

TDDDB68 Concurrent Programming and Operating Systems

Lecture 2: Introduction to C programming

Mikael Asplund
Real-time Systems Laboratory
Department of Computer and Information Science

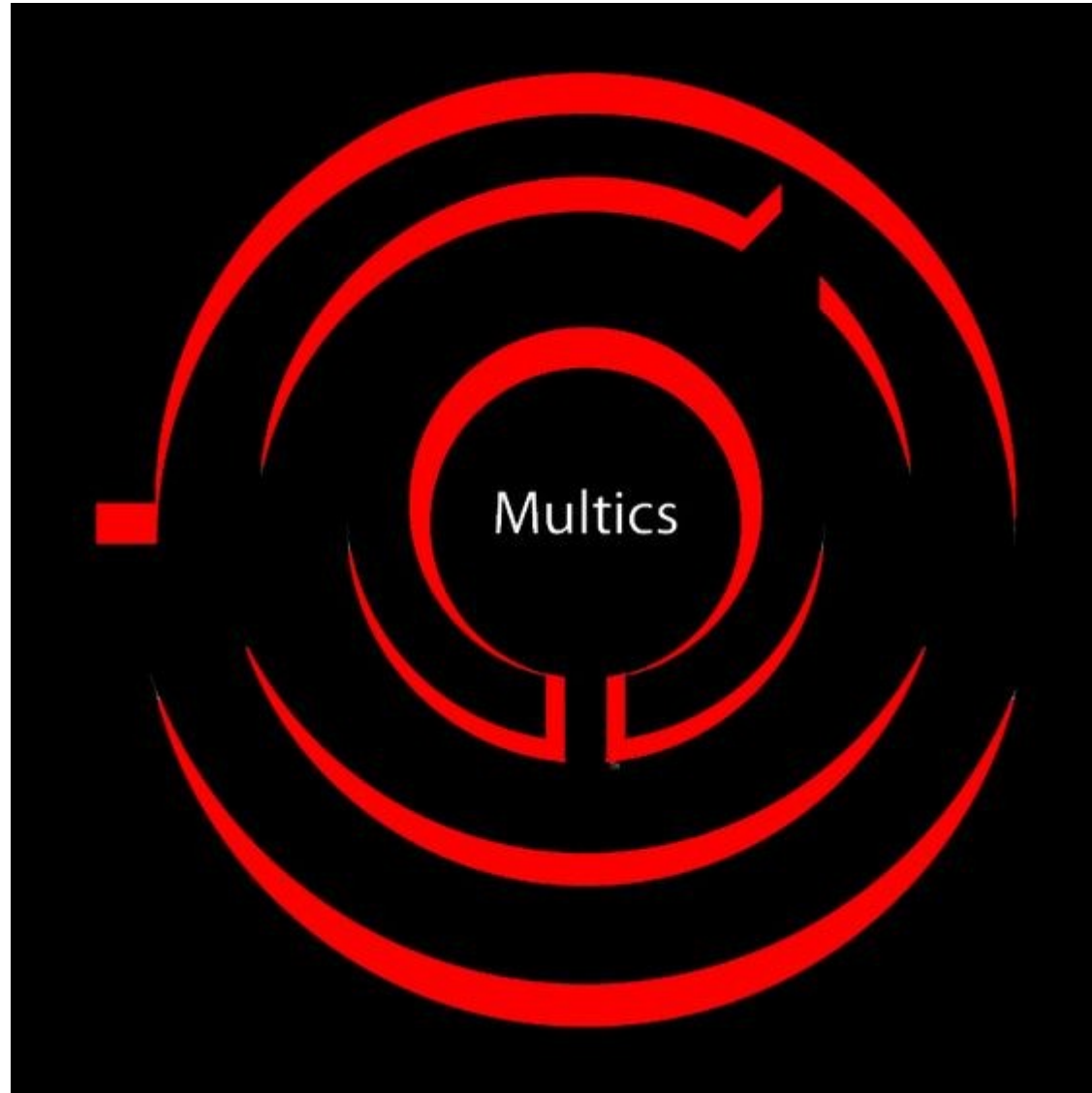
Copyright Notice:

Thanks to Christoph Kessler for much of the material behind these slides.

