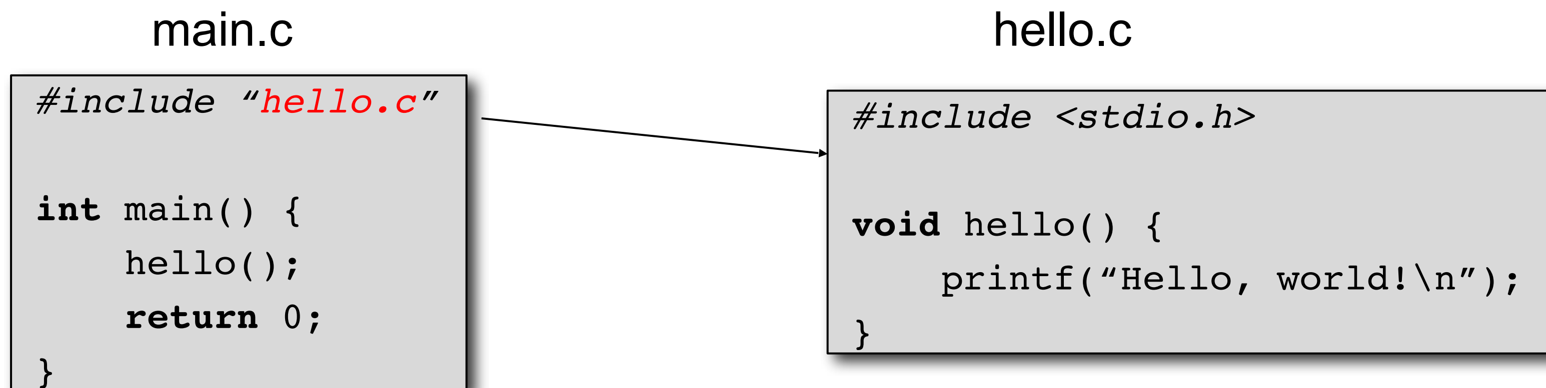
- Even better: both use **the same declaration**
  → Use the *preprocessor*

Programs Consisting of Multiple Files (3)



- This is the correct way to do it!
  - `#include` is a *preprocessor command*
  - preprocessor *copies* contents of header file to its location in the file
  - File paths in „“ are relative to the directory of the current .c file
  - File paths in <> relate to compiler-defined and platform-specific directories

