Lab Class Scientific Computing 2022, WISM454

Adriaan Graas, Week 3

C programming

Functions as contracts

Think of functions of contracts**, not as** implementations

```
double sqrt(double x);
```

I will return an
`x >= 0`

I will give an
`x >= 0`

Contracts, intuitively

- 1. The function is responsible of checking the contract
  - **The program should fail if the caller violates the contract.**
- 2. A good contract is generic
  - **Function works for many cases**
- 3. A good contract is restricted by types
  - **Do not ask for** int **if the function only works for** unsigned int
- 4. A good contract is unambiguous about what it does
  - **Works on input arguments and returns in return values**
  - **Has no unexpected effect elsewhere in the program**

Is this a good or bad contract?

```
/* Writes number to file.
 * Contract: file `name` must exist already. */
void write_to_file(char * name, int x) {
    ...
    if (file_not_exists(name)) {
        printf("Error: file does not exist!");
        exit(0);
    }
    ...
}
```

Is this a good or bad contract?

```
/* Writes number to file.
 * Contract: file `name` must exist already. */
void write_to_file(char * name, int x) {
    ...
    if (cannot access file) {
        printf("Error: file does not exist!");
        exit(0);
    }
    ...
}
```

Better contract!

```c
/* Writes number to file.
 * If file does not exist, returns -1. */
int write_to_file(char * name, int x) {
    ...
    if (cannot access file) {
        return -1;
    }
    ...
}
```

Has this function a good or bad contract?

```c
/* Generates a random number
 * from Student's t distribution
 * using the RANDU LCRNG. */
void draw() {
    struct * LCRNG randu = {...};
    return student_t(next(randu));
}
```

Better already!

```
/* Generates a random number
 * from Student's t distribution. */
void draw_from_student_t(struct * LCRNG rng) {
    return student_t(next(rng));
}
```

What about this?

```
/* Computes x^(1/3) */
double cubic_root(double x) {
    return pow(x, 1/3);
}
```

Utrecht University

What about this?

```c
/* Computes x^(1/3)
 * Requires x >= 0. */
double cubic_root(double x) {
    if (x < 0) {
        printf("Invalid argument: x < 0.\n")
        exit(0);
    }
    return pow(x, 1/3);
}
```

```c
/* Computes x^(1/3)
 * Returns -1 if not x >= 0.*/
double cubic_root(double x) {
    if (x < 0) {
        return -1;
    }
    return pow(x, 1/3);
}
```

Utrecht University

What about this?

```c
/* Computes x^(1/3)
 * Requires x >= 0. */
double cubic_root(double x) {
    if (x < 0) {
        printf("Invalid argument: x < 0.\n")
        exit(0);
    }
    return pow(x, 1/3);
}
```

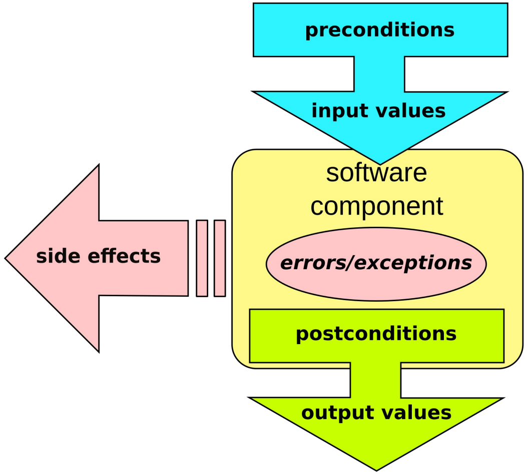
Stricter contract.

```c
/* Computes x^(1/3)
 * Returns -1 if not x >= 0.*/
double cubic_root(double x) {
    if (x < 0) {
        return -1;
    }
    return pow(x, 1/3);
}
```

More flexible contract. Returning -1 may be confusing. The caller could have checked for x < 0 themselves.

