Computational Foundations I

Martin Werner

Nov 22, 2022

CONTENTS

Download as PDF

Computational Foundations is a two-semester course in the aerospace bachelor and aims at laying out fundamentals of working with computers including aspects from theoretical computer science, computer engineering, and programming selected for their relevance in aerospace engineering.

Material

- Lecture 1: Basics / States / Operating Systems / Command Line
  - *The Disk Operating System (DOS)*
  - *Microsoft Windows*
  - *Unix and Linux*
- Lecture 2: Imperative Programming
  - *Imperative Programming*
  - *C++ - First Steps*
- Lecture 3: Imperative Programming II
  - *Tutorial: Niki the Robot*
  - `Script Algorithms`
  - `Slides Recursion`
  - `Slides Algorithms`
  - `Slides C/C++`
  - `Solution to Riemann Integral`
  - `Whiteboard Complexity`
  - *A Selection of Introductory Programming Challenges*
  - *Note that pointers are not yet taught.*
- Lecture 4: Loops and Recursion
  - *A Set of Loop and Recursive Programming Tasks*
  - `Whiteboard Riemann Integral`
  - *Learning Tasks*

Time Plan

This lecture is organized into thirteen units out of which you can influence quite a few depending on the previous knowledge in the group and the interest.

First the dates in 2022/23:

Date	Topic
18.10.2022	Lecture
25.10.2022	Lecture
01.11.2022	Public Holiday
08.11.2022	Lecture
15.11.2022	No Lecture (TUM regulation)
22.11.2022	Lecture
29.11.2022	Lecture
06.12.2022	Lecture
13.12.2022	Lecture
20.12.2022	Lecture
	Christmas Break
10.01.2023	Lecture
17.01.2023	Lecture
24.01.2023	Lecture
31.01.2023	Lecture
07.02.2023	Lecture
———	———————

Part I

Basic Knowledge

CHAPTER

ONE

THE DISK OPERATING SYSTEM (DOS)

In order to fully understand Microsoft Windows and in order to get to advanced usage capabilities, it is unavoidable to understand, how Microsoft Windows has emerged. In the old days of computing, a company named Microsoft introduced an operating system known as Disk Operating System (DOS) which was used to run most personal computers in these days. From a user perspective, this operating system did all the work of starting up the computer and configuring hardware and running programs. Therefore, a command line was designed and equipped with programs and technology to support basic computing tasks.

MS Dos was a single tasking operating system (except TSRs, which have been small programs that were able to run in the background). This dictates the logic under which interaction with MS Dos is cut into sequential steps. In a nutshell, the computer tells the user that it waits for an instruction by showing a command prompt.

```
C:\>
```

The user is then supposed to enter a command which then runs a program or a builtin instruction. Only four types of information are available to both the user and programs:

- The current working directory (CWD) which is a combination of a drive and a folder relative to which all file operations are performed
- A set of environment variables, such as the %PATH% variable
- The name of the program and a space-separated list of arguments to the command

In this context, the DOS provided a set of commands for working with these states and software vendors could provide additional programs. While DOS can be considered history, all current Windows versions include a Command Prompt feature, which provides a DOS-like command line to perform tasks on Windows computers. For an overview, we list a few DOS commands and ask you to explore them yourself on a Windows computer (or in FreeDOS in a virtual machine like Virtual Box).

Each command starts with the name of the command or program and a set of arguments where arguments starting with a "/" are considered switches that just influence the behavior. Many file-oriented commands allow you to use wildcards. A wildcard in a string matches zero or more character (*) or exactly one unspecified character (?). For more clarity:

- stat*.bat would match status.bat as well as stat.bat
- data?.dat would match data0.dat, but not data10.dat

Basic programs

- <drive letter>: to change the drive
- CD to change the directory
- DIR to list a directory, consider switches /P and /S which change the behavior
- MD to create a directory
- RD to remove a directory (only if empty)

- TREE shows all files below the current working directory
- ATTRIB show and modify attributes like write protection on files
- COPY is used to copy files
- DEL deletes files (synonymous with ERASE)
- EDIT provides a simple editor (EDLIN before MS DOS 6.0)
- FIND searches for a string in a file
- MORE pages a file to the screen
- MOVE moves a file to a different location
- PRINT is used to print a file
- REPLACE works like copy but replaces the file in the target
- TYPE outputs the whole content of the given file
- XCOPY extends COPY to be able to copy whole directories and trees
- CLS clears the screen
- DATE shows and modifies the date
- TIME shows and modifies the time
- ECHO is used to control whether commands are shown or not (mainly in batch files)
- FDISK is used to set up hard disks (partitions, etc.)
- FORMAT organizes a file system on floppy disks or hard disk partitions
- HELP shows help for a dos command (use it in the tutorial!)
- SET shows configuration information and environment variables and modifies them
- VER shows the version of DOS in use

With these commands, it is possible to organize a computer quite nicely yet remaining simple and self-explaining. Another interesting aspect about DOS which is visible in modern versions of Windows is the fact that file names were heavily restricted in early versions allowing 8 characters for the file name and 3 characters for the file name extension separated with a dot. Extensions have always been used to mark the type of file, for example article.txt was a text file suitable for the edit program while word.exe was a program known as Word which could be run by writing the command word (without the extension) when in the same directory or when the directory was mentioned in the PATH variable.

Note: Assignment 1:

- Install VirtualBox and run FreeDOS inside
- Using the FreeDOS command line, create a folder structure representing Germanys federal structure. Start by creating a folder Germany, within this folder, create one for each state.
- In each state, create a file capital.txt and write into it only the name of the capital of the state in one line (be sure to end the line)
- Show the tree (and submit as a solution) using the command line
- (Advanced) Output all member states
- Learn about the redirection of output using the > pipe

- Look around on your Windows PC if you have one. Where is your data stored technically, where is Documents located? Where are Downloads? Try to find out about this by just running the "Command Prompt" or "Eingabeaufforderung" in German language.
- Create a file on the Desktop of your Windows Computer, use the extension .txt and write Hello Windows into it. Then open the file with the Windows GUI which should spill up your favourite editor.

DOS used drive letters A, B, C, etc. to distinguish different disks. In the very early days, one typically had no hard drive in an MS DOS computer, but one floppy disk. This disk was known as A and all the life was taking place in A:>. A bit later, many computers came up with a second floppy drive. Now, the drive A was used to start the operating system (DOS) and provide commands, while B:> was used to store data. As floppy disks are sometimes usable (when a valid disk has been inserted) and sometimes unusable, both letters are reserved up to today. Typically, the first drive letter assigned to hard drives or other modern devices is, therefore, C:>. In almost all versions of Windows today, Windows is installed on a drive with letter C. If the main drive has more partitions, drive letters are used sequentially, such that a two-partition setup often has a drive D:>. On many other computers, D: already refers to some CD drive or USB stick. In a nutshell, drive letters are assigned along the alphabet and as the first two letters are reserved for floppy drives, the first letter in every-day use is C. As a consequence of this unknown dynamics, it has become a tradition to map the first network drive with the letter Z and to continue backwards (if you are in a network-enabled environment). Furthermore, some companies have started to use a drive called H like "Home" for the home drive of a user.

Note: Assignment Two: Install Windows (at least once)

Windows Installation Tutorial. When you start working with Windows a lot, you will face the situation that your computer is not working properly anymore. In this tutorial (accessible only for people with a valid Windows license), we will install Windows into a virtual machine to train the procedure. In order to help you for your future, we ask you to do this on a virtual disk of 20 GB (use a FCOW disk to save space) which you partition into three disks: Disk C (the main Windows disk) shall use 10 GB, while a drive D of 5 GB and a drive E of 5 GB shall be available as well. Therefore, you can use the partitioning tool part of the Windows installer. We will use Windows Education 11, but the procedure is not much different for any (unmodified) Windows.

CHAPTER

TWO

AUTOMATION IN WINDOWS

Since the beginning (including MS DOS), the builtin functionality can be used to automate routine tasks to some extent. In Windows, batch files are being used. In their simplest form, batch files just list a sequence of commands to be executed one after another. But batch files can have more aspects such as looping over files, asking for user input and other advanced patterns. As many of our students will be exposed to a Windows with no access to advanced scripting software, it is worth knowing some basic aspects of how to write batch files on Windows.

As said, a batch file is just a text file in ASCII format and each row of the file represents a command which you could as well type into the command prompt. However, it gets more interesting if one realizes the following three functionalities:

- Disabling the output
- Looping over files
- Using arguments given to the batch file

The first aspect is simple, but important: in basic batch files, each command is first output before the command is executed and the output of the execution is output. This means that it is a bit tricky to have concise output from a batch script which is really useful for the user. In a batch file, the output of a command can be suppressed by prefixing it with @ why it is good practice to start all batch files with a line

```
\@echo off
```

The output of this line is suppressed (it would typically output "ECHO is disabled") and the output of commands is disabled for the remainder of the script.