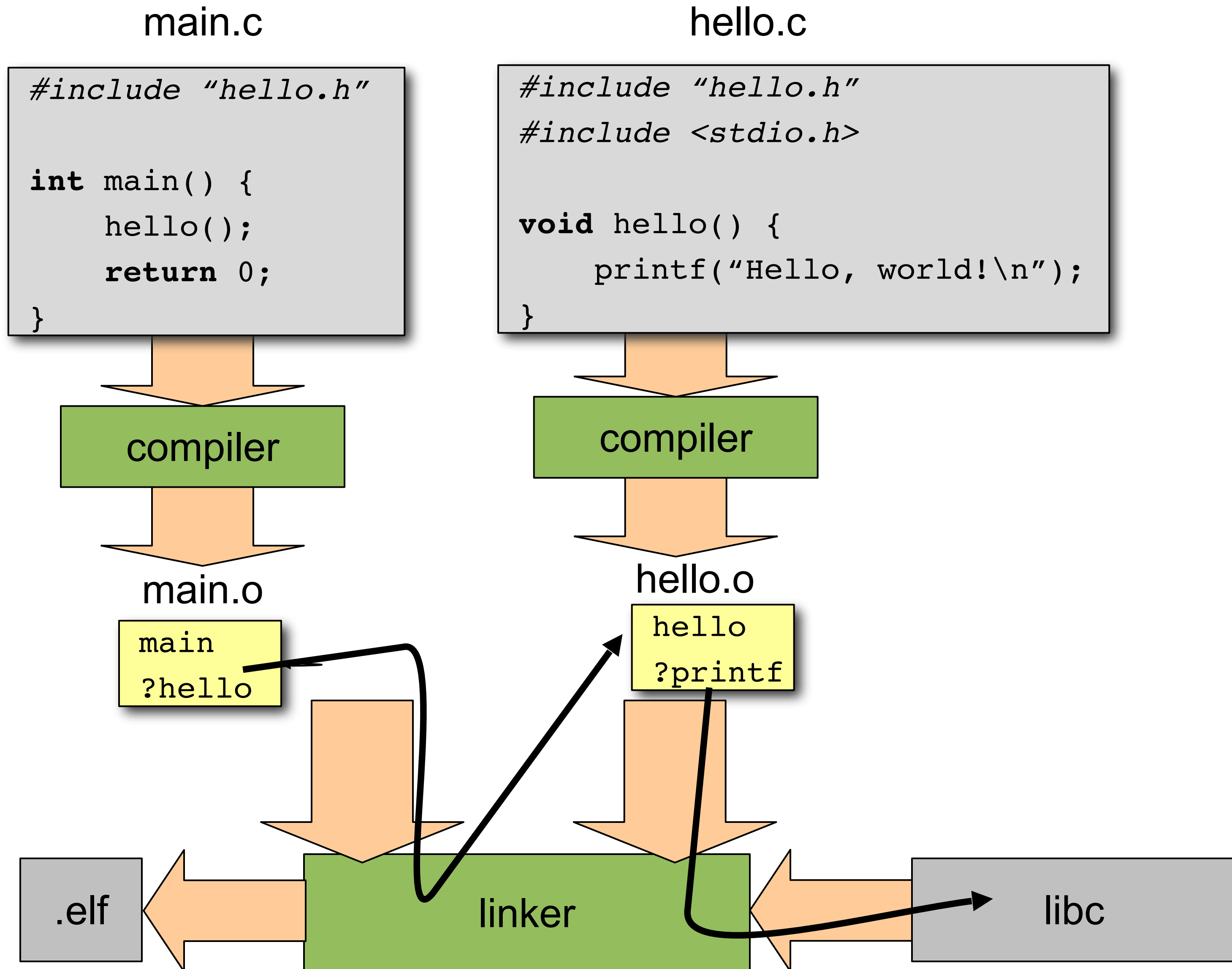
Programs Consisting of Multiple Files (4)



- **Never use this!**

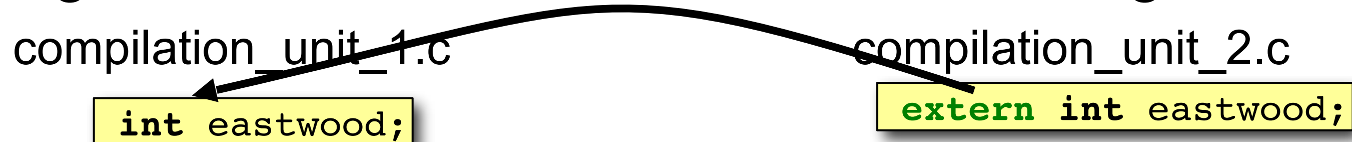
```
$ gcc main.c hello.c -o hello_world.elf
/tmp/ccvCawGO.o: In function `hello':
hello.c:(.text+0x0): multiple definition of `hello'
/tmp/cc6gea2y.o:main.c:(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

Programs Consisting of Multiple Files(5)



Modules

- For global variables we distinguish between...
  - variables only accessed from inside the module they are declared in
  - and variables which are also accessed from other modules
- Access to global variables of other modules using `extern`



- `static` makes global variables *invisible* for other modules

  - Enables *encapsulation* of data inside a module
  - Prevents *name collisions* at link time
  - Considered good programming style

Modules (2)

- To access functions in other modules, `extern` is *not* required...
- Functions can also be declared `static`
  - Used for functions that should only be visibly inside of a module (i.e., the current C source code file); these are not part of the module interface
- Local variables can also be declared `static`
  - *This has a completely different meaning!*
  - Variable value “survives” between subsequent function calls:

```c
unsigned odd_number(void)
{
    static unsigned n = 1;
    return n += 2;
}
```

Preprocessor

- Definition and use of preprocessor symbols

```
#define Horse 4711
printf(“%d”, Horse);
```

transformed into

```
printf(“%d”,4711);
```

- conditional compilation

```
#define Horse 4711
#ifdef Horse
printf(”Horse.\n”);
#else
printf(“No Horse.\n”);
#endif
```

transformed into

```
printf(”Horse.\n”);
```

- Preprocessor is smart enough not to modify strings
- ...but it‘s not very much smarter...

