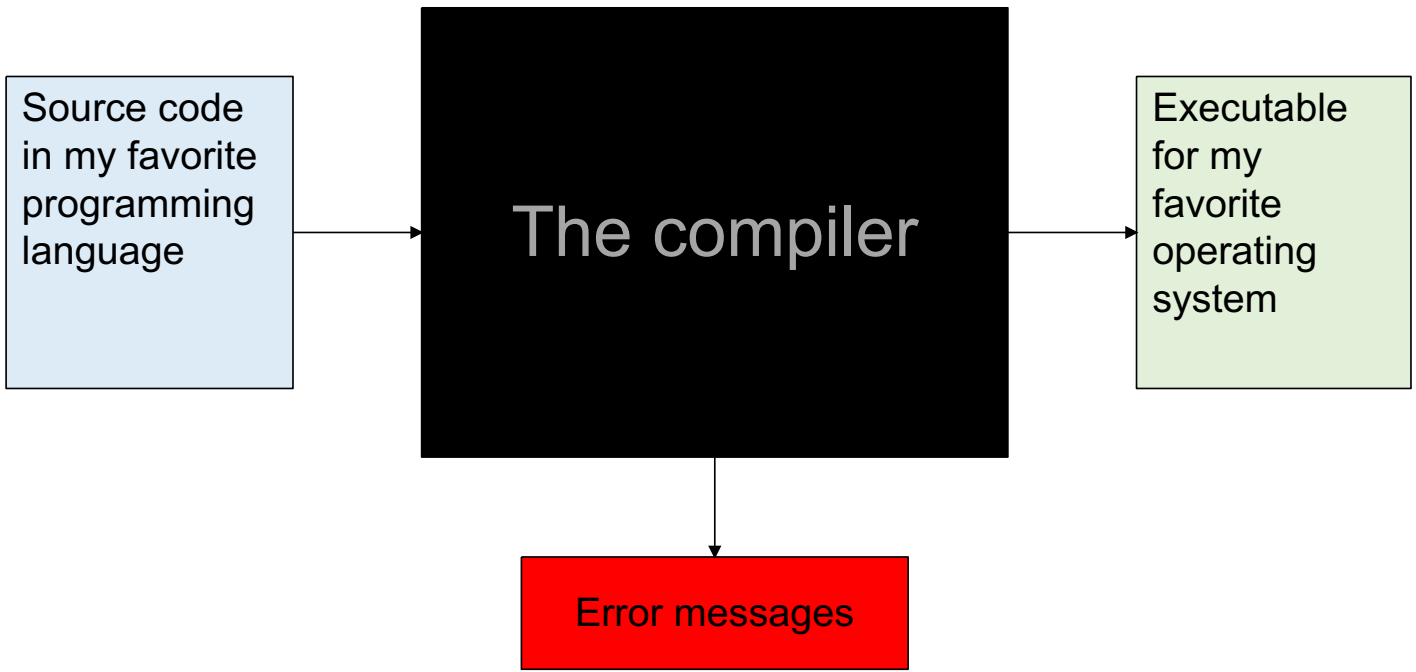
Programming Languages

NSWI170 Computer Systems

Jakub Yaghob, Martin Kruliš

Naïve view of a compiler



Formal view of a compiler

- From slides of the course Compiler Principles
  - Let's have an input language L_{in} generated by a grammar G_{in}
  - Let's have an output language L_{out} generated by a grammar G_{out} or accepted by an automaton A_{out}
  - The compiler is a mapping $L_{in} \rightarrow L_{out}$, where for all w_{in} in L_{in} exist w_{out} in L_{out}. The mapping does not exist for w_{in} not in L_{in}
- Don't worry!
  - You will learn this in the Automata and Grammars (NTIN071) course (obligatory) and then Compiler Principles (NSWI098) course (elective)

Naïve understanding of a grammar

- Formal description of a language
  - Rules – struktura / formát / pravidla
  - Lexical elements – jednotlivá určená slova