Design-by-contract programming

- Preconditions
    - **Conditions that should result in legal and correct behavior.**
    - **If not obvious, should be checked by the function.**
- Postconditions
    - **The guaranteed output.**
- Side-effects
    - **Changing state of something outside the function. This is less transparent.**
- Invariants
    - **Conditions that still hold after the function has been called.**
        - Either on arguments or on some external state



Think about your function design

- Think about functions in your program:
  - **What requirements do my functions have?**
  - **What do I do when requirements are not met?**

```
/* Given an x, and a LCRNG (a,c,m)
 * produces the next x.
 * ...
 */
unsigned next(struct * LCRNG rng, unsigned int x) {
    // exit with error when requirements are not fulfilled
}
```

Compilation

Run-time and compile-time

Run-time vs. compile-time

- It is often necessary to choose if something needs to be a “run-time” or a “compile-time” decision.
  - **Run-time and compile-time refer to moment during the program’s execution, and moment during the program’s compilation.**

Run-time	Compile-time
● Reading input arguments from the terminal ● Reading data out of a file.	● Turning code on or off with comments ● Using macro’s (#define, #if, #else, …) ● Compiler options, such as optimizations ● Constants in the code

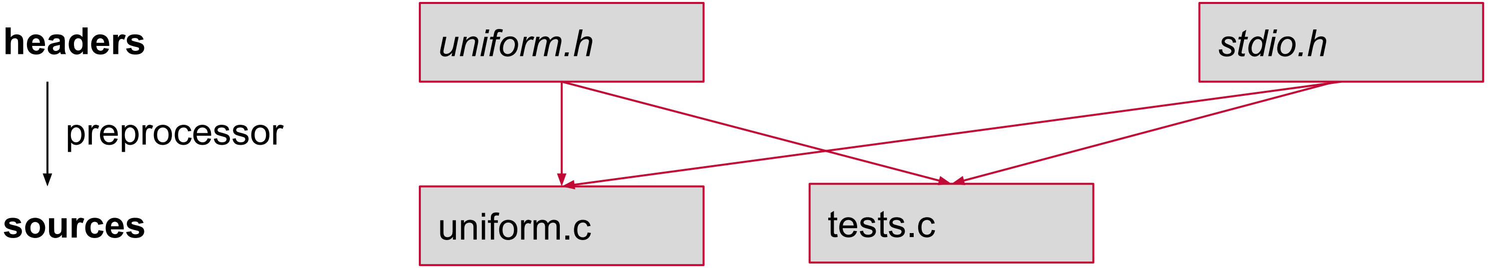
Run-time vs. compile-time

- Typical run-time decisions
  - **Algorithm parameters**
  - **Output requirements**
- Compile-time decisions
  - **Platform choices, such as precision of computation**
  - **Whether or not have debugging statements enabled**
- The compiler can not optimize run–time decisions
- Run-time options require usually a bit more work to implement