```
#define Horse 17+4
if (3*Horse == 63) {
    ...
}
```

transformed into

```
if (3*17+4 == 63)
{
    ...
}
```

Preprocessor (2)

- Parameterized macros

```
#define SQUARE(x) x*x
printf(“%d”,SQUARE(3));
```

transformed into

```
printf(“%d”,3*3);
```

- No space between name und „(“, no „;“ at the end of the line!

- pitfall *brackets*

```
#define SQUARE(x) x*x
printf(“%d”,SQUARE(1+2));
```

transformed into

```
printf(”%d”,1+2*1+2);
```

- Solution: use brackets around *every use* of the parameter as well as the overall expression!

```
#define SQUARE(x) ((x)*(x))
printf(“%d”,SQUARE(1+2));
```

transformed into

```
printf(”%d”,((1+2)*(1+2)));
```

- Macros spanning multiple lines: use „\“ at end of line

```
#define SQUARE(x) \
          ((x)*(x))
```

Preprocessor (3)

- Parameterized macros
  - **pitfall:** *side effects!*

```
#define SQUARE(x) ((x)*(x))
int x = 2;
printf("%d",SQUARE(++x));
```

transformed into

```
printf("%d",((++x)*(++x));
```

  - ...*what is the output*?
  - Even worse when using *function calls* with side effects!

```
...
printf("%d",SQUARE(launch_missile()));
```

- if conditions with expressions

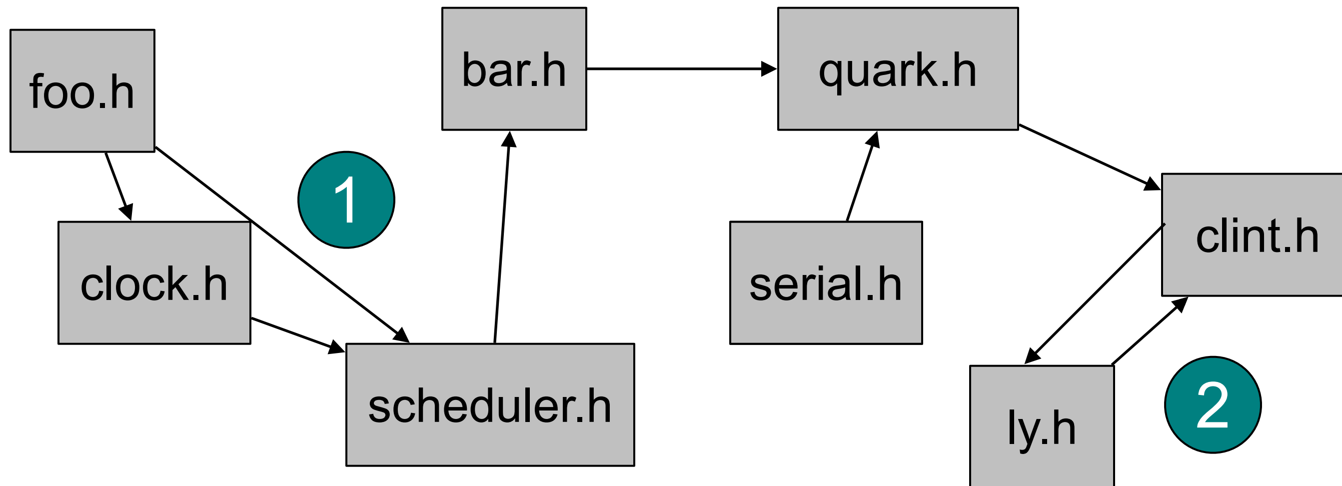
```
#define Horse 4711
#if Horse == 815
printf("Horse.\n");
#else
printf("No Horse.\n");
#endif
```

transformed into

```
printf("No Horse.\n");
```

Include Guards

- Scenario: multiple header files, mutual includes



  - Contents of header files repeat (at 1)
  - Endless recursion (at 2)
- Solution: include guards →

<whatever>.h