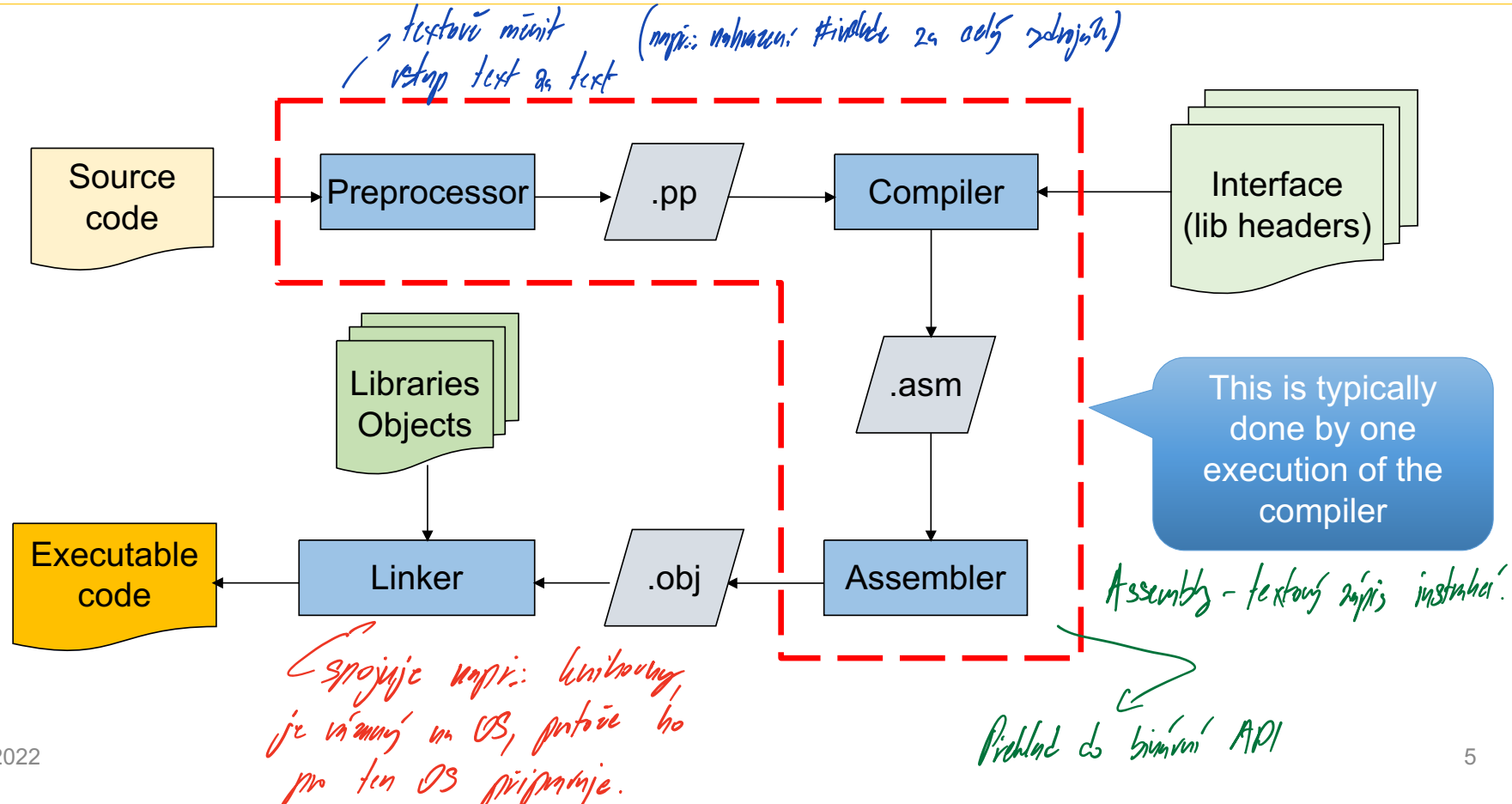
iteration-statement:

while (*expression* **)** *statement*

do *statement* **while (** *expression* **) ;**

for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

More practical view of a translation



Linker/librarian/loader

- Library
  - A collection of compiled source modules and other resources (in one file)
  - Static linking (.lib, .a) vs. dynamic linking (.dll, .so)
- Linking
  - Integrating the results of the different translations and libraries together into one executable for given OS
  - Relocations, position-independent code
- Loader
  - Part of OS, loads the executable into memory (and its dynamically linked libs)
  - Relocation again

Library example

```
// mylibrary.h
extern int var;
extern int open(const char *name);
extern int read(int f, int size,
                void *buf);
```