Now, it is very common that batch files are used to automate annoying commands to avoid typing and thereby avoiding typos as well. Imagine you are working on satellite images and you have downloaded 1,000 scenes from Sentinel 2. As you want to create a web map, you decided to reproject all of these files from their own projection into a WebMercator projection (EPSG 90009001). You figured out the right parameters for letting gdal_warp utility doing your work, for example

```
gdal\_warp \<scene\> -t\_srs ... (please figure it out yourself!)
```

Then, you can use a batch file like this one:

```
\@echo off
for %%f in (\*.tif) do (
gdal\_warp \<magic parameters here\> %%\~nf output/%%\~nf
)
```

This file takes all files with extension tif in the current directory and gives them as input to our magic gdal_warp command while putting the result into the same file, but in a directory output. Hence, be sure to have this directory created (or make the creation of this directory part of the batch script itself). In this way, you can now have a lot of coffee or go to bed while your computer is working through your archive of data.

Note: Assignment Three: Writing Batch Files

Create a batch file which reproduces the result of Assigment One. A batch file in Microsoft DOS and Windows is a simple ASCII text file with an extension of .BAT. Those files can be run from the command prompt just like EXE files by giving their name without extension. Use the @ECHO OFF as the first line to suppress the output of the individual programs. Explore on your own using the Internet, how one can get input from the user into a variable, how one can loop over files running a certain command for each file.

Note: Assignment Four: Learn some Latin

We will write a batch file program to train latin vocabulary. Therefore, we create a file with the latin name containing the translation (followed by a newline). Then, try to write a script that first selects a file at random (or any other way) and asks (by showing the filename representing the latin vocabulary). Then ask the user to input and (!maybe!) try to compare the user input with the file contents at least roughly. Note that this can be really tricky in BATCH programming, so any approximate solution is appreciated.

*Just some tips: To make it less complex, the solution to this task was not comparing the results, but rather showing the user input and the translation from the file. Further, we create two Batch files: one which loops over all files and another one which is called with the filename and presents a short dialog. *

Note: Assignment Five: Reproject Sentinel Scenes

Download a handful of Sentinel scenes from different locations, install GDAL and use the gdal_warp command line utility to bring all the files into the same projection.

CHAPTER

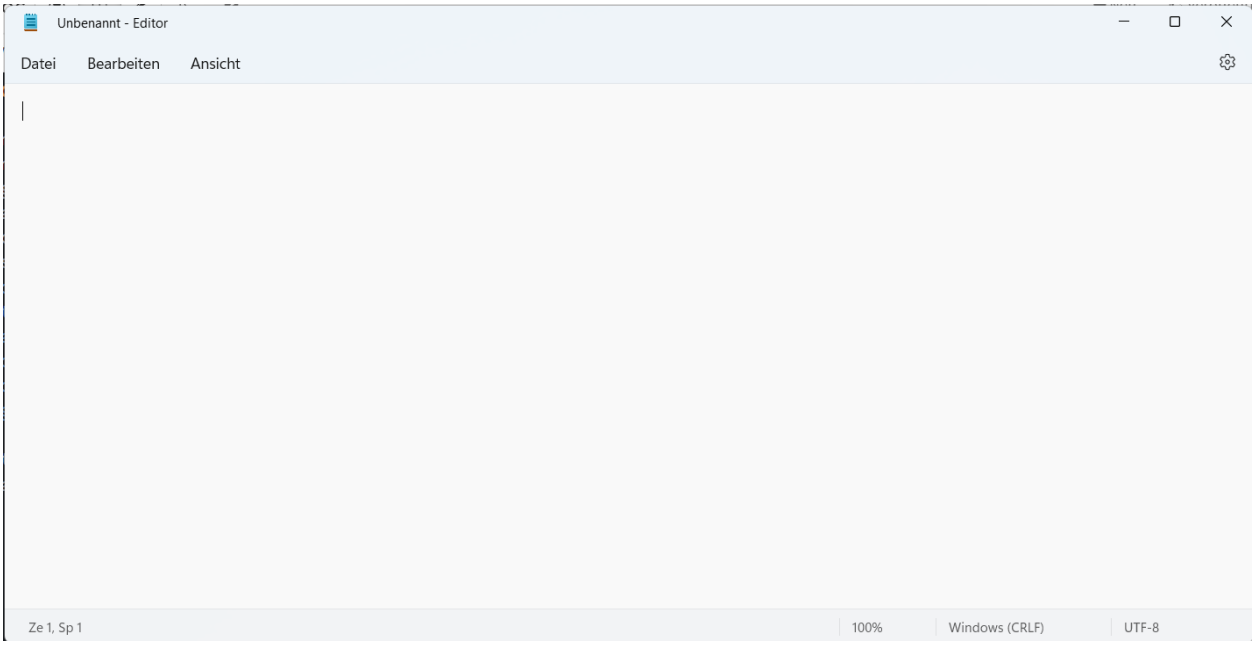
THREE

MICROSOFT WINDOWS

The Windows operating system has then emerged as a graphical user interface (GUI) on top of DOS. In its early versions, it was a graphical file manager, but the most important innovation was available with Windows 3.x, namely, an interaction scheme in which the graphical screen is subdivided into rectangular Windows with the following properties

- Windows can overlap each other
- Exactly one Window is active
- The active Window is in the front (fully visible)
- Windows can be resized and moved on the screen
- Windows provide buttons for minimizing, maximizing and closing the Window

In fact, a Window today looks like this editor window from the integrated Windows Editor Notepad:



Windows typically have more standardized aspects such as the following ones. Each Window has a **Title** line which contains an optional Icon (which is a menu) followed by a text (the Window title) and the three buttons on the right for minimizing, maximizing and closing the Window. Below the Title, a **menu bar** is located in which multiple text fields are displayed. When you click on them, a window is opened and can be navigated to send a command to the application. The bottom line (optional) is known as a status line and typically contains a few controls, at least one text. The remainder is called the Client Area and is used by the application.

In order to give you a more in-depth understanding, let us look at the principle of writing programs for MS Windows. A normal program is started, this one asks the Operating System to open a window with certain properties (size, location, etc.). One of these properties is a function which is then called by the operating system with events such as the following ones:

- WM_CREATE: Is sent when the Window is created
- WM_DESTROY: The window is destroyed
- WM_MOVE: The location has changed
- WM_SIZE: The size has changed
- WM_ACTIVATE: The window has become active
- WM_QUIT: The X has been clicked (or an equivalent hotkey was activated)
- WM_PAINT: Draw the client area (however this is done)

Without expecting everyone to understand completely at the first time you read this article, here is a complete Windows API example drawing a rectangle. It is written in the C language we will learn, and serves as a primary example. It has been made available at Github and is referenced by the MSDN as well MSDN Article on WM_PAINT

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int nCmdShow)
{

    // Register the window class.
    const wchar_t CLASS_NAME[]  = L"Sample Window Class";

    WNDCLASS wc = { };

    wc.lpfnWndProc   = WindowProc;
    wc.hInstance     = hInstance;
    wc.lpszClassName = CLASS_NAME;

    RegisterClass(&wc);

    // Create the window.

    HWND hwnd = CreateWindowEx(
        0,                              // Optional window styles.
        CLASS_NAME,                     // Window class
        L"Learn to Program Windows",    // Window text
        WS_OVERLAPPEDWINDOW,            // Window style

        // Size and position
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

        NULL,       // Parent window
        NULL,       // Menu
        hInstance,  // Instance handle
        NULL        // Additional application data
```

(continues on next page)

(continued from previous page)

```
        );

    if (hwnd == NULL)
    {
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);

    // Run the message loop.
    MSG msg = { };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;

    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            // All painting occurs here, between BeginPaint and EndPaint.
            FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));
            EndPaint(hwnd, &ps);
        }
        return 0;
    }

    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

This very clearly illustrates what is happening: The program creates a new Window which refers to a function in our program (`WindowProc`) and this function is called by ourselves with all messages we can Peek and Dispatch in our main program.

The nature of an event-driven system is that after a lot of initialization, the main progress of the system follows an event-driven nature: an ordered sequence of information (called events) will dictate the behaviour.

By the way, this is also the core reason why sometimes windows are hanging and not reacting: This happens, when messages like WM_PAINT or WM_CLOSE are not delivered, because the event loop is not running properly. Windows typically blurs out the Window and shows a dialog about this problem.

As a consequence, good programs will have to make sure that all messages are quickly handled maybe by making other parts of the program proceed asynchronously, for example in a thread.

It is interesting to see how a concurrent impression has been created by using the graphical user interface, which is

inherently non-parallel: there is only one active program at a time.

CHAPTER

FOUR

UNIX, LINUX, AND POSIX

The Linux family of operating systems has been improving in importance in the last decades. The project started as an illustrative implementation of a simple UNIX kernel on top of the 80386 computer and is now the leading operating system in terms of global impact.

In order to understand Linux, one has to look back into the history as well. Linux is modeled after Unix which is a family of operating systems developed for mainframes. Such computers have been very expensive and available long before the personal computer was available, but a single UNIX mainframe was used by many members of a company.

Hence, topics like user management, access control and parallel execution of different things (for different users) have been at the heart of Unix development, while the simplicity (despite all risks) of the DOS immediate mode was not accessible.

At this point, before looking into the guiding principles of Unix and Linux, we can mention a name here: the basic Unix was mainly developed by Ken Thompson and Dennis Ritchie who has also had huge impact on the development and success of the C programming language. In fact, Unix has been implemented mainly in assembler (a rather raw machine language with almost no abstractions), but later translated in large parts into the C language. And this is one of the earliest predecessors of the programming language we will focus on.

Furthermore, a few principles have been fixed in the early days and have proven successful enough not to be changed that much in the years to come. This group of decisions is referred to as the Unix philosophy, which we will revisit when talking about software and programming.