```
#ifndef __WHATEVER_H__
#define __WHATEVER_H__
/* header file contents */
#endif
```

Unions

- **`struct`**
  groups multiple elements in memory *one after the other*
- **`union`** groups multiple elements in memory *on top of each other*
  - i.e. the *same memory area* can be accessed as `int` as well as an array `char[4]`(on a 32 bit machine)

```
#include <stdio.h>

union dontknow {
    char ly[4];
    int eastwood;
}

union dontknow whatever="foo";

int main() {
    printf("%x\n", whatever.eastwood);
    whatever.eastwood=7496034;
    printf("%s\n",whatever.ly);
    return 0;
}
```

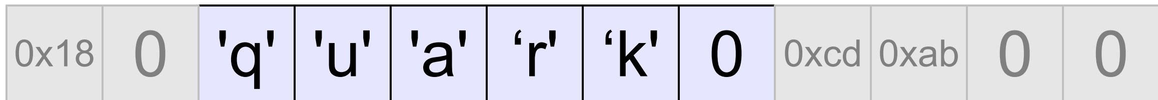
Only the *first* element can be initialized!

```
$ ./unions.elf
6f6f66
bar
```

Arrays in Memory

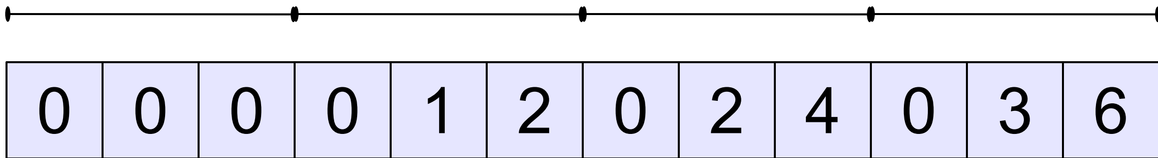
- Array elements are stored in memory one after the other

```
short int foo = 24;
char wort[] = “quark”;  /* == {'q','u', ... 'l',0}*/
int kuckuck = 0xABCD;
```



- The elements can, in turn, be complex data structures
  - e.g., a 3-element array

```
int mult[4][3] = { {0,0,0},
                   {0,1,2},
                   {0,2,4},
                   {0,3,6} };
```



Arrays in Memory (2)

```
struct element {
   int x,y;
   char text[16];
};

struct element matrix[3][2] = { { {0,0,"null,null"}, {0,1,"null,eins"} },
                                { {1,0,"eins,null"}, {1,1,"eins,eins"} },
                                { {2,0,"zwei,null"}, {2,1,"zwei,eins"} }
                              };
```

```
Contents of section .data:
 8049520 00000000 00000000 2c940408 0000(                  ..
 8049530 00000000 00000000 00000000 0000(                  ..
 8049540 00000000 00000000 6e756c6c 2c6e756c  ........null,nul
 8049550 6c000000 00000000 00000000 01000000  l...............
 8049560 6e756c6c 2c65696e 73000000 00000000  null,eins.......
 8049570 01000000 00000000 65696e73 2c6e756c  ........eins,nul
 8049580 6c000000 00000000 01000000 01000000  l...............
 8049590 65696e73 2c65696e 73000000 00000000  eins,eins.......
 80495a0 02000000 00000000 7a776569 2c6e756c  ........zwei,nul
 80495b0 6c000000 00000000 02000000 01000000  l...............
 80495c0 7a776569 2c65696e 73000000 00000000  zwei,eins.......
```

matrix[1][1].x

Arrays and Pointers

- Array identifiers can be seen as *constant pointers* to the start of the array

```
char text[] = “quark”;
char *c = text; /* synonymous to &(text[0]) */
*c = 'k';
```

- Pointers can be used like array identifiers

```
c[1] = 'w';  /* text is now “qwark”! */
```

- ...but not always the other way round
  - Array identifiers are no variables, but constants!
  - They have no address in memory – different to pointers!
  - &text is nonsentical, but returns the same address as `text`!