Preprocessor

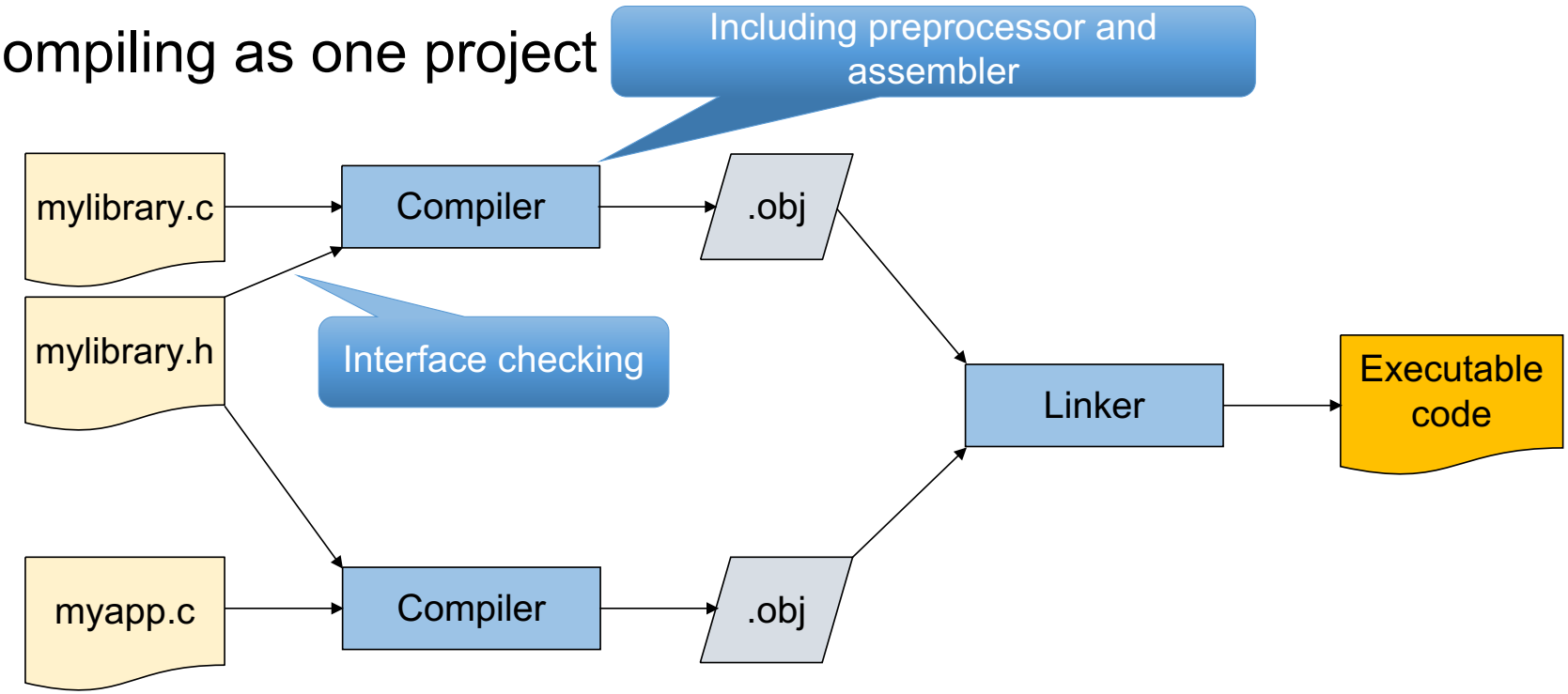
```
// myapp.c
#include <mylibrary.h>

int main(int argc, char **argv) {
  char buf[10];
  int file = open("myfile.txt");
  int err = read(file, 10, buf);
}
```

Linker/loader

```
// mylibrary.c
#include <mylibrary.h>
int var = 8;
int open(const char *name) { ... }
int read(int f, int size, void *buf)
{ ... }
```