The lecture notes are partly based on Silberschatz's, Galvin's and Gagne's book ("Operating System Concepts", 7th ed., Wiley, 2005). No part of the lecture notes may be reproduced in any form, due to the copyrights reserved by Wiley. These lecture notes should only be used for internal teaching purposes at the Linköping University.

A bit of history

1965-1970



Unix

- More straightforward than Multics
- Early 1970's
- Originally implemented in assembly
- Needed a programming language

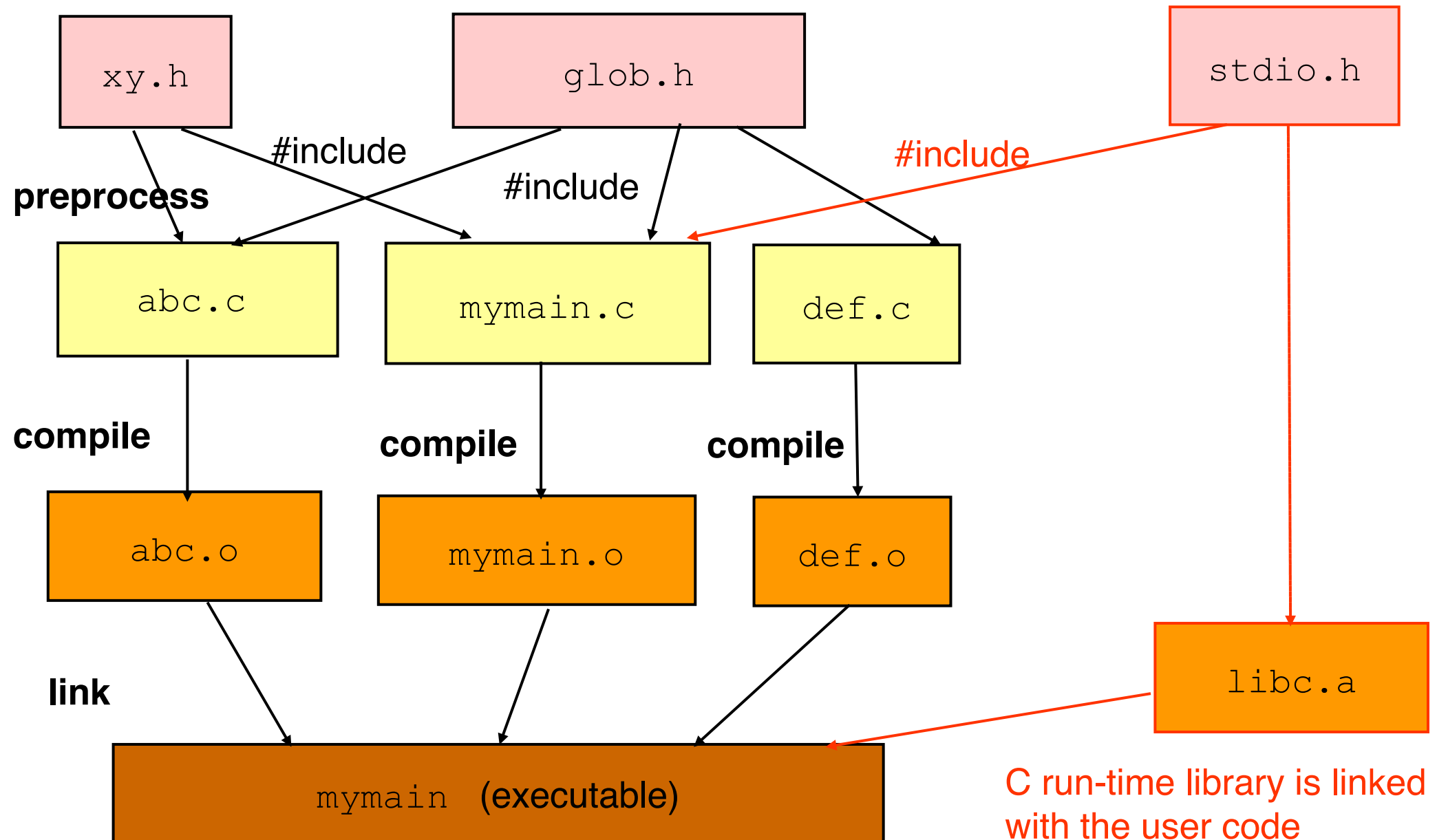
C programming language

- Successor of B, variant of BCPL
- Book 1978 by Brian W. Kernighan and Dennis M. Ritchie ("K&R-C")
- 1989 ANSI standard
- Latest standard: C11

Basic principles

- Imperative
- Typed
- Medium abstraction level
- Structure
 - Flexible
 - Typically functionality-oriented

File relationships



A first program

Data types in C

- **Primitive types**
 - int, char, etc.
- **Composite data types**
 - arrays
 - structs
 - unions
- Programmer can define new type names with **typedef**

Primitive data types

- integral types: **char**, **short**, **int**, **long**; **enum**
 - can be signed or unsigned, e.g. **unsigned int** counter;
 - sizes are implementation dependent (compiler/platform)
 - use **sizeof** (*datatype*) operator to write portable code
- floatingpoint types: **float**, **double**, **long double**
- pointers

Constants and Enumerations

- **Constant variables:**

```
const int red = 2;
```

```
const int blue = 4;
```

```
const int green = 5;
```

```
const int yellow = 6;
```

- **Enumerations:**

```
enum { red = 2, blue = 4, green, yellow } color;
```

```
color = green; // expanded by compiler to: color = 5;
```

- **With the preprocessor:**

- symbolic names, textually expanded before compilation

```
#define red 2
```

```
...
```

(no =, no semicolon)