```
*text = 'k';  /* text is now “kwark”! */
/* text = c;  WOULD THROW AN ERROR! */
c = (char*) &text; /* correct address, incorrect type */
```

Pointer Arithmetics

- We can “compute” using pointer and array identifiers:

```
char text[] = “quark”;
char *c = text+1;
*c = 'w';                    /* “qwark” */
*(text+4) = 'b';             /* “qwarb” */
*(c-1) = 'z';                /* “zwarb” */
```

- `text[4]` is another expression for `*(text+4)`
- `text+1` can be written as `&(text[1])`
- even `c[-1]` is possible instead of `*(c-1)`!

Pointer Arithmetics (2)

- This code outputs “quark” three times

```
char text[]="quark";

int i;
char *c;

for (i=0;i<7;i++)           /* normal array access */
    printf("%c",text[i]);

for (i=0;i<7;i++)           /* using pointer arithmetics */
    printf("%c",*(text+i));
                            /* more pointer arithmetics */
for (c=text;c<=&text[6];c++)
    printf("%c",*c);
```

p: pointer, *s*: scalar value

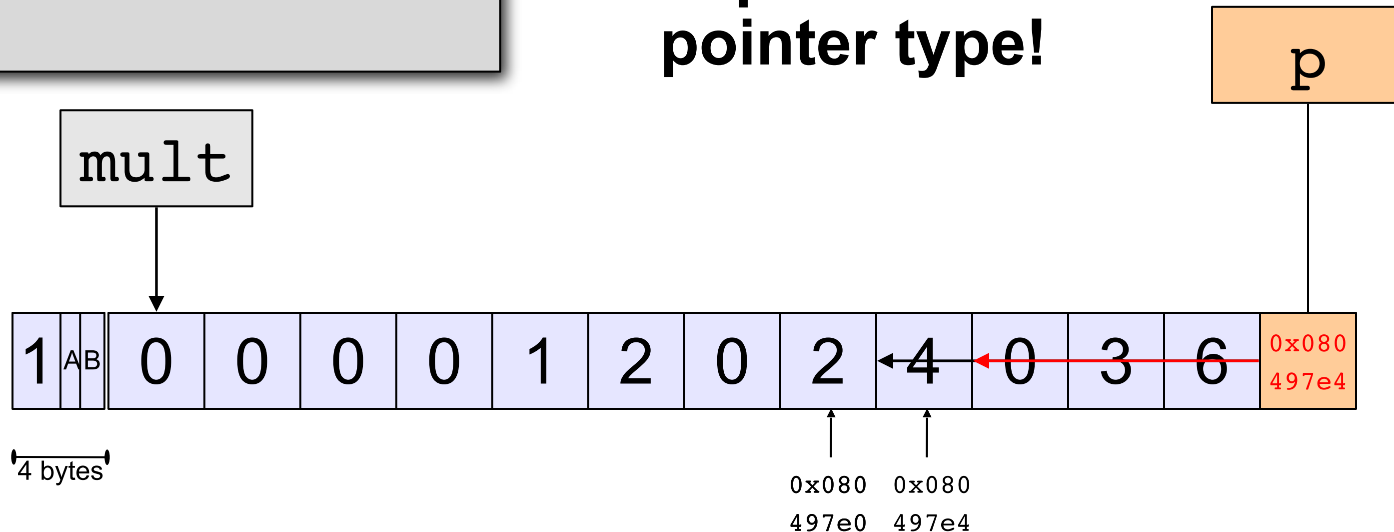
```
  p+s   is equal to  &(p[s])
*(p+s)  is equal to    p[s]
  p++   is equal to    p=&(p[1])
```

Pointer Arithmetics (3)

```
short int dummy = 1;
char bla='A',blb='B';
int mult[4][3] = { {0,0,0},
                   {0,1,2},
                   {0,2,4},
                   {0,3,6} };
int *p = &mult[2][1];