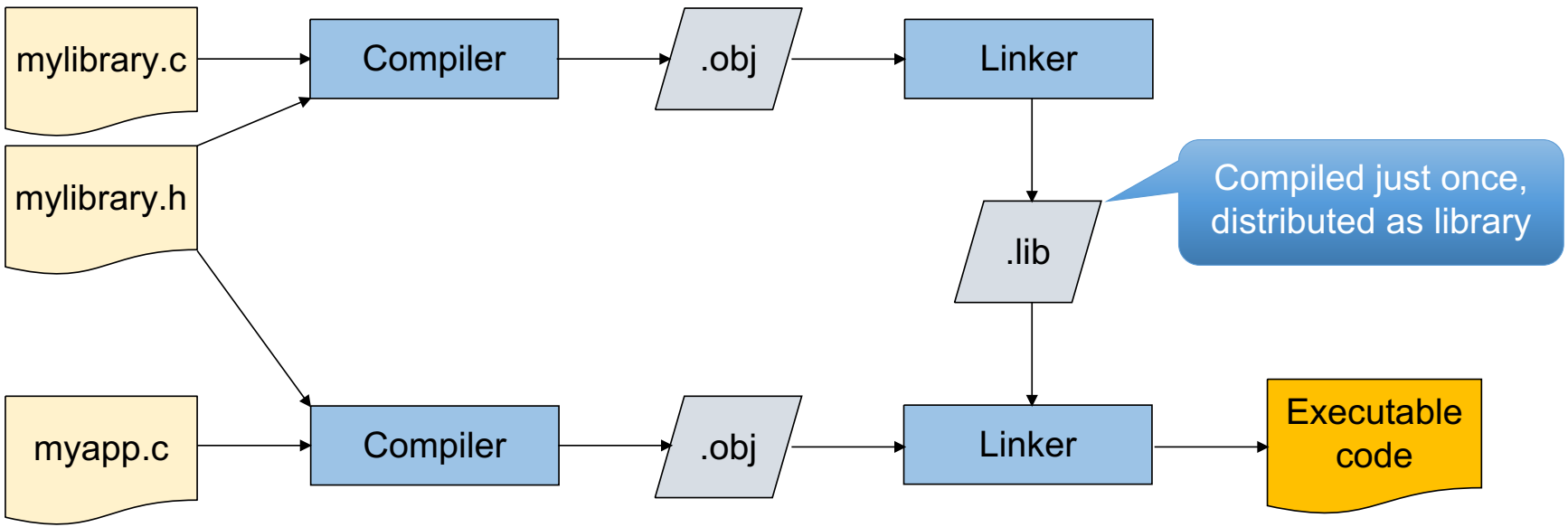
Library example compilation

- Compiling as one project



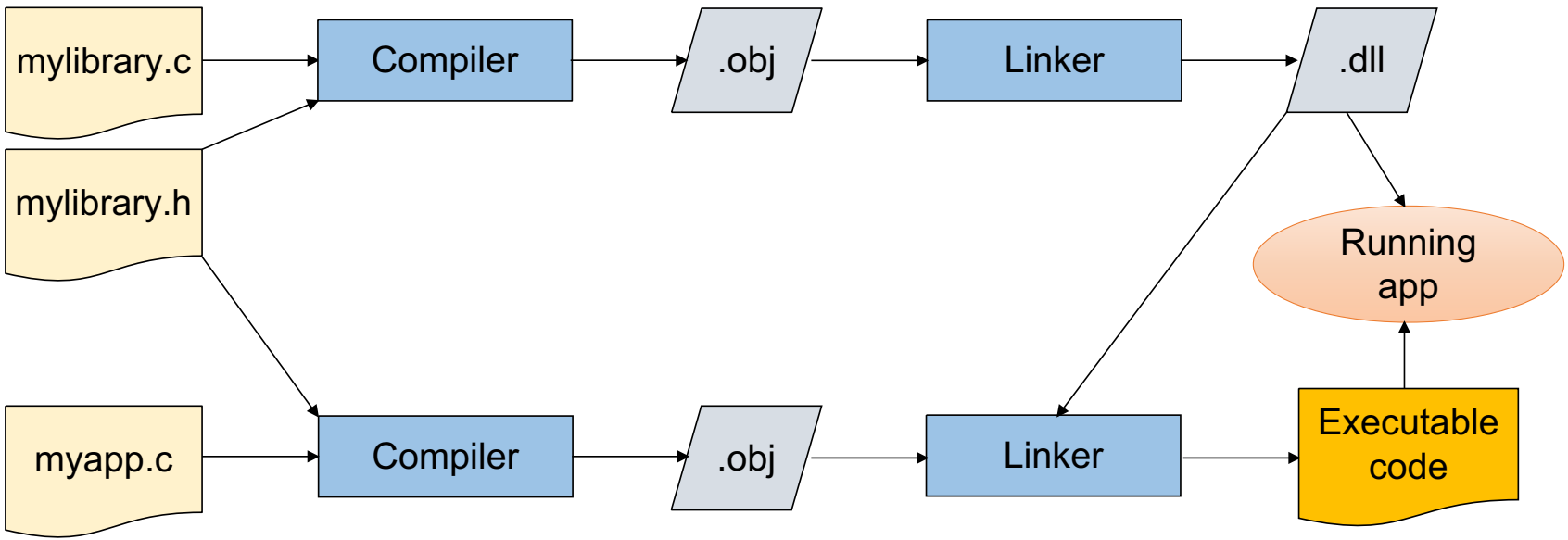
Library example compilation

- Compiling separately, linking as static library



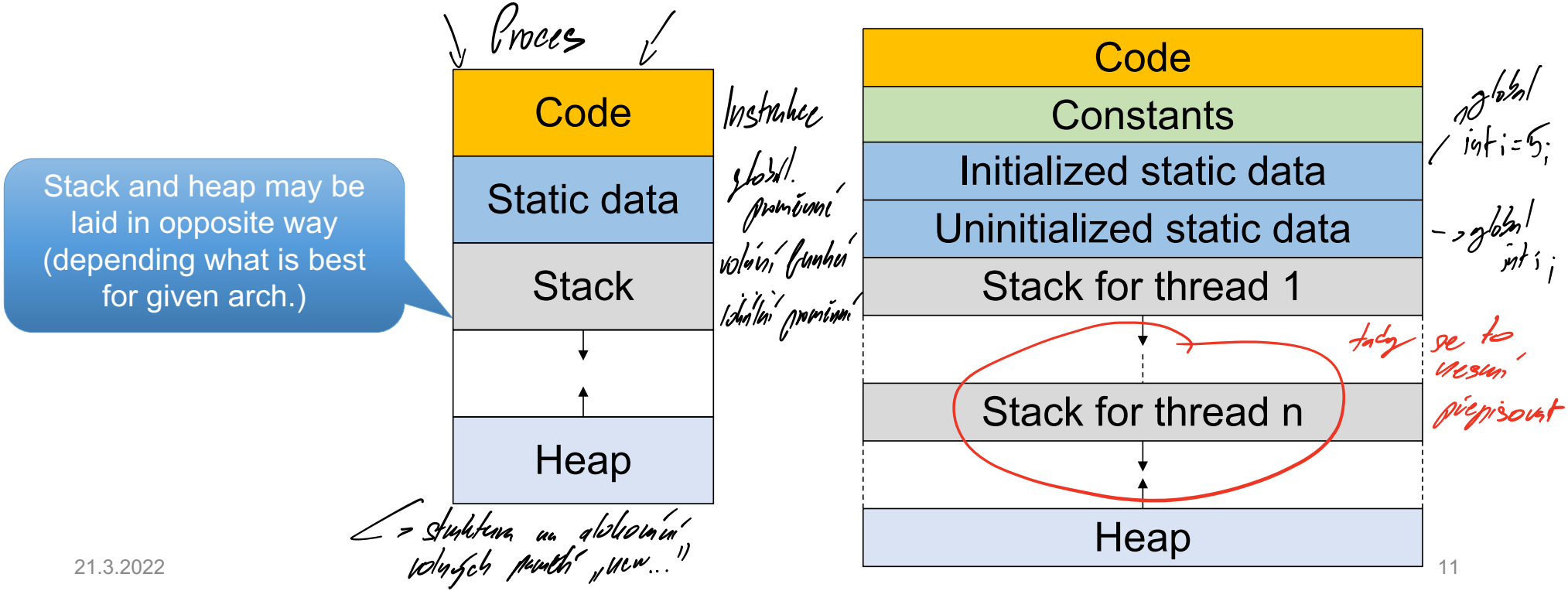
Library example compilation

- Compiling separately, linking as dynamic library

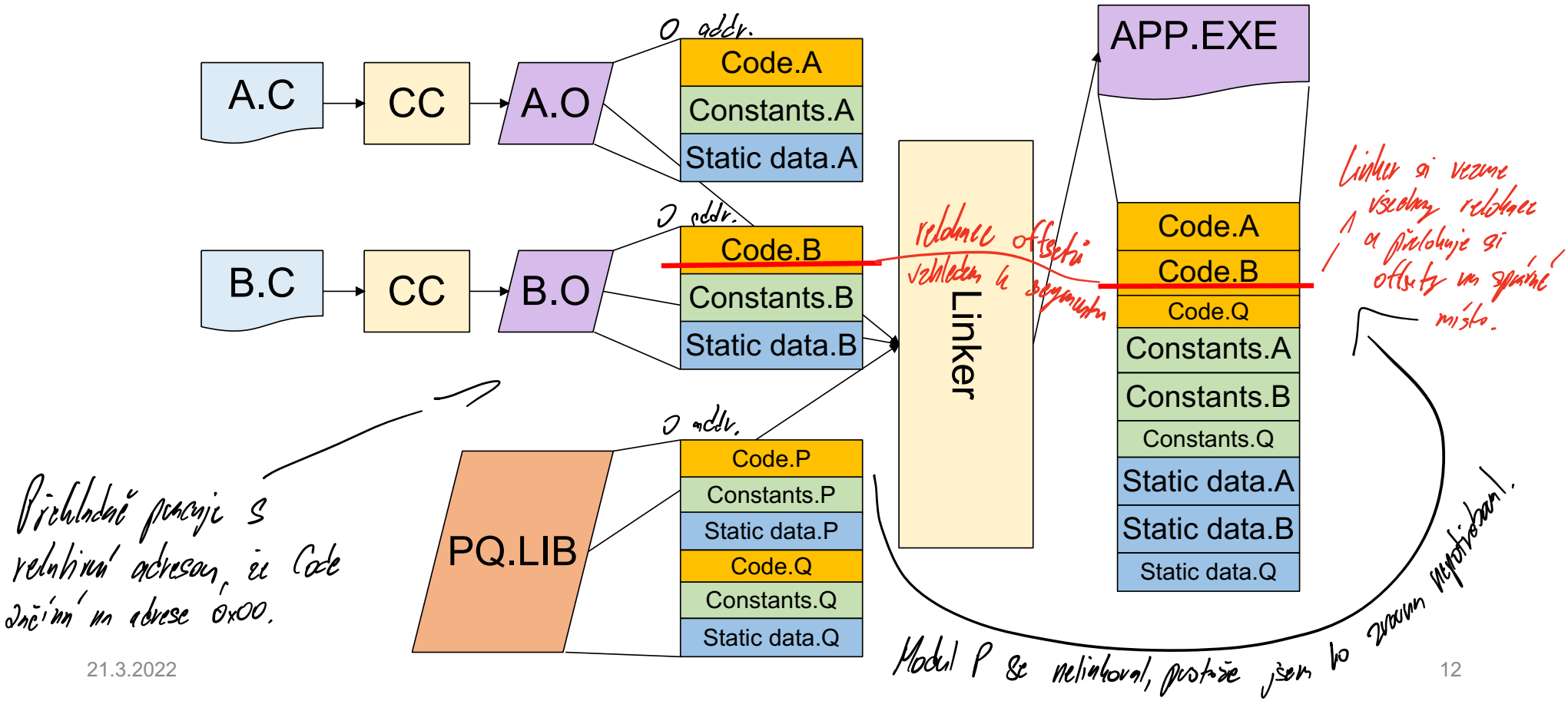


Memory organization

- Memory organization during procedural program execution



Linking


A.C
CC
A.O
0 addr.
Code.A
Constants.A
Static data.A
B.C
CC
B.O
0 addr.
Code.B
Constants.B
Static data.B
PQ.LIB
0 addr.
Code.P
Constants.P
Static data.P
Code.Q
Constants.Q
Static data.Q
Linker
APP.EXE