A famous formulation of the Unix philosophy is due to Douglas McIllroy:

- Write programs such that they do only one thing and they do it well
- Write programs such that they can work together
- Write programs such that they work on textual streams as this is the universal interface

This is often oversimplified to

> Do only a single thing at a time and do it well

Or (more or less the same) as the KISS principle:

> Keep it simple stupid.

There is a lot to these aspects and we will learn a lot about it. More down to the point, Mike Gancarz gives the following list

- Small is beautiful.
- Make each program do one thing well.
- Build a prototype as soon as possible.
- Choose portability over efficiency.
- Store data in flat text files.

- Use software leverage to your advantage.
- Use shell scripts to increase leverage and portability.
- Avoid captive user interfaces.
- Make every program a filter.

This will guide our gentle introduction to Linux in the sequel.

4.1 POSIX

Unix has led to a joint understanding of how computers should work and it has been very successful. But it has also quickly become an area of debate (see Unix Wars). In order to unify and converge the market, a new standard has been carefully designed and developed (ISO/IEC/IEEE 9945) under the name POSIX.

This standar describes clearly and independent from a concrete implementation defintions of terms, the system interface (concretely in C language including header files), and the command line interpreter and a list of tools.

When learning C (or any other modern language), one will often touch exactly this standard even in a non-Unixoid environment (like Windows).

We will now skip over the definitions and the header files, as they naturally appear when learning programming, but jump right ahead to the list of mandatory utilities that you can expect any unix-like operating system to provide:

For a complete list, please refer to https://pubs.opengroup.org/onlinepubs/9699919799/idx/utilities.html

For our first interaction with these systems, we selected

- ar (nowadays often tar for tape archiver)
- at
- awk
- basename
- bc
- cat
- cd
- chgrp
- chmod
- chown
- compress (nowadays, bz2 and gzip)
- cp
- cut
- date
- dd
- df
- diff
- dirname
- du

- echo
- ed
- env
- ex
- expand
- expr
- false
- fg
- file
- find
- fold
- getopts
- grep
- head
- iconv
- id
- jobs
- join
- kill
- ln
- ls
- man
- mkdir
- mkfifo
- more
- mv
- nl
- paste
- patch
- printf
- ps
- pwd
- read
- rm
- rmdir
- sed

- sh (nowadays often bash)
- sleep
- sort
- split
- tail
- tee
- test
- time
- touch
- tr
- true
- ulimit
- umask
- uname
- uniq
- unlink
- vi (nowadays vim)
- wait
- wc
- who
- xargs
- zcat (gzcat, bzcat)

Note: Install a Linux, preferably Debian (with no tasks selected, video follows), and login to the system with the user account you created during installation. Then, inform yourself about the commands using the `man` command.

4.2 Linux as seen by a user

In order to understand Linux, we need to understand the command line version of it. With Linux being a multiuser operating system, our adventure starts with logging in to Linux by giving a username and its associated password. Then, we end up with a command line, most typically running the Bourne Again Shell (bash).

Similar to the DOS command line, the system now waits for your instructions and gives you some state information like the current working directory (PWD on Linux, accessible with the `pwd`comamnd). Again, you can now use the programs given to start working with a Linux computer.

First, we will have to navigate the system and in Linux there is no concept of drive letters. Instead, there is one file system root (/) and from there the journey begins.

We can move around by using cd, we can as well use the special directories . to refer to the current directory and .. to refer to the parent directory in the relevant contexts.

Note: Navigate to the main directory, then from there into the bin directory. This directory traditinoally holds user programs. By using a pipe character, you can make the output of a program becoming the input of another such that `ls | less` will let you page through all programs in the installation.

Navigate to /home. Show all directories using ls. This should now have one directory per user. Enter whoami to find out your user name.

Navigate to /etc, where Linux configuration is held. All programs should keep their configuration there as a plain text file. Look at the file /etc/fstab using an editor of your choice (vi, vim, emacs, nano).

Navigate to /proc. This holds kernel information and files to interact with the kernel. Look at /proc/meminfo, /proc/partitions and /proc/meminfo to find some information about your computer.

4.3 Example: Processing data in a Linux environment

As long as you stick with the Unix principles, Linux provides sufficient tools for 99% of your everyday tasks as a data scientist. No advanced software or programming is required in this context.

In order to illustrate the line of thinking, let us perform a rather simple task: count the words in the definition (RFC) of the HTTP protocol available from RFC

```
curl https://www.rfc-editor.org/rfc/rfc2616.txt > 2616.txt
```

will just download the document. To simplify the remainder, we create a local copy, so be sure to be in a reasonable directory (maybe create one) first.

In order to find the most frequent word, our task is now to break it down into individual words. We do this by relying on the `tr` tool. Read the man page, but we just turn every space into a newline:

```
cat rfc2616.txt | tr " " "\n"
```

In this command, the file is first output to stdout, but this standard output is bound to the standard input of the tr command. This takes every space (first argument) and turns it into a newline. The output is ugly and long (would keep scrolling for quite some time), hence, we can rely on head to see part of it

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | head -10




Network
Working
Group

martin@martin:~/lecture$
```

Okay, this looks nice, but empty lines will be dominating (this happens, because there are a lot of spaces in the document for layouting page numbers). Let us get rid of empty lines as follows (this is tricky, we will discuss this in the tutorial)

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
↪head -10
Network
Working
Group
R.
Fielding
Request
for
Comments:
2616
UC
```

Now, we can start counting the words. The easiest way to do this is to sort the output using `sort` and then use the `uniq -c` command to remove successive equal lines outputting the count of removed lines. Let us again limit the amount of screen usage with head:

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
↪sort |uniq -c | head -10
      1 "
      1 ""
      1 ""%"
      1 "#"
      1 "%
      3 "("
      2 ")"
      1 ")">
     26 "*"
      3 "*",
```

Here, now the first column shows the number of times a string has been seen and we need to find the largest ones. We do this by sorting again, but numerically using `sort -g` and reversing the direction. As we might be unsure whether it works, we can again work on reduced outputs by keeping the head in the command.

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
↪sort |uniq -c | head -10 | sort -rg
     26 "*"
      3 "*",
      3 "("
      2 ")"
      1 ")">
      1 "%
      1 "#"
      1 ""%"
      1 ""
      1 "
```

Now, we are almost there: Let us now really look for the ten most frequent words in RFC 2616:

```
martin@martin:~/lecture$ cat rfc2616.txt |tr " " "\n" | sed '/^[[:space:]]*$/d' |␣
↪sort |uniq -c | sort -rg | head -10
   3532 the
   1579 a
   1349 to
   1298 of
```

(continues on next page)

(continued from previous page)

```
   1006 is
    829 and
    773 in
    661 that
    653 be
    550 for
martin@martin:~/lecture$
```

When you take your time to learn a few (if not all) of the core utilities (GNU coreutils), you can solve almost all problems based on text files considerably faster than with any other environment.

CHAPTER

FIVE

IMPERATIVE PROGRAMMING

Programming is the principles procedure of telling a computer what it is supposed to do and there are various programming styles that can be used to program computers. However, the most widely used and most basic style of thinking of computers is the paradigm of imperative programming.

Imperative programming matches very well the nature of computers being rather dumb and simple machines. In imperative programming, the programmer takes the role of an imperator and provides instructions to the computer in a very precise ordered form. These instructions come from a comparably small set of instructions which the computer must support.

5.1 A Robot Model of Intrinsic Instructions

A very basic model of a computer can be imagined as a small robot that has two functions: turn right by an angle of $\pi/2$ and go one step into the direction currently looking at. When building such a machine, one immediately also designs a programming language which has two operations (we call them intrinsics, because they are the operations really implemented in hardware). Let us give names to them: `move` for moving one step, and `turn` for turning left by $\pi/2$.

Now an imperative program is very much like a shell script: from the language that we have defined, we can create a text file with one intrinsic instruction per line (this form of machine code is typically called assembler).

The program

```
move
turn
turn
move
```

would thus be interpreted as an imperative program like: first move into the direction you are facing, then turn to the left, then turn again to the left, then move again.

So far, we have introduced a few concepts that should be highlighted:

- intrinsic instructions are those steps that a physical computing machine can perform immediately
- Each intrinsic instruction gets a name
- A series of intrinsic instructions can be written into a file which is an imperative program and interpreted as a sequential instruction

Let us reflect a bit more on the situation of such a robot. In a certain sense, the previous program does define very precisely what the robot is supposed to do, but neither in which state (location and orientation) it has been in the beginning nor in which state it is afterwards. In fact, each operation is well-defined local to the robot (we know how the wheels are to be moving), but not well-defined with respect to the robot in the world. Therefore, we would need to give or fix initial conditions.

For a real imperative program, these intiial conditions are the state that is held in the operating system about the program, for example, the current working directory (CWD).

5.2 Procedures

A very typic additional definition in an imperative programming setting is the notion of a procedure. A procedure is an imperative program (e.g., a sequence of instructions) such that this sequence can be referred to as a new operation in the programming language. These are the so-called non-intrinsic operations.