- In C, constants are often capitalized: RED, BLUE, ...

Variable declaration/definition

glob.h

```
/*Comment: declaration of
globally visible functions
and variables: */
extern int incr( int );
extern int initval;
```

Note the difference:
Declarations announce signatures (names+types).
Definitions declare (if not already done) AND allocate memory.
Header files should never contain definitions!

abc.c

```
#include "glob.h"

// definition of var. initval:
int initval = 17;

// definition of func. incr:
int incr( int k )
{
    return k+1;
}
```

mymain.c

```
#include <stdio.h>
#include "glob.h"

int counter; // locally def.

void main( void )
{
    counter = initval;
    printf("new counter: %d",
        incr( counter ) );
}
```

More on composite data types

Structs

Unions

- Unions implement variant records
 - all attributes share the same storage
- Unions break type safety
- If handled with care, useful in low-level programming

Arrays

Pointer: type + address

The * symbol

- The * symbol has four **separate** uses:
 - Used for declaration/definition of a pointer:
`int *px;`
`int **py;`
 - Used to dereference a pointer:
`*px = 5;`
`int b = **py;`
 - Multiplication:
`a = 3*4;`
`a *= 4;`
 - In comments:
`/* this is a comment */`

The & symbol

- The & symbol has three **separate** uses:

- Getting the address of a variable:

```
int *p = &a;
```

- Bitwise and:

```
unsigned int x = y&z;
```

```
y &= z;
```

- Logical and:

```
if a && b {
```

```
    ...
```

Pointer arithmetics

Pointer arithmetics

- Integral values can be added to / subtracted from pointers:

```
int *q = p + 7;
```

```
// new value of q is (value of p) + 7 * sizeof(pointee-type of p)
```

- Arrays are simply constant pointers to their first element:

- Notation `b[3]` is "syntactic sugar" for `*(b + 3)`

- `b[0]` is the same as `*b`

b:

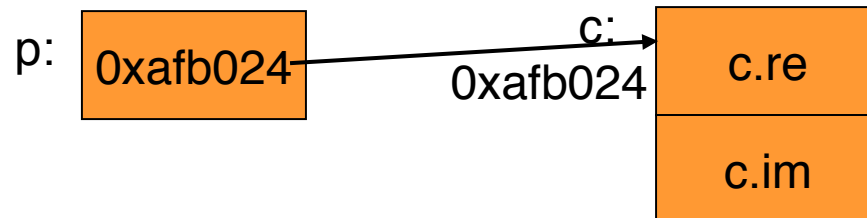
3	6	8	4	3
---	---	---	---	---

- A pointer can be subtracted from another pointer:

- **unsigned int** `offset = q - p;`

Pointers and structs

- **struct** My_IComplex { **int** re, im; } c, *p;
p = &c;
 - p is a pointer to a struct
- p->re is shorthand for (*p).re
 - Example: as p points to c, &(p->re) is the same as &(c.re)
 - Example: elem->next = NULL;



Why do we need pointers in C?

- Defining recursive data structures (lists, trees ...) – as in Java
- Argument passing to functions
 - simulates reference parameters – missing in C (not C++)
 - `void foo (int *p) { *p = 17; }` Call: `foo (&i);`
- Arrays are pointers
 - Handle to access dynamically allocated arrays
 - A 2D array is an array of pointers to arrays:
 - `int m[3][4];` // `m[2]` is a pointer to start of third row of `m`
- For representing strings – missing in C as basic data type
 - `char *s = "hello";` // `s` points to char-array `{'h','e','l','l','o', 0 }`
- For dirty hacks (low-level manipulation of data)

Pointer type casting

Pointer type casting

- Pointer types can be casted to other pointer types
 - `int i = 1147114711;`
`int *pi = &i;`
`printf ("%f\n", * (float *) pi));` // prints 894.325623
 - All pointers have the same size (1 address word)
 - But no conversion of the pointed data! (cf. unions)
 - Compare this to: `printf("%f\n", (float) i);`
 - A source of type unsafety,
but often needed in low-level programming
- **Generic pointer type:** `void *`
 - Pointee type is undefined
 - Always requires a pointer type cast before dereferencing

Pointers to functions (1)

- Function declaration
 - `int f(float);`
- Function call: `f(x)`
 - `f` is actually a (constant) pointer to the first instruction of function `f` in program memory
 - Call `f(x)` dereferences pointer `f`
 - push argument `x`; save PC and other reg's; `PC := f`;
- Function pointer variable
 - `int (*pf)(float);` // `pf` is a pointer to a function
// that takes a **float** and returns an **int**
 - `pf = f;` // `pf` now contains start address of `f`
 - `pf(x);` // or `(*pf)(x)` dereferencing (call): same effect as `f(x)`

Pointers to functions (2)

- Most frequent use: generic functions
- Example: Ordinary sort routine
 - `void bubble_sort(int arr[], int asize)`
`{ int i, j;`
`for (i=0; i<asize-1; i++)`
`for (j=i+1; j<asize; j++)`
`if (arr[i] > arr[j])`
`... // interchange arr[i] and arr[j]`
`}`
 - Need to rewrite this for sorting in a different order?
 - Idea: Make `bubble_sort` generic in the compare function

Pointers to functions (3)

- Most frequent use: generic functions
- Example: Generic sort routine
 - `void bubble_sort(int arr[], int asize, int (*cmp)(int,int))`
`{ int i, j;`
`for (i=0; i<asize-1; i++)`
`for (j=i+1; j<asize; j++)`
`if (cmp (arr[i] , arr[j]))`
`... // interchange arr[i] and arr[j]`
`}`
 - `int gt (int a, int b) { if (a > b) return 1; else return 0; }`
 - `bubble_sort (somearray, 100, gt);`
`bubble_sort (otherarray, 200, lt);`

