COMPLEX LAB
SYSTEMS PROGRAMMING
DAY 1: GETTING STARTED WITH THE TOOLS

TILL SMEJKAL
COVID-19 Hygiene

• avoid clusters of people waiting to enter the room
• please wear a mask in the hallway toward the room
• once in the room you may take it off
• observe the signage on the doors
• keep distance between work spaces
Course Goals

- low-level systems programming
- advanced C++ skills
- basic multi-threaded programming
- intimate tools knowledge
- hands-on experience
- good preparation for practical courses
Earning Credits

- you can get credits within the modules INF-MA-PR, INF-E-4, and DSE
- 4 week-hour complex lab
- solve practical assignments after the workshop
Duty Roster

• start at 9.30 AM each morning
• end at latest 3.30 PM
• lunch break
• additional breaks on demand
• ask questions early and often
• feedback is very welcome
Topics

• Day 1: Getting Started with the Tools
• Day 2: Multithreading
• Day 3: Debugging Techniques
• Day 4: Inline Assembler
• Day 5: Underneath POSIX
• Day 6: C++ Basics & Details
• Day 7: Rust for Systems
Today’s Agenda

• C/C++ basics
• how to program without an IDE
• using just a Linux shell and basic tools
• dissecting a compiler invocation
• automating with make
• understanding compiler warnings and errors
Exercise 1: First Steps

- create a directory where you will file all course material
- create a subdirectory in it named day1
- in there, create a subdirectory named exercise1
- in this subdirectory, create a file hello.c using a text editor and enter the following code:
  ```c
  int main(void)
  {
      printf("Hello World\n");
  }
  ```
- indicate when you are done
Exercise 1: First Steps

• change into the directory `exercise1` and run `gcc hello.c`
• run the created file
• What does the warning mean?
• edit `hello.c` to fix the warning
• recompile and run again
• change compiler command to create an executable named `hello`
Exercise 2: Arguments

• change hello to take command line arguments
  • hint: change main to
    ```c
    int main(int argc, char *argv[])
    {
        ...
    }
    ```

• print the first argument after the „Hello World“ default text

• make sure to check the number of arguments (`argc`) before accessing the `argv` array
Exercise 2: Format Strings

- the \% is special in printf strings
- placeholder where succeeding parameters are inserted
  - \%s C-string
  - \%c single character
  - \%d signed decimal
  - \%u unsigned decimal
  - \%p pointer
- don’t do this: \texttt{printf(argv[1]);}
- instead, do this: \texttt{printf("\%s\n", argv[1]);}
Exercise 3: Moving to C++

• create a new directory `exercise3` next to `exercise1`
• copy `hello.c` to `exercise3/hello.cc` and open `hello.cc` in your editor
• convert the code to C++
  • use `std::cout` instead of `printf`
  • include `<iostream>` instead of `<stdio.h>`
• compile the file:
  `gcc -Wall -o hello hello.cc`
Exercise 4: Dissecting g++

• pre-process
  
g++ -E -o hello.i hello.cc

• compile
  
g++ -S -g -o hello.s hello.i

• assemble
  
g++ -c -g -o hello.o hello.s

• link
  
g++ -o hello hello.o
Exercise 4: Dissecting g++

• compare object file of C++ source to object file of C source
• check size of executable hello
• check output of nm hello
• call strip hello and check size of hello and nm-output again
Making Friends with make

- `make` conditionally runs shell commands
- often used for build systems, can do a lot more
- automatically determines, which parts of a program need to be recompiled
- speeds up development and prevents forgotten recompiles
- a **Makefile** is a list of rules
  
  target: prerequisites
  
  commands
- by default, `make` executes the first rule of **Makefile**, traditionally using target name **all**
Exercise 5: Using make

- delete the `hello` binary
- write a `Makefile` to create `hello` from `hello.cc`
- call `make` twice and make sure it does not recompile
  - hint: `make` only executes a target’s commands, if the target does not exist or any of the prerequisites is newer
Exercise 5: Using make

- modify the Makefile to treat warnings as errors
- Why does make not recompile?
- modify Makefile to fix
Exercise 5: Using make

• create a function `name` without parameters or return value that prints your name
• call that function `name` from the `main` function in the file `hello.cc`
• we don’t use command line arguments any more
• `make` and run `hello`
Exercise 5: Using make