Code.A
Code.B
Code.Q
Constants.A
Constants.B
Constants.Q
Static data.A
Static data.B
Static data.Q

Run-time

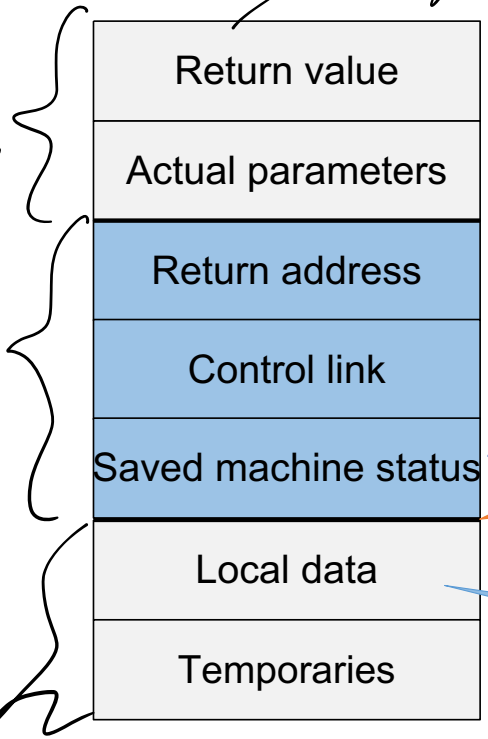
- Static language support
  - Compiler
  - Library interface
    - E.g., header files (C/C++)

} To, co potřebuje za předpoklad

- Dynamic language support
  - Run-time program environment
    - Storage organization
    - Memory content before execution
    - Constructors and destructors of global objects
  - Libraries
  - Calling convention

Function call

- Activation record (stack frame)
  - Return value
    - May be larger (structure)
    - Small values returned via registers
  - Saved machine status
    - Return address to the code
    - (Some) registers
  - Control link
    - Activation record of the caller



Function call

- Calling convention
  - Public name mangling
  - Call/return sequence for functions and procedures
    - Housekeeping responsibility
  - Parameter passing
    - Registers, stack
    - Order of passed parameters
  - Return value
    - Registers, stacks
  - Registers role
    - Parameter passing, scratch, preserved

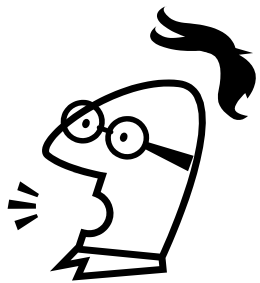
Public name mangling

[CZ] Překlad mangle:
- mandlovat
- rozsekat, roztrhat, rozbít, rozdrtit, těžce poškodit, potlouci, pohmožditi
- *přen.* pokazit, **znetvořit**, **k nepoznání změnit**, překroutit, zkomolit