Storage classes in C (1)

- **Automatic variables**

- Local variables and formal parameters of a function
- Exist once per call
- Visible only within that function (and function call)
- Space allocated automatically on the function's stack frame
- Live only as long as the function executes

```
int *foo ( void ) // function returning a pointer to an int.
{
    int t = 3; // local variable
    return &t; // ?? t is deallocated on return from foo,
    // so its address should not make sense to the caller...
}
```

Storage classes in C (2)

- **Global variables**

- Actually a misnomer, should be called extern storage class
- Declared and defined outside any function
 - `extern int y; // y seen from all modules; only declaration`
 - `int y = 9; // only 1 definition of y for all modules seeing y`
- Space allocated automatically when the program is loaded

- **Static variables**

- **static int** counter;
- Allocated once for this module (i.e., not on the stack) even if declared within a function!
- Value will survive function return: next call sees it

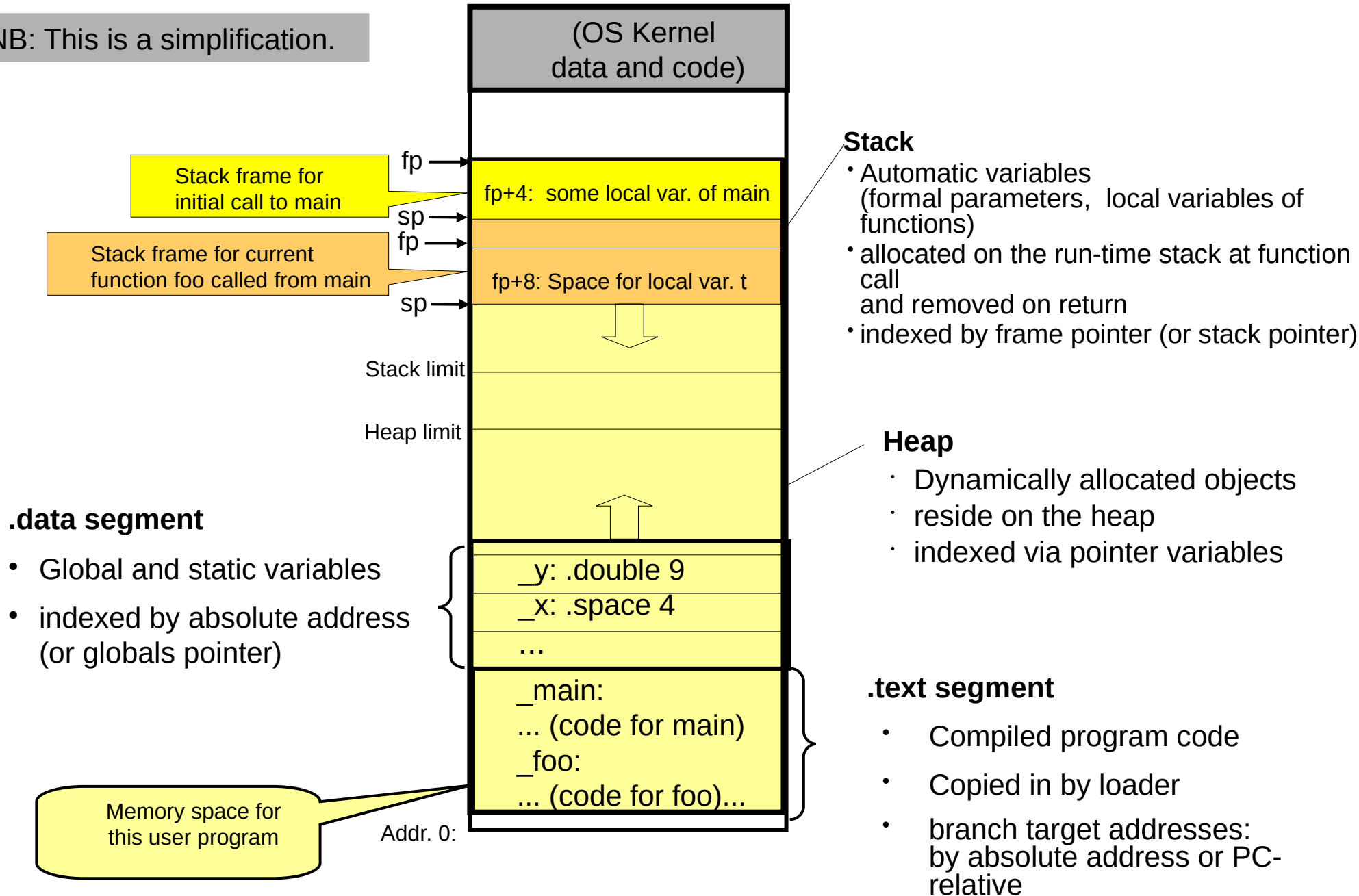
Dynamic memory

Dynamic allocation of memory in C

- `malloc(N)`
 - allocates a block of N bytes on the heap
 - and returns a generic (**void ***) pointer to it;
 - this pointer can be *type-casted* to the expected pointer type
 - Example: `icplx *p = (icplx *)malloc (sizeof (icplx));`
- `free(p)`
 - deallocates heap memory block pointed to by `p`
- Can be used e.g. for simulating ***dynamic arrays***:
 - Recall: arrays are pointers
 - `int *a = (int *) malloc (k * sizeof(int));` `a[3] = 17;`

Run-Time Memory Organization

NB: This is a simplification.