int main() {
   p++;
   return 0;
}
```

- Also works for arrays which are *not* of type `char`
- since $p+s$ = `&(p[s])`, p is *not* **incremented** by 1 (Byte), but rather **by 4**!

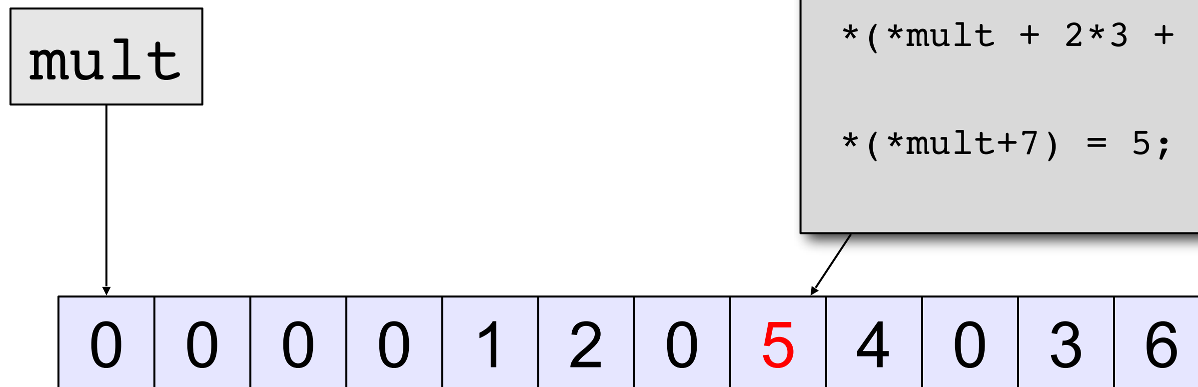
→ **Address difference depends on the pointer type!**



Multi dimensional Arrays

- Pointer arithmetics also works for multi dimensional arrays
- Six different ways to write the same operation →

```
char mult[4][3] = { {0,0,0},
                    {0,1,2},
                    {0,2,4},
                    {0,3,6} };

mult[2][1] = 5;

*(mult[2]+1) = 5;

(*(mult+2))[1] = 5;

*(*(mult+2)+1) = 5;

*(*mult + 2*3 + 1) = 5;

*(*mult+7) = 5;
```



Strings

- Literals

```
char foo[]="quark";
int main () {
     foo[0]='k';
     return 0;
}
```

  - operates on char array char-Feld with contents { ‘q’, ‘u’, ‘a’, ‘r’, ‘k’ }

*

```
Contents of section .data:
 8049540 71756965 73656c00
                 quark.
```

```
char *foo="quark";
int main () {
     foo[0]='k';
     return 0;
}
```

```
$ ./literals.elf
Segmentation fault
```

  - Pointer to read-only data

*

```
Contents of section .rodata:
 8048428 03000000 01000200
71756965
 73656c00 ........quark.
```

- What about this code?

```
int main () {
     char foo[]="quark";
     foo[0]='k';
     return 0;
}
```

*) output of `objdump -s programm.elf`

Strings

- Inputting strings
  - **dangerous** (buffer overflow!)

```c
char foo[64];
scanf("%s",&foo);
```

  - better:

```c
char foo[64];
scanf("%63s",&foo);
```

- Library functions

```c
char *strcpy(char *dest, const char *src);
int strcmp(const char *s1, const char *s2);
char *strcat(char *dest, const char *src);
```

  - operate character by character until '\0' is read
  - also potentially dangerous
  - better: `strncpy, strncmp, strncat`
  - additional parameter limits processing to *n* characters

Further reading (if you want to dig deeper)…

1. Brain W. Kernighan, Dennis M. Ritchie
   The C Programming Language (2nd Edition)
   Prentice Hall 1988, ISBN 978-0131103627
   – still *the* standard book on C programming, but this doesn't include more recent developments in C
2. Peter van der Linden
   Expert C Programming: Deep C Secrets (1st Edition)
   Prentice Hall 1994, ISBN 978-0131774292
   – this is more of a second book for somewhat experienced C programmers, written in a more conversational style and supplemented with real-world examples and tricks
3. Igor Zhirkov
   Low-Level Programming: C, Assembly, and Program Execution on Intel® 64 Architecture
   Apress 2017, ISBN 978-1484224021
   – this book concentrates on the relation of computer architecture, assembly and C code; it covers some interesting topics on compiling C code
4. W. Richard Stevens, Stephen A. Rago
   Advanced Programming in the UNIX (R) Environment (2nd Edition)
   Addison-Wesley 2005, ISBN 978-0201433074
   – the standard book on programming in C on Unix-like systems