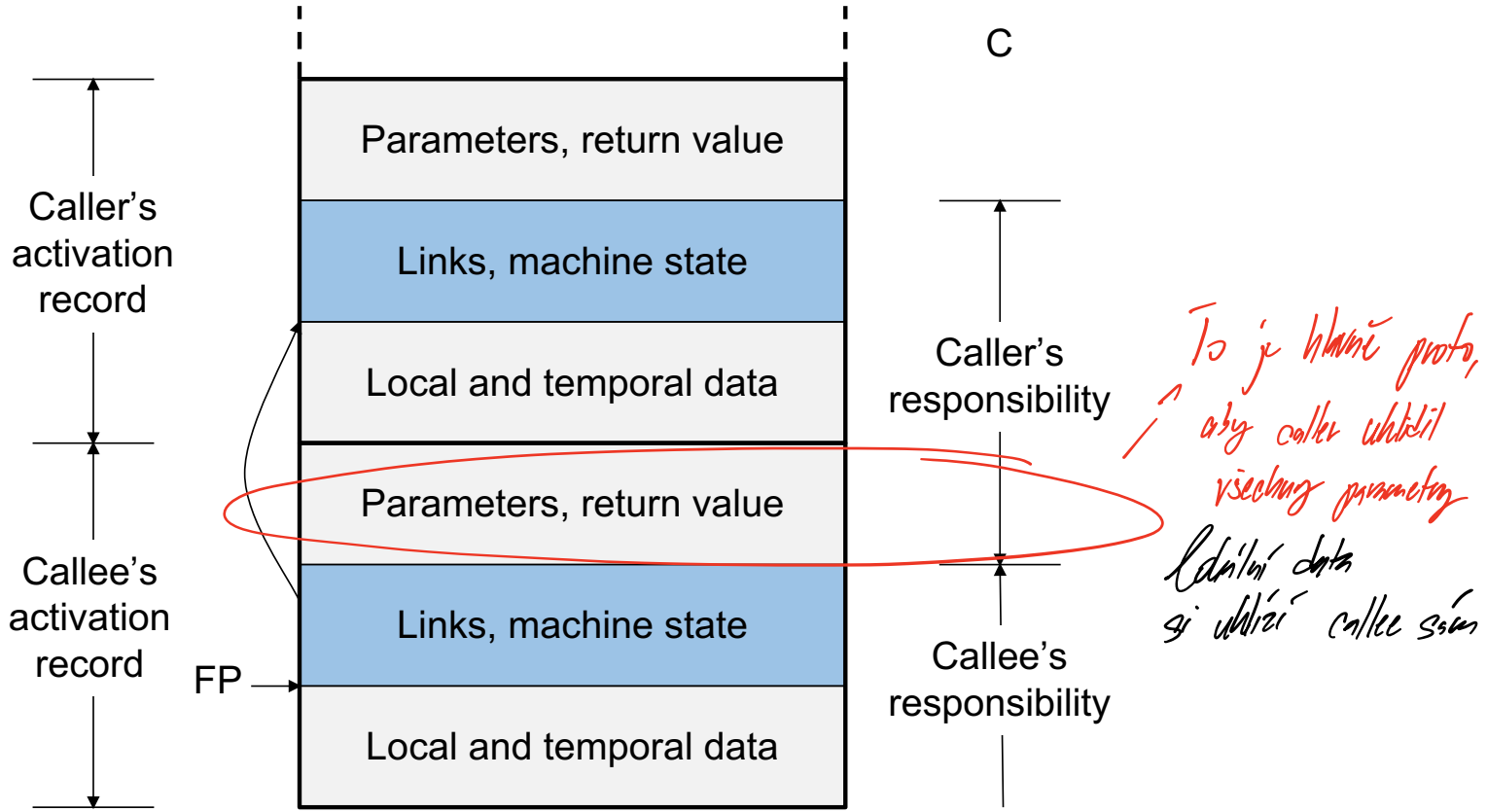
- Examples:

long f1(int i, const char *m, struct s *p)

_f1	MSVC IA-32 C __cdecl
@f1@12	MSVC IA-32 C __fastcall
_f1@12	MSVC IA-32 C __stdcall
?f1@@YAJHPBDPAUs@@@Z	MSVC IA-32 C++
_f1	GCC IA-32 C
__Z2f1iPKcP1s	GCC IA-32 C++
f1	MSVC IA-64 C
?f1@@YAJHPEBDPEAUs@@@Z	MSVC IA-64 C++