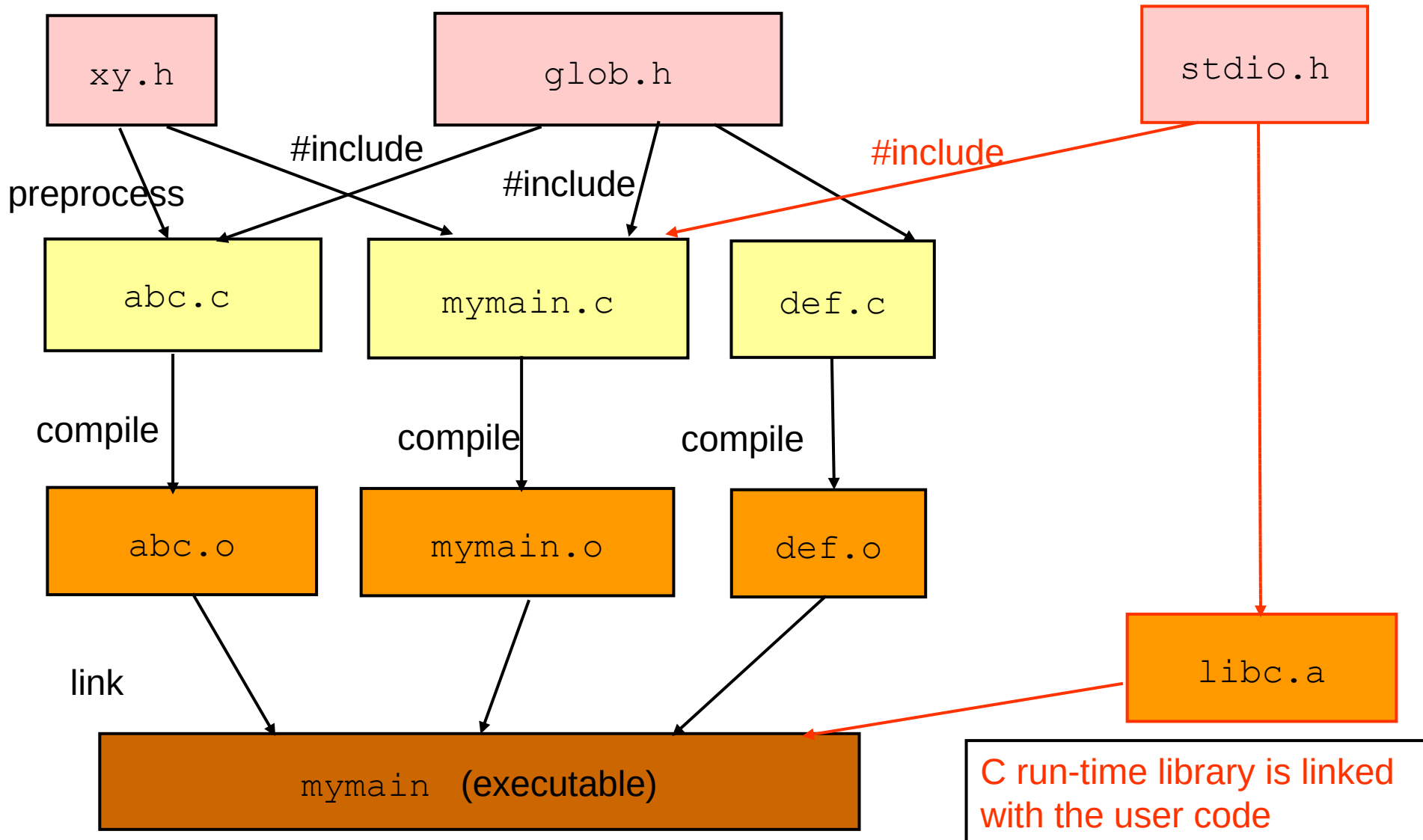


C: There is much more to say...

- Type conversions and casting
- Bit-level operations
- Operator precedence order
- Variadic functions
(with a variable number of arguments, e.g. printf())
- C standard library
- C preprocessor macros
- I/O in C
- ...

- Now: Compiling, Linking, Loading, Debugging

Compiling and Linking



Compiling C Programs

- Example: GNU C Compiler: gcc (Open Source)
- One command calls preprocessor, compiler, assembler, linker
- **Single module:** Compile and link (build executable):
 - gcc mymain.c executable: a.out
 - gcc -o myprog mymain.c rename executable
- **Multiple modules:** Compile and link separately
 - gcc -c -o mymain.o mymain.c compiler only
 - gcc -c -o other.o other.c compiles other.c
 - gcc other.o mymain.o call the linker, -> a.out
- **make** (e.g. GNU gmake)
 - automatizes building process for several modules
- Check the man pages!

How to relocate code?

Relocation table:

- line 3, patch base+1
- line 3, patch base+5

Symbolic
addressing:

...

Definition of data X

...

If (X) goto P

...

P: ...

Relative
addressing:

0: ...

1: Space for data

2: ...

3: If (*1) goto 5

4: ...

5: ...

Absolute addressing:

243: ...

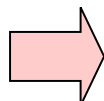
244: Space for data

245: ...

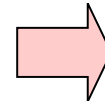
246: If (*244) goto 248

247: ...

248: ...