For our robot, the following program

```
turn
turn
turn
```

means to turn left for three times. This, of course mimicks the result of turning right. In many programming languages, there is a way to make this a new instruction of the language called a procedure.

```
proc turnright:
  turn
  turn
  turn
```

Our minimal computer (the robot) with its tiny set of intrinsic instructions (move and rotate) can now be programmed with a third instruction `turnright`

5.3 Ports and Functions

As a next step in the co-evolution of a computer and a programming language, one might want to be able to react on real-world input, that is, something coming from outside the robot. For a robot, we could imagine a sensor that just tells us whether the place we would move to is occupied or not.

That is, we extend our hardware with something we will call a port as it brings external information into the system. And we will extend the programming language with an intrinsic function to model this port.

A function is a procedure in the sense that it can be run and that it can be built together from other instructions and functions, but it **returns a value.** For the case of the port letting us experience the next location being occupied, this return value can have two states: occupied or not occupied. Due to the huge impact of George Bool on the behavior of such values (this British mathematician passed away already in 1864 long before digital computers have been realized), any two-valued information in a computer is called a Boolean value or bool for short.

Concretely speaking, a function would now look like `is_empty` and when this is `called` it would turn into the current value associated with the hardware port.

Now, with having ports, we can observe aspects of the surroundings (information exterior to the system itself) and in order to react to them, two aspects are introduced in imperative programming:

- control structures and
- expressions

5.4 Control Structures

As we are still looking for a minimal programming language, we could imagine that our robot is supposed not to crash with the surroudings, so maybe we just need to be able to make instructions like move **conditional** to the value of the port.

This control structure is often nown as if .. then .. else .. More concretely, we enable the following snippet of source code:

```
if is_empty then move else turn
```

This program would now always first check the condition of the if (run the intrinsic function) and if move is possible, it would move and if not, it would turn in the hope that we can move afterwards.

To complete this exposition of minimal control structures, there is another common way to use boolean information in imperative programs: to control the repeated execution of something. To this end, we introduce a loop called while:

```
while is_empty do move done
```

This means: as long as it is possible to move, continue moving. Note that by introducing a Boolean function, we would typically also introduce the two Boolean values as constants for our programs. They are typically referred to as true and false, hence, we can also write

```
while true do
  while is_empty do
    move
  done
  turn
done
```

5.5 Scopes

Another concept we silently introduced in the previous example is the idea of scopes. It is so common that within a conditional branch or a loop multiple instructions need to be placed that we try to avoid to introduce a function for it. Because if we had to, we would make things into functions that are used only once. As an alternative, a **scope** is introduced which is a sequence of instructions that is taking the role of a single instruction like the loop body (what to do as long as a condition is true) or the two branches of a conditional information (what to do when the value is true, what to do otherwise).

Scopes have varying notations: sometimes with braces (C++, Java)

```
{
move
turn
}
```

sometimes with indentation (Python)

```
while true:
  move
  turn
```

sometimes with barrier words (Bash)

```
while true do
move
turn
done
```

sometimes with round brackets (DOS/Windows BAT files)

```
(
)
```

But they all serve the same purpose of quickly and locally (on the screen in the right location) bundling together instructions.

5.6 Memory

Now, this very small robot can be extended further to interact with a physical world. As you may have noticed, the robot is currently limited to a grid of points it can reach as we only implement movement by $\pi/2$. Assume now, we give the robot some small things it can deposit into the grid cell he is currently in, and then also sense, and maybe even pick up. That is, we extend the robot device with three functions represented by three intrinsics:

- a function deposit to put something into the current cell. Let us assume that it fails if the cell is already filled with such an item.
- a function pick to clean the current cell
- a function has_item to check if the current cell is filled with an item.

With these three intrinsics, we can write a lot of algorithms and it can be fun. The interesting aspect is that the 2D world provides us with the ability to have a concept like a variable.

5.7 Variables

A variable is an area of memory to hold information together with an interpretation associated with this information. For the 2D robot case, a variable is often such a thing as a stack of items like in the following drawing. The identity of the variable being the column in the grid while the value is the height of the stack.

In all imperative programming languages, the idea is similar: there is an axis of varying identity (think like rows in a table) and each variable typically has a name (just a string) in order to use it. The second axis in programming languages describes together the amount of space that is needed and the interpretation of the space. For example, you can have a 32 bit integer number (reserving 32 bits = 4 bytes of memory) assigned an identity like `i` or the same amount of memory for a 32 bit floating point number `f`.

In high-level languages, it is not uncommon that variables are accessible by name during runtime, in lower level languages, the names are only available during compilation and will be removed (for efficiency) while compiling the source code to a executable.

5.8 An example robot program sketch

Assume we have our robot being on the lowest block of a tower of blocks each with a marker. Let us assume, we want to interpret the height of this tower as an integer number, say `h` like height. Let us assume further, we want to compute the value `2h`. How would we proceed?

5.8.1 Algorithm Design

It is pretty clear and intuitive, how the robot can solve the problem: For each marked block, it creates two marked blocks somewhere else. More concretely, let me lay down a proposal as follows:

- The robot starts on the lowest block of a tower
- The robot walks upwards to the highest block
- The robot picks up the marker (the tower has reduced in height by one)
- The robot goes down as long as markers are there (to the ground floor, so to say)
- The robot goes a step to the right
- THe robot walks up the tower as long as possible going to the first non-marked place
- The robot deposits a marker
- The robot goes up
- The robot deposits a marker
- The robot goes up
- The robot deposits a marker
- The robot goes down as long as possible
- The robot goes left (and is back in the state we want)

This needs to be repeated until we have taken the whole first tower.

Now, with this overview of the algorithm, we continue with an implementation strategy: there are quite a few things that are semantically bounded in the sense that we can precisely and simply describe the state before and after a part of our program. We could for example implement a procedure `climb` and use it twice: once to climb up the left tower and once to climb up the second tower. Which tower to climb will just be based on the current location of the robot.

Our program skeleton grows slowly. In order to have a more narrative programming style, we introduce another idea of programming, namely to put human readable text comments into the source code to illustrate some aspects and further to use such comments to specify the expectations of implementations. Note that some modern programming languages like C++ allow us to have such specifications during compilation in one or another way, but this is a rather new feature and has not yet been voted into many standards. But the topic to keep an eye on is known as contracts.

```
# function climb
# precondition: the robot looks up
# invariant: the robot keep the same X coordinate
# postcondition: has_marker == false && has_marker @ below would be true
def climb:
  # needs to be written
```

Only with this contract information, it is possible to write a semantically correct climb function, especially the precondition is requried: otherwise we would never know where we are going and the robot would need a compass or other means to find its orientation.

Let us complete this procedure at least:

```
# function climb
# precondition: the robot looks up
# invariant: the robot keep the same X coordinate
# postcondition: has_marker == false && has_marker @ below would be true
def climb:
  while has_marker:
    climb
```

Hence, we can start writing a part of our program completely assuming we are in the start location (bottom of left tower):

```
climb
turn
turn
move
pickup
climb # this now climbs down
```

Now we realize that our first intuition that there is a climb function is not a very good and compact one as we also have to climb down. So let us update the precodnition

```
# function climb
# precondition: the robot looks up or down
```

We can then extend the program by going to the side

```
...
turn
move
turn
```

Now, we are one block to the side looking up again. We can then deposit markers twice:

```
climb  # go up
deposit # marker one
climb  # go up
deposit # marker two
turn    #look down
turn
climb   # walk down
turn   # look back
turn
turn
move
turn    # up in origin
```

This completes our imperative formulation with a function climb that we have used multiple times making it much easier to use. And we have seen that even our simple robot can do computations (doubling an integer number represented as the height of a stack of markers).

In fact, similar robots are used to teach programming. For example, Richard Pattis has introduced Karel the robot with a very similar set of features as our imaginary robot and Nikolaus Wirth has popularized this in Germany as Niki the robot

Actually, I found that Niki is still available in a historic Windows version and it runs at least on my Windows 11. So please go ahead and have a look at Niki the robot.

CHAPTER

SIX

THE C AND C++ LANGUAGE

6.1 Introduction

The C programming language is the earliest form of the C and C++ family of programming languages. Nowadays, plain C is used in kernel programming (e.g., the Linux kernel), in long-established code bases (e.g. postgresql), and in very resource-constrained situations (e.g., embedded devices). It also presents a subset of modern C++ and (with very few incompatibilities), every modern C++ compiler will compile plain C source code without any issues.

C and C++ are compiled languages and the compilation process is decomposed into a few tasks and translations that can run independetly from each other. A modern GUI will orchestrate these steps, but it is mandatory to understand the translation procedure to some extent when learning C++.

To keep things simple, a C project typically contains a bunch of C (extension *.c) or C++ (extension *.cpp) files. These contain source code which a *compiler* can translate to object code. Object code is a form of machine code which is not executable as it can be incomplete. In a second phase, all object codes are linked together with system libraries such that all missing symbols are either coming from one of the object files of the project (hence, from your source code) or from some libraries (either system or additional libraries).

In order to deal with the situation that your source code depends on functionality that is not available to the compiler, because it is in a different file or coming from the operating system libraries, the C system provides a second type of file in which only the *signature* of functions is defined. In this way, the compiler is able to generate code to understand the libraries contents (it is kind-of a table-of-contents) well enough to compile the object file without looking into the actual implementation.

Summarizing, we have C++ files (*.c,* .cpp) and header files (*.h,* .hpp) and we translate each soure code file (*.c,* .cpp) into an object file (*.o) in order link together all objects with libraries (*.o, *.lib) into an executable (*.exe on Windows, just a name on Linux).