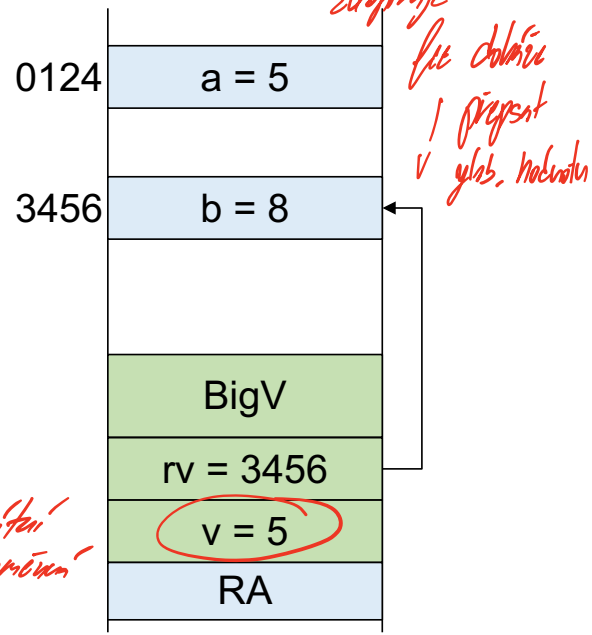
Call/return sequence



Parameter passing

- Call by value
  - Actual parameter is evaluated and the value is passed
  - Input parameters, the parameter is like a local variable
  - C
- Call by reference
  - The caller passes a pointer to the variable
  - Input/output parameters
  - **&** in C++ (hidden pointer)

```
BigV fnc(int v, int &rv);
BigV r = fnc(a, b);
```



Variables

- Named memory holding a value
  - Has a type (statically typed languages)
  - Storage
    - Static data
      - Global variables in C
    - Stack
      - Local variables in C
    - Heap
      - Dynamic memory in C/C#
  - Dictionary (Python, PHP, JavaScript…)
    - Not precisely a storage, it is a dynamic structure (variable name -> some value)

Heap

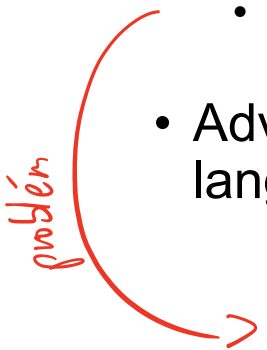
- Storage for dynamic memory
- Allocate
  - Use all features from dynamic memory allocation
    - Free blocks evidence
    - Allocation algorithms
      - Extremely simple and fast incremental allocation
- Deallocate
  - Explicit action in some languages
    - C, C++
  - Automatic deallocation by garbage collection
    - Remove burden and errors
    - Works only with good knowledge of live objects and references

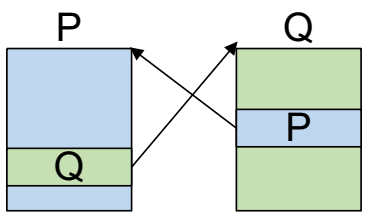
↑ Alokátor je hrozně rychlý s GC, protože se inkrementálně alokuje...

Garbage collection

- Automatic removal of unused memory blocks
  - Advantages
    - No dangling pointers, no double free, no memory leaks, allows heap consolidation and fast allocation
  - Disadvantages
    - Performance impact, even execution stall, unpredictable behavior

- GC strategies
  - Tracing
    - Reachable objects from live objects
  - Reference counting
    - Problems with cycles, space and speed overhead
  - Advanced versions for languages with heavy use





Portability

- Source code portability
  - CPU architecture
    - Different type sizes
      - C, C++
    - Fixed type sizes
      - C#, Java
  - Compiler
    - Different language “flavors”
      - C++ - gcc, msvc, clang, …
    - Use only syntax and library from a language standard
  - OS
    - Different system/library calls
      - Linux, Windows
    - Sometimes easy
      - BSD sockets

Portability by VM (Není to virtualizace Hyper-V)

- "Binary" portability
  - Old technique for ensuring portability of a code among different HW
  - Used by many "modern" languages
    - Java, C#
  - Compiler translates a source code to the intermediate language
    - Abstract instructions
    - Java: bytecode, C#: CIL (binární formát)
  - Native VM compiled for a given architecture
    - Java: JRE, C#: CLR
  - VM interprets intermediate language in a sandbox

(- chrání program, nemohl šahat nikam mimo
- To ale není rychlé
- Jinak je pomalejší)

Překladač přeloží do abstraktního mezikódu, který je kompatibilní se všemi systémy, pak na každém systému je spouštěč, který spustí abstrakci a interpretuje to na procesor

Solving speed problems

- JIT – překládám přímo na hardwarovou architekturu, vše překládám si nechám na cache. Vždám
  - Just-in-Time ale až to potřebuju, nikoliv předem. Takže třeba první spuštění je pomalejší, pak už
  - Translate intermediate code to the native code on demand už to slušíš
- AOT Distribuce stále v mezikódu, při instalaci udělám dlouhý proces translace a jde
  - Ahead-of-Time to do nativního kódu na procesoru.
  - Translate the whole program in intermediate code to the native code during the installation

Discussion