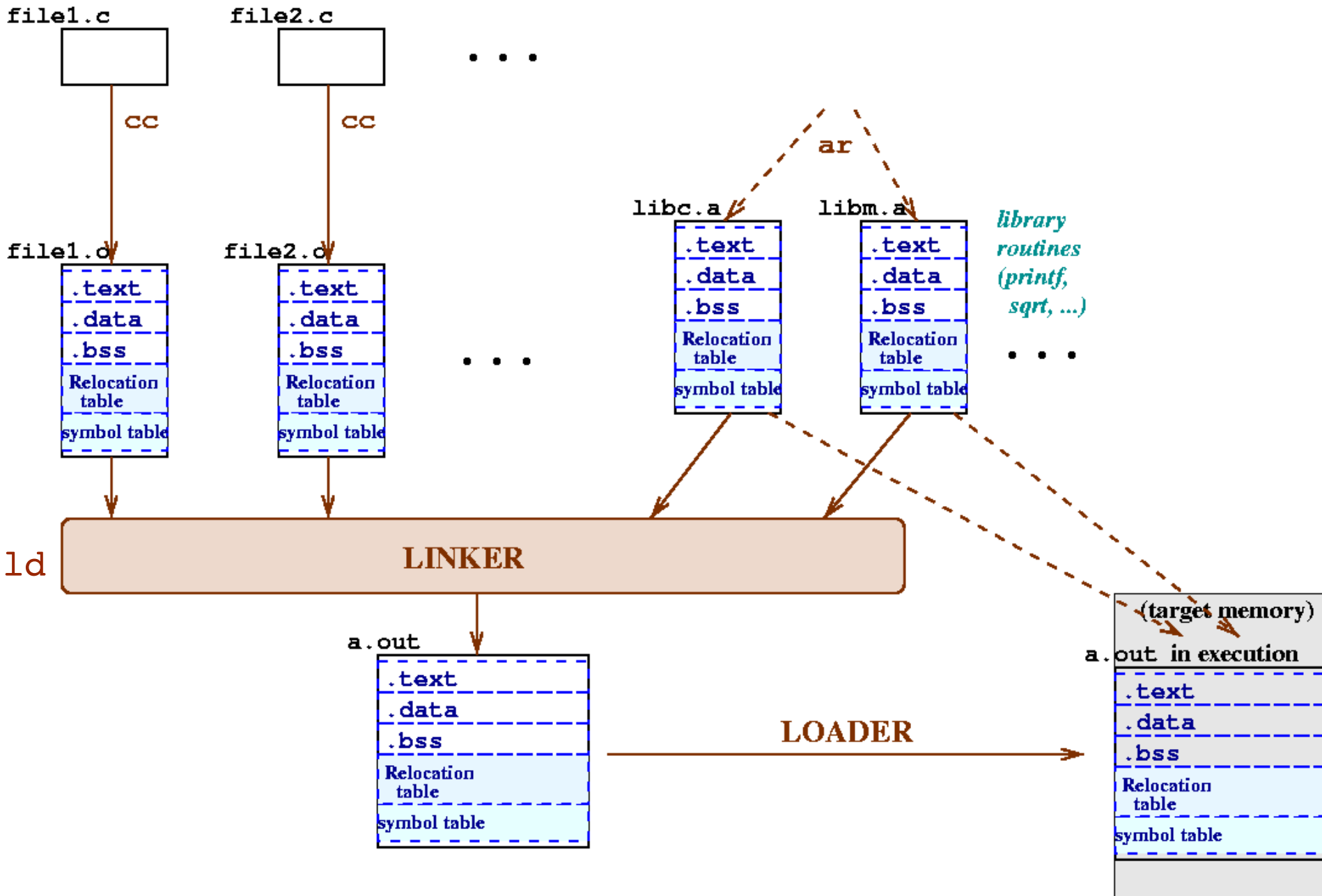


as



Static
loading

Linking Multiple Modules



Background: How the Linker Works

- 1) Read all object modules (including library archive modules)
 - 2) Merge the code, data, stack/heap segments of these into a single code, data, stack/heap segment
 - 3) Resolve global symbols (e.g., global functions, variables): check for duplicate globals, undefined globals
 - 4) Write the resulting object module, with a new relocation table
 - 5) Mark the resulting file executable.
- **Variants of static linking:**
 - Dynamic linking
 - Dynamic loading

Sources of information

- **Lab-Kickoff Session**
 - Programming exercise:
C pointer programming and debugging
- **GDB tutorial**
- **Books on C**
 - B. Kernighan, D. Ritchie: *The C Programming Language, 2nd ed.*, Prentice Hall, 1988. Covers ANSI-C.
 - P. Darnell, P. Margolis: *Software Engineering in C*. Springer, 1988.
 - H.M. Deitel, P.J. Deitel: *C How to Program*. Prentice-Hall, 1992.
- **Online resources**