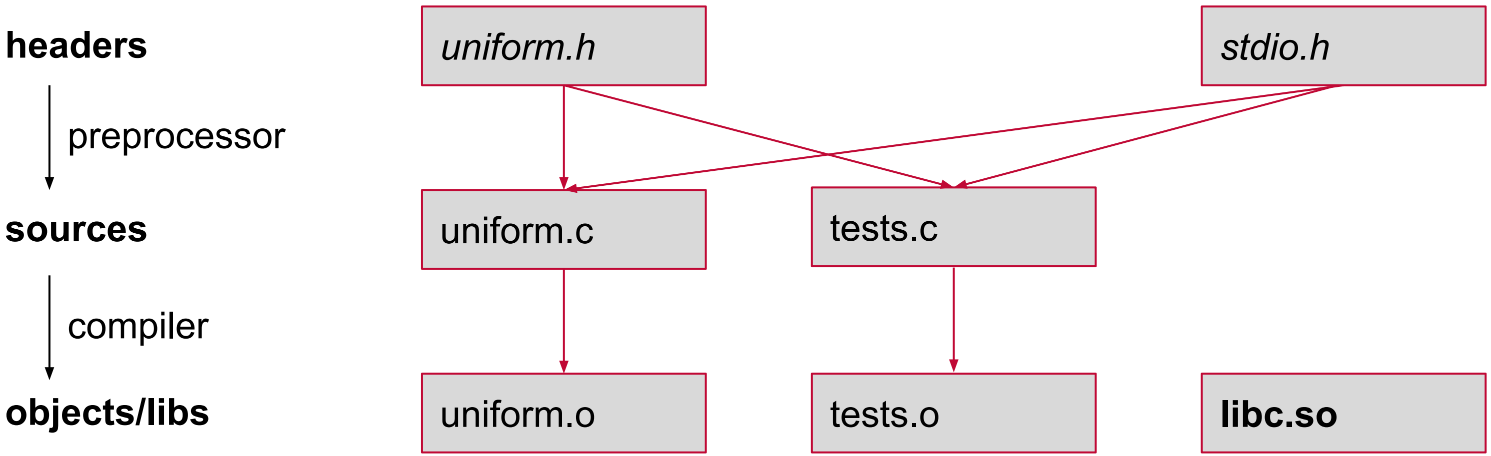
Compilation

Libraries

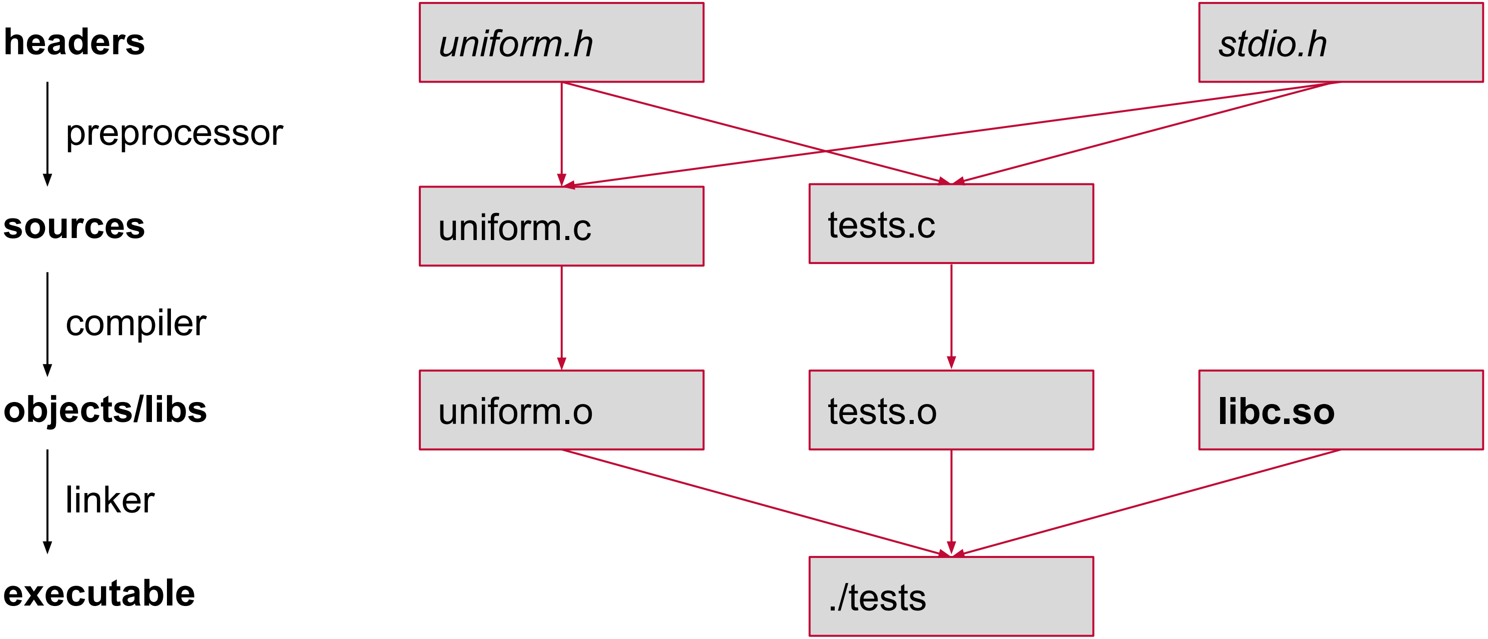
Compilation again



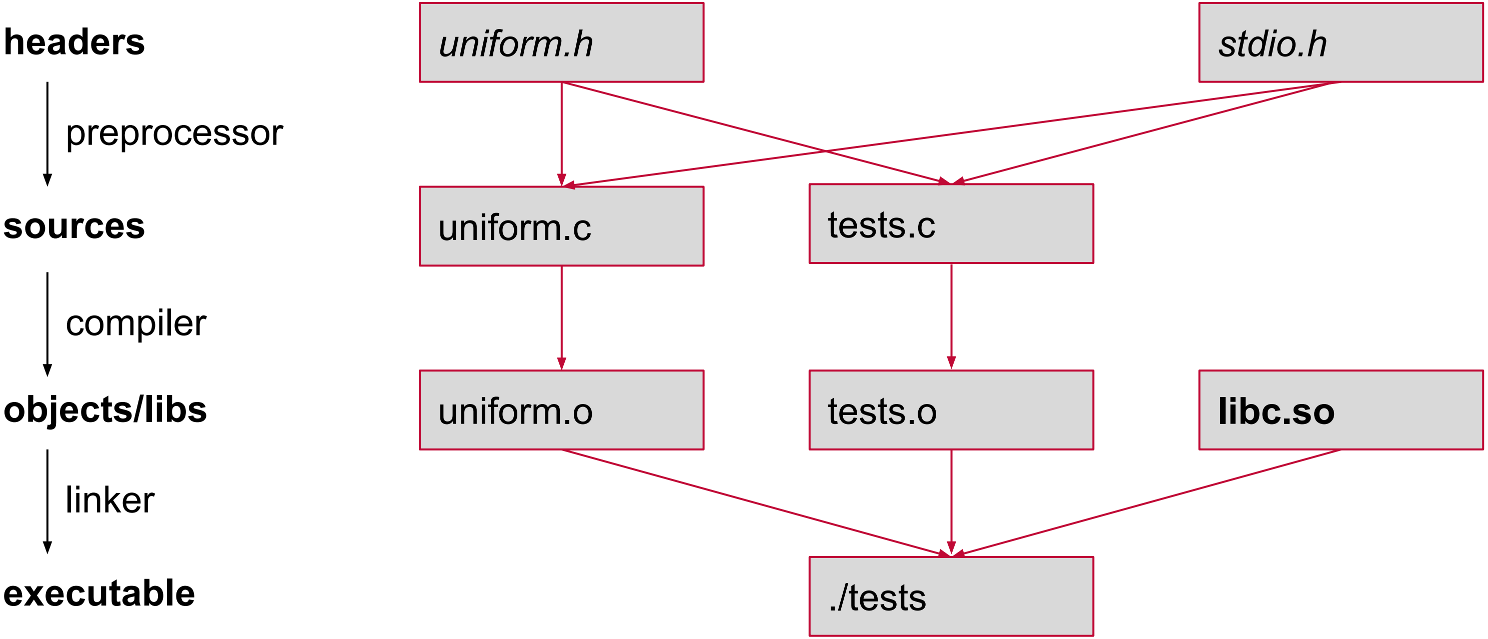
Compilation again



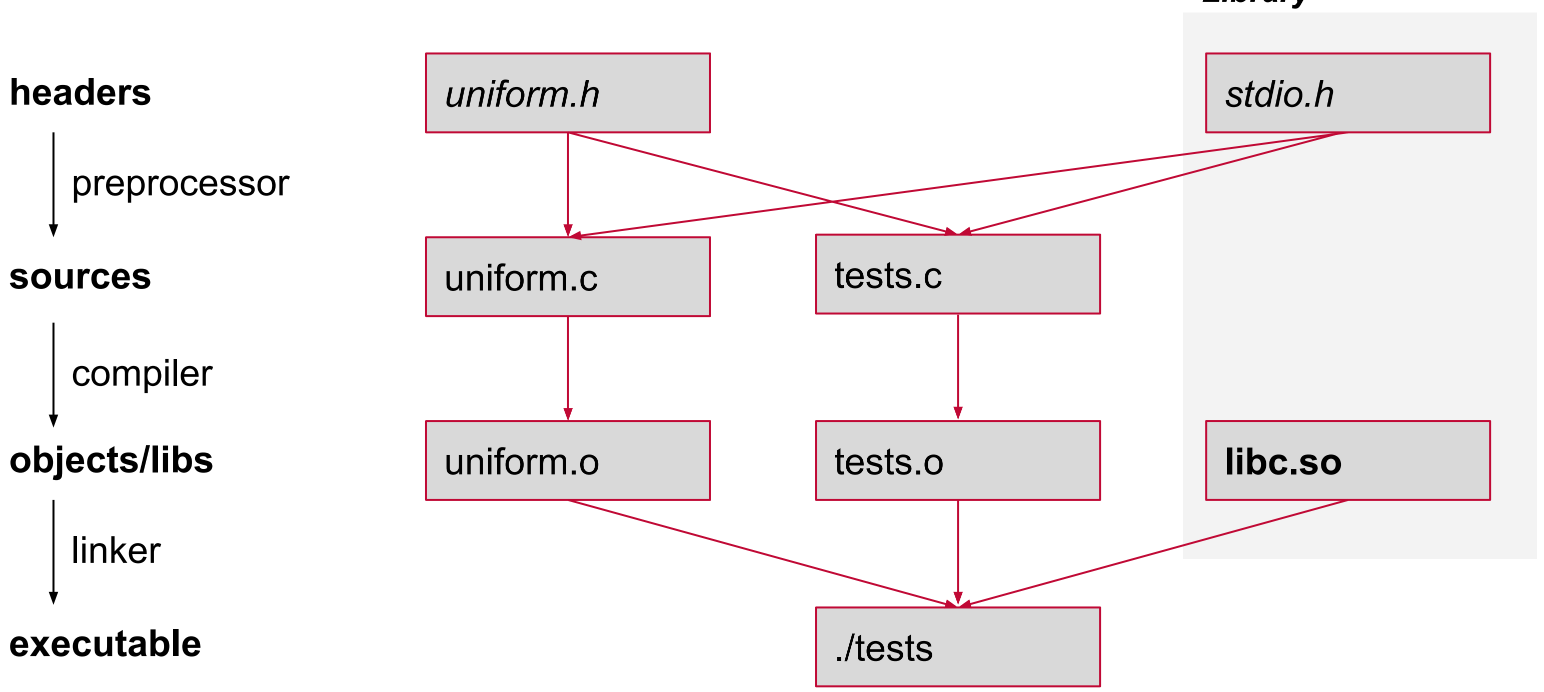
Compilation again



Compilation again



Compilation again



Libraries

- A library is a reusable C/C++ component, consisting of
  - **An archive of object files (.o)**
  - **Header files**
- A library can be build the same way as an executable
  - **A non-executable outcome of the linker**
- Three types of libraries:
  - **Shared libraries (an shared-object file .so + headers)**
    - Loaded into memory when program starts. Also called “dynamic” libraries.
  - **Static libraries (an archive .a file + headers)**
    - Compiled into your own program, similar to your own .o files.
  - **Header-only libraries (no archive)**
    - Compiled into your own program, via #includes.

Installing a (shared) library from an external party

- Step 1: downloading a library
    - **Either as sources: .c files + .h files or as precompiled library: .so file + .h files**
    - **Linux/WSL: precompiled library may be available through package manager**