With this theory, we are set to start learning the core language while already understanding a bit the files that we will create.

6.1.1 Hello World

The following program is the first program you should learn and understand. It just prints out “Hello World” to the screen. We will provide four information aspects in this script: the source code file (C++), a Makefile (which contains the compiler arguments, at the moment just compiling, the output of compiling (could become interesting later) and the output of running the program (note that the first line is the invocation for the case we are later giving arguments). In the HTML version of this book, they are alternative with each other (tabs), in print, they are a bit lengthy, but in this way everything remains complete.

Source

```
#include<stdio.h>

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

Makefile

```
all:
        g++ -Wall -o 01_oldhelloworld 01_oldhelloworld.cpp
run:
        ./01_oldhelloworld
```

Build Output

```

```

Run Output

```

```

The file (like any other C++ files) first lists a few of header files to include (here stdio.h, defined in the POSIX standards, see https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdio.h.html.

From this library, we are relying on the printf function to just output a string.

C is an imperative language and instructs a computer step by step, the entry point (where this processing starts) is a special function called main (WinMain for Windows programs). Similar to what we have seen for the shell examples, the n denotes a special character "new line" and makes sure that the program does not only output Hello World!, but also continues to the next line.

Note: There is a very traditional incompatibility between Windows and Linux with respect to new lines: The Linefeed (LF, n) is sufficient on Linux and Unix, but on Windows, lines are typically ending with CR-LF: first a carriage return (CR, r), then a line feed (n).

The printf function is a powerful utility you can find in all programming languages and it can format various values into the output, what we will cover a bit later, but it has also some downsides, mainly with respect to security and reliability. Hence, printf is not used that much in safety-oriented code.

6.1.2 Hello C++ World

In contrast to C, C++ has introduced the concept of an iostream which better reflects the nature of the standard input and standard ouput. A stream is an object that can represent a file, a network connection, or just a terminal window and one can stream various information into it. Therefore, an operator << is implemented, which takes variables, strings, or whatever information. In this way, type conversion is safe and automatic.

In order to output data to the user, the C++ libraries define a header iostreams which define cout (C output), cerr (standard error output) and cin (for input, used with a >> stream operator).

Hence, the same program written in idiomatic C++ looks like this:

Source

```
#include<iostream>

int main(void)
{
   std::cout << "Hello World!" << std::endl;
   return 0;
}
```

Makefile

```
all:
        g++ -Wall -o 00_helloworld.out 00_helloworld.cpp
run:
        ./00_helloworld.out
```

Build Output

Run Output

6.2 Functions and Signature-based Selection

A little bit later, we will introduce many different types, but for now, we will look into how we can write our own functions similar to the printf functions. A function is a unit of code that can be reused in your own code. Let us consider the following already quite involved example:

Source

```
#include<iostream>

int doubled(int a)
{
   return (a*2);
}

float doubled(float a)
{
   return (a*2);
}

int main(void)
{
   std::cout << "doubled(4)=" << doubled(4) << std::endl
             << "doubled(4.5)=" << doubled(4.5f) << std::endl;
   return 0;
}
```

Makefile

```
all:
        g++ -Wall -o 02_functionsignatures 02_functionsignatures.cpp
run:
        ./02_functionsignatures
```

Build Output

```

```

Run Output

```

```

While this example does not look that much more complicated, we introduced many aspects that are difficult to understand. Let us unroll it a bit. In C++, we are allowed to define multiple functions to have the same name. This is particularly wanted, because sometimes, we can implement an algorithm just different if the arguments are of different types. In our example, we define two times a double function, once it takes an integer argument (int, more on types later, it is just an integral number) or a fractional number (float).

The compiler selects the right function in a complex procedure: For the first call, `doubled(4)`, the type of the constant 4 is integer; hence, the system decides that the first function `int doubled(int)` should be called. For the second call, we have a problem: if we remove the f from the argument, the compiler will tell us the call is ambiguous. What the compiler actually means in this situation is that there is no function that strictly fits to the type (4.5 will be considered a double value in the first place, not a float value, more on this later). Hence, the compiler would generate code and covert the double into an integer if only the first function would exist, it would convert the value into a float if only the second variant would exist. But the compiler has no precedence and raises an error. We resolved this by using a literal: for many

constants, the type can be influenced by putting a character to the constant: 4.5f just means the value 4.5 as a floating point number of the type `float`.

6.3 Code Libraries

In many situations, we will implement functions that we want to use from multiple programs. For example, we could implement a sort function and use it in many different programs if we need to sort. In this situation, we need to put our implementations into an isolated C++ file, create a header file with the so-called *function prototypes* which are just fixing the name of the function and the type of the return value and of all arguments. In this way, we can include the header in our own program (one which has a main), generate code using the functions in the compilation step, and then link together our program with the library into a proper executable.

More concretely, we could move the two implementations of the `doubled` functions into their own C++ file:

Library

```
int doubled(int a)
{
   return (a*2);
}

float doubled(float a)
{
   return (a*2);
}
```

This one can now be compiled independently from everything else. Note that we do not even include iostream here, because we are not using anything from iostreams. In order to make this available, we formualte a header

Header

```
#ifndef DOUBLED_H_INC
#define DOUBLED_H_INC
int doubled(int a);
float doubled(float a);
#endif
```

which contains the signature lines of the functions only (ending with a semicolon). It is valid, but not very widely used to only give the types of the arguments such as

```
float doubled (float);
```

In order to make sure that these definitions cannot be loaded twice (not really needed here, but problems will be coming) all of this header is protected from double inclusion using the preprocessor. All lines with # are processed even before compiling and in order to avoid a double include (maybe you are including the file directly, but some library you include also includes this header) we check if a symbol specific to the header (it is common to use the capitalized filename as depicted) and if not, we define it (such that it is defined in the sequel) and close with `#endif` at end of file. In modern C++, this can be replaced with a simple `#pragma once` in the beginning, but this is not that widely used today.

The remaining project then looks like this:

Source

```
#include<iostream>
#include "03_doubled.h"

int main(void)
{
   std::cout << "doubled(4)=" << doubled(4) << std::endl
             << "doubled(4.5)=" << doubled(4.5f) << std::endl;

   return 0;
}
```

Makefile

```
all:
        g++ -c 03_doubled.cpp -o 03_doubled.o
        g++ -c 03_library.cpp -o 03_library.o
        g++ -o 03_library 03_library.o 03_doubled.o
run:
        ./03_library
```

Build Output

```

```

Run Output

```

```

A small thing to note here is that the include uses ” instead of < and >. This is an old relict and refers to being able to include from the current directory. Libraries to be included from the compiler default location are included with <> and things local to the project with “”.

Especially the Makefile (*.mk) is interesting here as I spelled out all three steps for you: compiling (-c) the library, then compiling the main program part. Afterwards linking together the object files into an executable.

CHAPTER

SEVEN

TURTLE GRAPHICS EXAMPLE

Turtle graphics is a very basic graphics mode that has been used a lot in computer theory and education. It is a very simple model in which computer graphics are generated by a small turtle (also known as the cursor) by drawing a long and single connected line.

This can be used to learn imperative programming and recursive programming in a more visual way.

Some background is given on Wikipedia

7.1 Implementation in C

As C and C++ do not contain graphics functionality in the core of the language (though high performance graphics is not far away through OpenGL and Vulkan and high quality graphics is provided by cairo, we again rely on gnuplot to help us with visualizing scientific data.

The gnuplot script we are providing just requires a text file of coordinates. That is a file in which every line is considered a point consisting of two floating point numbers separated by a space.

The special aspect here is that we are not going to use printf (though we could by using the piping mechanism like

```
./turtle > turtle-graphics.dat
```

but we open a file. Therefore, we rely on `fopen`. fopen returns a pointer (you don't need to know what this is for now), but it returns the special value NULL if something is wrong. If not, there is a printf-like function `fprintf` which we can use to print into this file. The first argument is said file pointer.

The file turtle.cpp implements everything. There are three global variables for the pose of the turtle: `x`, `y`, and `angle`. These are updated by two functions: move() which moves by a distance (if not given a default distance of one is taken). This uses cosine and sine functions and does convert angle to radians due to the library functions `cos`and `sin` being in terms of radians while many people think in degrees.

This conversion is simple: you divide radians by 2π as this brings you a fraction of the circle (like a percentage where 1 means once around the circle, 0.5 means halfway around). And we multiply this with 360. In the implementation, we cancel out a factor of two and divide by π in order to multiply by 180.

Change the program to generate awesome turtle graphics, we will post a few tasks on the web page.