- move the code of the function `name` into an own source file `name.cc`
  - only move the `name` function, `main` stays in `hello.cc`
  - in `hello.cc`, add the line `void name();` instead
- modify `Makefile` to also compile and link `name.cc`
  - create one binary `hello`
- fix the errors and warnings and rerun `make`
Exercise 5: A Possible Solution

SRC = hello.cc name.cc
OBJ = $(SRC:.cc=.o)

hello: $(OBJ)
    g++ -o $@ $+

%.o: %.cc Makefile
    g++ -Wall -Werror -c -o $@ $<
**Header Files**

- **function declarations** make a function and its signature known within a scope

  ```
  void name();
  ```

- **function definitions** define what is done whenever the function is invoked

  ```
  void name()
  {
    std::cout << "name" << std::endl;
  }
  ```
Header Files

- declarations provide the interface, definitions the functionality
- header files are used to publish declarations
- the header file is included
  - where the function is used, so the compiler knows about it and can check the signature
  - where the function is defined, to detect mismatches between declaration and definition
Exercise 6: Header Files

• write and use a header file name.hh for the function name

• What is the difference between
  
  #include <name.hh>
  and
  
  #include "name.hh"
Exercise 7: Inline Functions

• for very small helper functions, the function call overhead can be avoided by inlining
• make the `name` function an inline function by moving its definition from `name.cc` to `name.hh`
  • hint: prepend the definition with the `inline` keyword
• What happens, if `hello.cc` includes `name.hh` more than once?

• note: this is a sidetrack, we will come back to the un-inlined version after this exercise
Exercise 8: More make Magic

• add a clean rule to remove generated files
• use dependencies to enable recompiles on header changes
  • find the `g++` option to generate a dependency file from a source file
  • extend `Makefile` to generate dependency files
• use them in the `Makefile`
Exercise 8: A Possible Solution

SRC = hello.cc name.cc
OBJ = $(SRC:.cc=.o)
DEP = $(SRC:.cc=.d)

hello: $(OBJ)
    g++ -o $@ $+

%.o: %.cc Makefile
    g++ -MMD -Wall -Werror -c -o $@ $<

clean:
    rm -f $(OBJ) $(DEP) hello

-include $(DEP)
Libraries

- common platform functions are used by virtually every program
- code is packaged into libraries
- static and dynamic libraries
- static libraries
  - are just archives of object files
  - are linked with your own object files into a binary at compile time
  - not relevant at runtime
  - are created with `ar`
  - a symbol index is added with `ranlib`
Exercise 9: Static Library

• create a new directory `exercise9`
• copy your final `hello.cc, name.cc, name.hh` and `Makefile` there
• turn `name.cc` into a static library `libname.a`
  • bonus points for implementing recursive make
  • create a subdirectory `lib` for `name.*`
• create a `Makefile` in that subdirectory to create the static library
• modify the existing `Makefile` to also build in the `lib` subdirectory
Exercise 9: Solution Snippet 2

SRC = hello.cc
LIB = libname.a

hello: hello.o $(LIB)
g++ -o $@ $+

$(LIB): name.o
    ar -cr $@ $+
    ranlib $@

%.o: %.cc Makefile
    g++ -Wall -Werror -c -o $@ $<
Dynamic Libraries

• linked in two stages
  • at compile time, the linker only verifies that all symbols are available
  • at runtime, the dynamic loader
    • checks, what libraries the executable needs
    • loads them into memory
    • attaches them to the executable

• advantages:
  saves disk space and memory due to sharing

• disadvantage:
  longer application startup time
Exercise 10: Dynamic Library

• turn \textit{libname.a} into a dynamic library \textit{libname.so}

• hint: \texttt{g++ -shared} might be interesting to you
  • use \texttt{-dynamiclib} on macOS

• run \texttt{ldd} on your dynamically linked \texttt{hello} binary
Source Code Management

• developers on large projects need
  • change tracking
  • change synchronization and merging
  • tagging and branching
• even on your own, small projects you might want a history of changes with easy rollback
• then a SCM system is for you
  • subversion (svn)
  • Git
• basic operations: checkout, update, commit
Exercise 11: SCM

• change into the day1 directory and checkout the subversion repository to get the exercise11 directory:
  
  `svn co http://svn.inf.tu-dresden.de/repos/advsysprog/day1/exercise11`

• call `make` in that directory
• let’s walk through the warnings and errors
• when you are done, review your changes with `svn diff`
Recap

- first steps with C and C++
- learned what a compiler does
- how to use header files
- static and dynamic libraries
- automating build commands with make
- subversion source code management
- tools: file, nm, objdump, strip, ldd