    - **MacOS: precompiled library may be available through Homebrew, MacPorts or Fink.**
- Step 2: compilation (if downloaded as sources)
    - **Often there is a README or INSTALL file with instructions.**
    - **Almost always a script is provided for compilation: either a Makefile, Automake, or CMake file.**
- Step 3: installation
    - **.so and .h files (and other things such as documentation) are copied into installation directories.**
        - If system user (root) installation typically in system dirs
        - If own user, you may install anywhere, for example in ~/local/

Resolving shared libraries

- Shared libraries (.so) files need to be found when the program is executed.
  - **Option 1: install .so file into system path**
  - **Option 2: tell where .so file is when starting the program**
  - **Option 3: hardcode .so location into the executable (ELF)**
- Option 1
  - **The system automatically searches for .so files in system directories, such as /usr/lib. Nothing needs to be done.**
- Option 2
  - **Execute a program with an environment variable**
    - LD_LIBRARY_PATH=/path/to/lib/dir ./program
- Option 3
  - **Compile the executable with**
    - gcc program.c -o program **-Wl,--rpath -Wl,/path/to/lib/dir**

Utrecht University

Tips

Header tips

- About header files:
  - **Don't forget** #pragma once.
  - **Structs are** declarations**, so belong in headers.**
- Using comments in code:
  - **Comments about how a function works (contract) in headers**

```c
/* Draws a uniform random number in [0, 1). */
double draw(struct UniformDistribution *distr);
```

  - **Comments about implementations in sources.**

Don’t forget: function calls are slow

- Function calls are a big performance overhead, especially for small computations.

```
unsigned next(struct * LCRNG rng, unsigned int x) {
    if (some_condition) {
        next_for_quick(rng, ...);
    } if (...) {
        next_schrage(rng, ...);
    } else ...
}
```

- If you want to use function calls in computations, make sure to compile with -O3, or research about **inline** functions.

This week

- Deadline postponed by one week
- Today / this week:
  - **Try to do exercise 1.7.2 (installing TestU01) today**
  - **This and next week 1.7 and 1.8**