```
#include<stdio.h>
#include<math.h>


double x;
double y;
```

(continues on next page)

(continued from previous page)

```
double angle; /// in degrees

FILE *f;

void move(double len=1)
{
    x = x + len * cos(angle / 180.0 * M_PI);
    y += len * sin(angle / 180.0 * M_PI);
    if (f != NULL)
       fprintf(f, "%f %f\n",x,y);
}

void turn(double by)
{
    angle += by;
    // this is completely unneeded, but might be nice for debugging and is a common␣
 ↪pattern
    // it is only efficient when you are doing moderate turns. For massive turns (e.g.
 ↪, angle=10e9) this
    // will be long loops.
    while(angle > 360.0) angle -= 360;
    while (angle < 0) angle += 360;
}


int main()
{
   f = fopen("plot.dat","w"); // this will overwrite
   if (f == 0) {
      perror("File Problem: ");
      exit(-1);
   }

   // init the turtle
   x = y=angle=0;
   // do some turtle graphics (this is a star, is it?)
   for(int i=0; i < 100; i++){
      move(100);
      turn(140 );
   }
      fclose(f);

   return 0;

}
```

7.2 Visualizing the Turtle Graphics

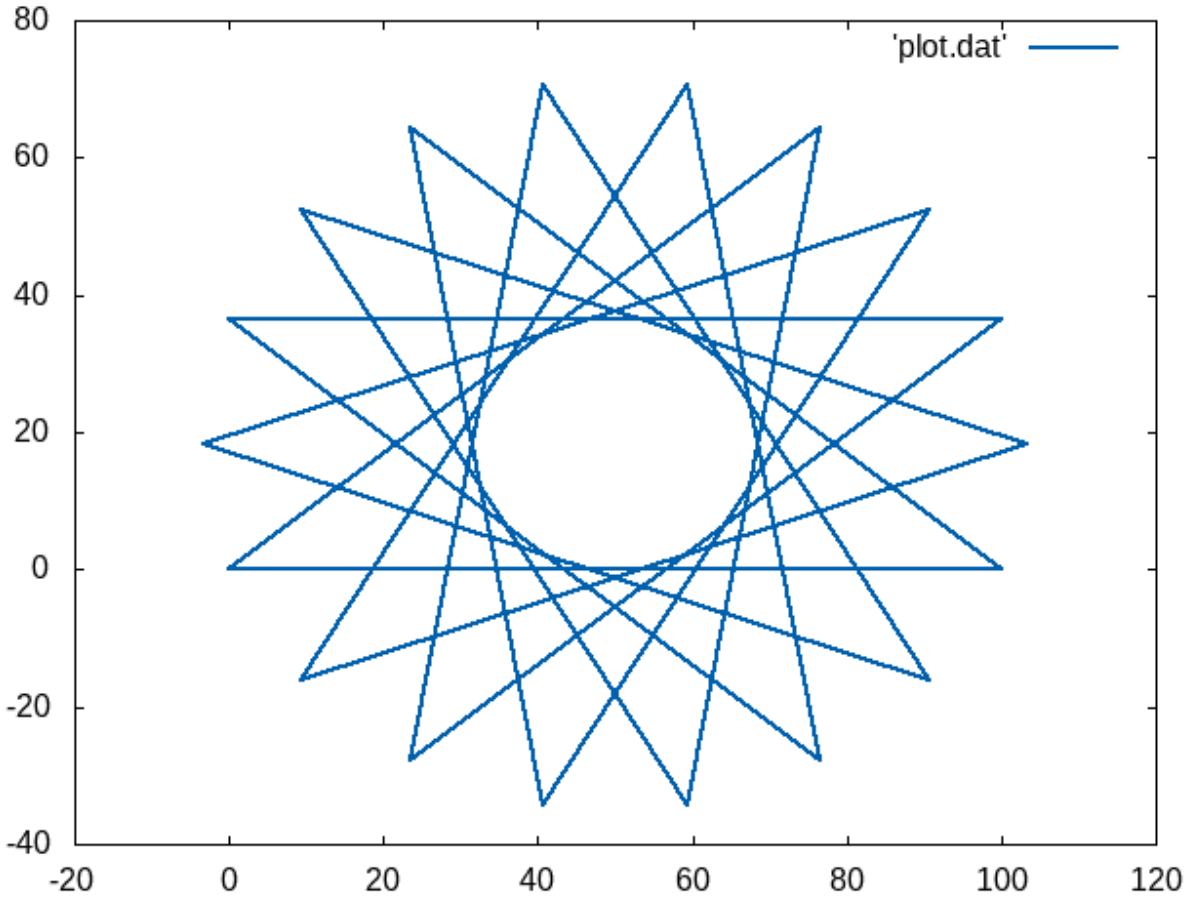
To get you started, we provide a script that generates a PNG file (well, this is handier for the lecture as we typically work remotely on a Linux machine).

```
set term png
set output "/var/www/html/gnuplot.png"

set style line 1 \
    linecolor rgb '#0060ad' \
    linetype 1 linewidth 2 \
    pointtype 7 pointsize 0

plot 'plot.dat' with linespoints linestyle 1
```

By removing the first two lines (activating PNG and giving a filename), gnuplot should pop up with a window.

The script is easy: we just give some visualization parameters and draw all line segments and points from `plot.dat`. As pointsize is zero, you don't see points, but sometimes you might want to activate it, for example, set it to 1.5.

If you did it the right way, you have a nice star from the reference implementation looking like this:



7.3 The Makefile

It is very common that while developing a system, a few things have to be called in the right order. GNU Make is a very common tool that allows you to automate such things.

In its simplest form, it is given as a text file called Makefile which lists at least one target. A target is just a nickname for a set of tasks or a name of a file if a file is to be generated as the outcome of the target.

```
all:
        g++ -o turtle turtle.cpp
        ./turtle
        gnuplot graphics.plt
```

In our minimal example above, the nickname is all. If you invoke make, it will look for the main target (either one given on the command line or the first target in the file) and will try to build it. Therefore, it first builds all dependencies (here are none specified, we will refine this Makefile later). When the dependecies are refreshed, it will run all lines. Each line needs to start with a TAB character (no spaces!) to group them together. And make is really picky, if the first line fails, it will abort the target.

In other words:

```
make
```

will try to make the target all by compiling turtle.cpp into turtle, then running turtle, then running gnuplot. If any of those fails, an error is generated and subsequent aspects are not performed anymore.

You will see more powerful Makefiles through the time…

Part II

Tutorial Assignments

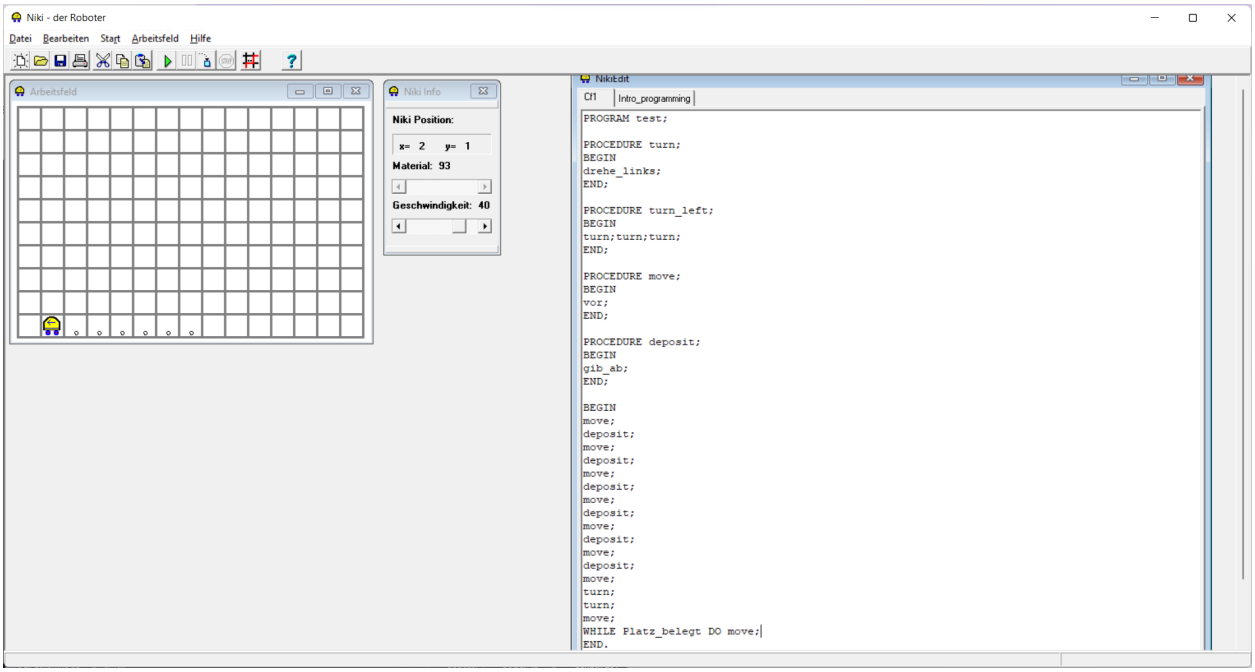
CHAPTER

EIGHT

TASK SHEET 1: NIKI THE ROBOT

Niki is a traditional environment to teach programming. It is about a small robot that has a very limited set of sensors and actions. However, Niki can be programmed in various ways to solve quite complex problems. As Niki does not have a concept of variables, it uses the environment of the robot to store information. In order to efficiently solve some advanced tasks, Niki provides support for recursion, allowing simple solutions to some complex problems.

Below screenshot shows the result of our joint exploration of Niki in the lecture:



8.1 Learning Outcome

We learn how to use loops and conditions to solve some toy problems for Niki. We prepare for the principle of recursion.

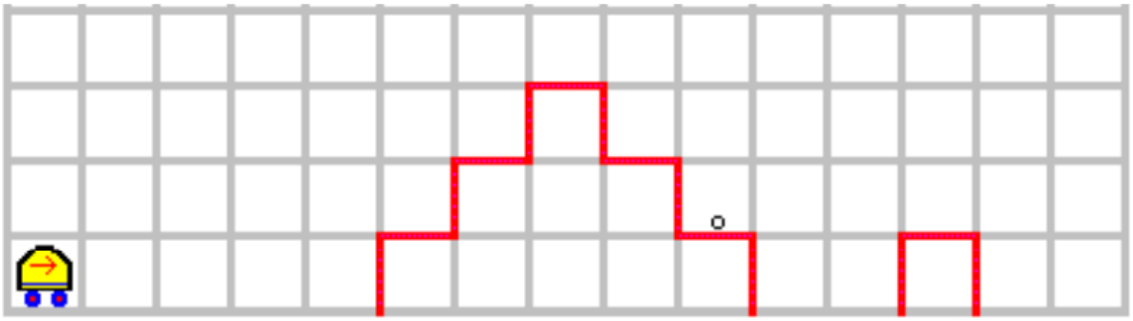
8.2 Task 1: Doubling Numbers

Program Niki to initialize a vertical tower of deposited markers starting at location 1/1. The height of the tower shall be interpreted as an natural number. Assure and assume that the robot is located on the basement of the first tower looking right. Now, write a program (decompose it into procedures where sensible) to double the number by creating a tower right of the given tower with double the height. (Tip: An oral explanation of a possible implementation is given in the lecture video)

8.3 Disclaimer

Below assignments have been taken from the Niki manual. They have been formulated by various people, but not from me.

8.4 Task 2: Staircase



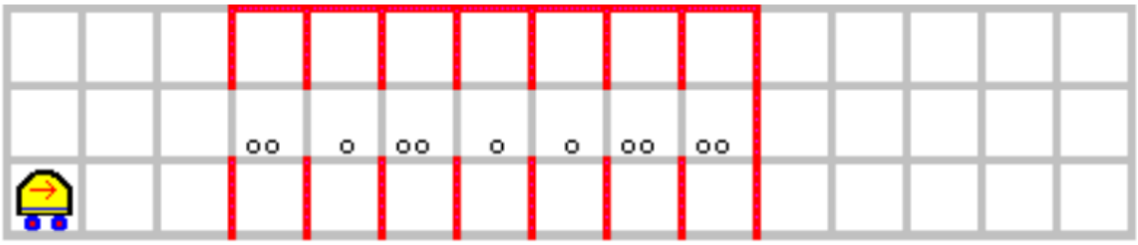
Recreate the given world (click on Arbeitsfeld->Verändern), now the mouse will show you what changes a click would do. These include

- placing markers
- placing walls
- placing the robot

a) Write a program that knows the world and picks up the marker, brings it to the podest and deposits it there.

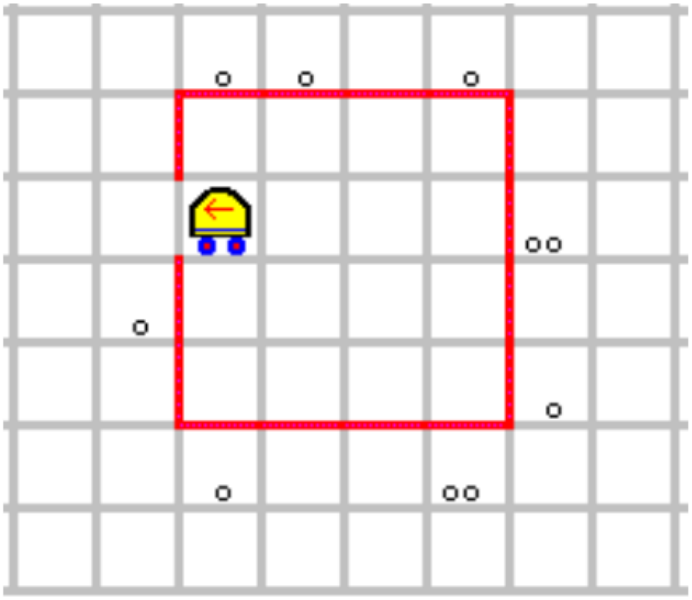
b) Based on domain knowledge that there is first a double staircase and then a podest, find the marker on the podest without knowing the detailed shape or position of the marker and bring it to the podest

8.5 Task 3: Storage



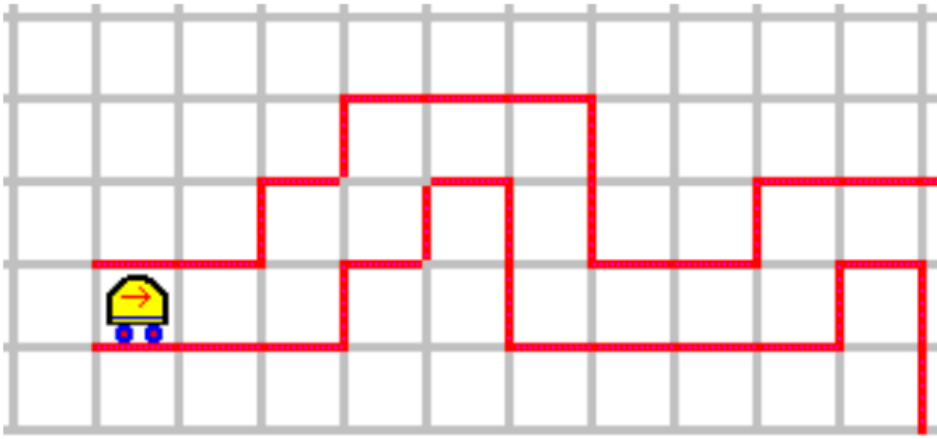
Niki is supposed to sort the large items (two markers) into the bottom line of the depicted storage space and the small ones (one marker) into the upper row. In a first version, he keeps the X location of all of those. In a more advanced version, the result is supposed to be that the both rows are filled from left to right.

8.6 Task 4: Waste Collection



Niki is somewhere within a hall. The hall has one single entrance. Along the outside wall, there is waste deposited. Niki is supposed to pick up all the waste (not knowing where exactly it is, just that it is adjacent to the wall). Write a program for it. (Tip: Try to program it such that left of Niki is always a wall. This is called left hand rule and would walk around any polygons inner or outer boundaries. With the one hole, this rule would change from the inner to the outer boundary and back). This tip enlightens us with a simple fact: sometimes very hard programs can be written pretty concisely by finding an invariant (a natural language condition that remains true throughout the program or program part like a loop). Then, we do a mathematical proof that this invariant is exhausting the problem space (no programming needed) and implement the program (only make sure the invariant is never broken). This approach leads to a notion of algorithmic correctness we will discuss later.

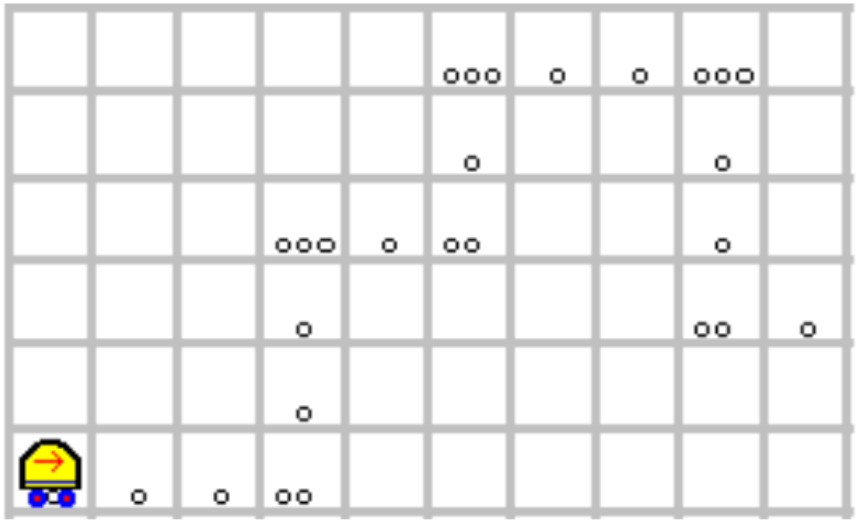
8.7 Task 5: Tunnel



Let Niki pick up the item in the following case

In a few weeks, extend this task by bringing the item back to the beginning of the tunnel of unknown shape. Therefore, either memory (which Niki does not have) or recursion (which Niki has) is requied.

8.8 Task 6: Signs in the world



The graphics visualizes a map in which Niki is supposed to follow the marked cells. The following rules are in place

- a cell filled with one marker means go straight.
- two markers and three markers stand for specific turns

CHAPTER

NINE

TASK SHEET 1: BASIC C AND C++

C and C++ are important imperative programming languages and we are on our way to master these languages. This task sheet is a collection of a wide range of short programs you are supposed to write in order to gain routine in basic structures such as functions, loop, conditions, input and output.

9.1 Task 1: PrettyPrint

In the lecture, we have seen the hello world program. Extend this program to draw a frame around. Examples can be found on https://texteditor.com/ascii-frames/.

Maybe your program should output

```
╔═══════════╗
╟ Hello World ╢
╚═══════════╝
```

9.2 Task 2: PrettyPrint 2

Now extend your program to print a string into a frame. A string in C++ is given as a pointer of type char. `strlen` can be used to find the length of a string. Complete the following program snippet:

```
void pprint(char *what)
{
   printf("╔");
for /*extend this one to print enough of = using strlen(what) */
   printf("╗");
 printf("╟ ");
 printf("%s",what);
 /*...*/
}
```

Make sure that your box wraps the text sensibly (one space character before and after your text)

9.3 Task 3: PrettyPrint3

Extend the previous result in the following way: For all strings shorter than 50 characeters, draw the box as before. For strings longer than 50 characters, compute a new string in which 47 characters are taken from the parameter and the last three characters are manually set to three dots. In this way, arbitrary long things will be printed as abbreviations into your box code. Node that strings shorter than 50 characters must not be abbreviated.

9.4 Task 4: Calculus Primer

As a simple example, consider the logistic curve: $f(x) = \frac{1}{1+e^{-x}}$ This function plays a crucial role in machine learning, so it is worth having a look at it early in your career.

9.4.1 Task 4.1 Plot the Function

As data visualization from C++ is challenging, we will rely on a powerful tool known as gnuplot to show our results. As a preparation, download gnuplot and have a look at http://www.gnuplotting.org/plotting-data/

In this example, gnuplot expects a file to contain your data. Remember that we can redirect the output of programs to files? Your task is to write a C/C++ program that generates a plot of the function as a text file in gnuplots format having always the X coordinate in the first column, the Y coordinate in the second column, and then a newline.

Write a program that generates a plot. In a first version, take constants in your program for start, stepsize, and end as to have a simple loop do all the work going over the X coordinates. Run the program with redirecting the output as follows: `yourprogram > plot.dat` Then, use gnuplot to inspect the data. Did you do it right?

9.5 Task 4.2: The Riemann Integral

As you know, one way of computing the integral of simple functions of one variable is to decompose the parameter space (x) into multiple locations (x_i) and replace the true function with a staircase function by using supremum or infimum of the function. In our case (except at X=0), there are no singularities and the function is monotonous. Hence, we can replace the supremum (resp. infimum) with maximum and minimum of the function values at the left hand side and right hand side of the interval. Given a distance parameter ϵ, write a function to compute the upper sum and lower sum (e.g., the one based on maximum and the one based on minimum) of the integral $\int_1^2 f(x)dx$ Write this function in a single loop looking similar to

```
double epsilon = 0.15;
double US=0; // upper sum
double LS=0; // lower sum
for (double x = 1; double x < 2; x+=epsilon)
{
  //...
  US += //...
  LS += ...
}
printf("We believe that the integral I follows\n");
printf("%.2f < I <= %.2f\n", LS,US);
```

Learn that two different things (computing the upper sum and the lower sum) that depend on the same intermediate values can often be combined.

9.6 Task 4.3: The Discrete Differential

Given a function

```
double f(double x){...}
```

approximate the differential at x=1 by creating a series of discrete difference quotients. Therefore, look at https://en.wikipedia.org/wiki/Difference_quotient and use the central difference formula for varying ΔP

Given a real number q, can you estimate the value of $\frac{\partial f}{\partial x}(q)$

Note that this approach to calculus lies at the heart of discretizations of partial differential equations and is an essential basic concept related to the method of finite elements which you might already know from simulation.

9.7 Task 5: Prime Numbers

Write a function

```
void print_primes(int max)
```

that outputs all prime numbers smaller or equal than a given parameter `max`. Therefore, first implement a predicate `is_prime` which returns a boolean value if or if not the function is prime. This should be tested by a simple loop. Division can be tested by using the modulo function: an integer value q divies a value p if and only if `p % q == 0` In this expression `%` denotes the modulo operation of taking the remainder of dividing `p` by `q`.

Remember, that a predicate is always a function in terms of some data which returns a boolean value.

https://en.wikipedia.org/wiki/Boolean-valued_function

9.8 Task 6: More on prime numbers

The following function is one in which there is still a lot of magic and unknown information. In number theory, the function $\pi(x)$ is the number of prime numbers smaller than x. Compute this number for $x \in [0, 1000]$ and plot it with gnuplot. Look at the staircase pattern, but also at the overall trend. It is growing with a surprising speed, isn't it? This is the heart of the security of current cryptosystems, hence, of all our online payment, communication, finance systems. If there were not enough prime numbers, it would be ridiculously difficult to find some. But as you might know, one needs prime numbers for some encryption algorithms like RSA. Luckily, prime numbers are so common, that we can quickly generate huge ones (even if they are so large, we will not be able to proof that they are prime).

In cryptography, we sometimes use random numbers together with a series of tests to find numbers of which we believe they are prime. They are known industiral-grade prime numbers.

https://en.wikipedia.org/wiki/Industrial-grade_prime

CHAPTER

TEN

TASK SHEET 2 (LECTURES 4-6)

Qualification Goal: In this task sheet, you are expected to develop routine in writing loops and recursions for simple cases. Some of the given formulas are helpful to know.

You can submit your results to Moodle for correction, but preferably as a single C file for all things you want to have corrected. *We will discuss these tasks in the lecture to some extent. But to develop some routine in programming, you need to solve these tasks yourself, preferably without help from the Internet. It is advisable to do these tasks repeatedly (or Google for different sum formulas) until you are able to do them.*

10.1 Task 1: Loops

Implement the following expressions using loops. If two sides are given, implement one as a loop, the other one as an expression. Put the values of the left hand side into a variable lhs, the right hand side into rhs and compare the two values. Output an error message if they don't fit.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Example:

```
#include<stdio.h>

int main(void){
   int lhs=0;
   for (int i=1; i<= n;i++)
     lhs += i;

   rhs=n*(n+1) / 2;
   print("LHS: %d\nRHS: %d\n", lhs, rhs);
   if (rhs != lhs)
     print("Something is wrong");
};
```

- $$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

- $$\prod_{i=1}^{10} n$$

- $$(a+b)^n =$$
- $$\sum_{i=1}^{10} i$$
- $$\sum_{i=1}^{10} (2i+3)$$
- $$n! = \prod_{i=1}^{n} i$$

10.2 Task 2: Binomial Coefficients

Please implement a factorial function first (you could find one in the script, but it is better to try it on your own). It should have the signature

```
int factorial(int k)
```

Then only implement the expression of the binomial coefficient.

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$$

Check it with some known values by hand (create the values and double-check your implementation).

The binomial coefficient is traditionally not defined as such a quotient, but in a recursive manner as follows:

First, we fix base cases

$$\left(\begin{array}{c} n \\ 0 \end{array} \right) = 1 = \left(\begin{array}{c} n \\ n \end{array} \right)$$

and then the recurisve relation

$$\left(\begin{array}{c} n+1 \\ k+1 \end{array} \right) = \left(\begin{array}{c} n \\ k \end{array} \right) + \left(\begin{array}{c} n \\ k+1 \end{array} \right)$$

Implement a recursive function. Note that you will have to rephrase the reccurence relation such that given n and k you can call the binomial function with certain values.

Create a table how long it takes to compute all coefficients up to m, more precisely the set of values:

$$C_m = \left\{ \left(\begin{array}{c} n \\ k \end{array} \right) \text{ for which } 0 \leq k \leq n \leq m \right\}$$

Your program is expected to fail (taking too long) for some cases from C_{65}.

However, there is a solution to this. And this is a good argument why we need to study *efficient* algorithms and why mathematical definitions are sometimes handy for proofs, but bad for computers.

Construct the algorithm given the following non-recursive definition.

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \prod_{i=1}^{k} \frac{n+1-i}{i}$$

Give the number of steps for a concrete instance and estimate the complexity for $n \to \infty$ assuming $k \leq n$.

Speed up this program by transforming the problem for $2k > n$ using the symmetry setting $k := n - k$.

Does this change the complexity? Does it change the runtime? Explain.

CHAPTER

ELEVEN

LEARNING TASKS

11.1 Category 1: General Advice Qualification Goals

These are learning tasks that are not relevant for the practical aspects of the examination. Simple and general questions of the category knowledge (e.g., "what is cd good for") can still be part. So you are expected to know the contents, but not to be able to apply and use them.

- Use the command line

11.2 Category 2: Routines

These are learning tasks that are heavily mandatory for your future work with computers. You are not only expected to solve the tasks, but to build routine abilities to solve them ***without mental load****. The typical way of learning is by repetition of even the same tasks with reducing support (e.g., first time you shall somehow get it done with Google, Stackoverflow, and help from others. But ultimately, you shall just sit down with not more than the plain task description and solve it with very few mistakes.*

By now, you should already have built some routine ability to

- set up C / C++ projects either in Visual Studio or on the command line
- output strings with printf, maybe even integrating local variables or function values you computed.
- implement loops that do what you intended
- understand the principle of recursion (building up at the moment)
- solve yourself the majority of the tasks given to you

11.3 Category 3: In-Depth Knowledge

*These are pieces of knowledge that you need to learn in a classical way. They come up in the examination and are important for understanding future aspects of the lecture and the general landscape of the computational method. Treat this knowledge as a mixture of theoretic knowledge (excerpts, definitions, learn by heart) and application-oriented knowledge (e.g., give complexity, give reasons to certain things, proof simple complexity statements).

You should by now work on learning to

- estimate complexity of simple functions consisting of loops and function calls, but not for recursion.

11.4 Category 4: Computer Science Excellence (out of scope!)

When I was asking the crowd for what the lecture is trying to teach, the first student statement has been the following, which I extended with the condition that makes clear that this is the definition of computer science in itself. And that it is not expected to happen during this lecture. At least not routinely…

If you want to become a famous computer scientist, find an interesting problem and

- take a mathematical formulation and make it efficient as an algorithm