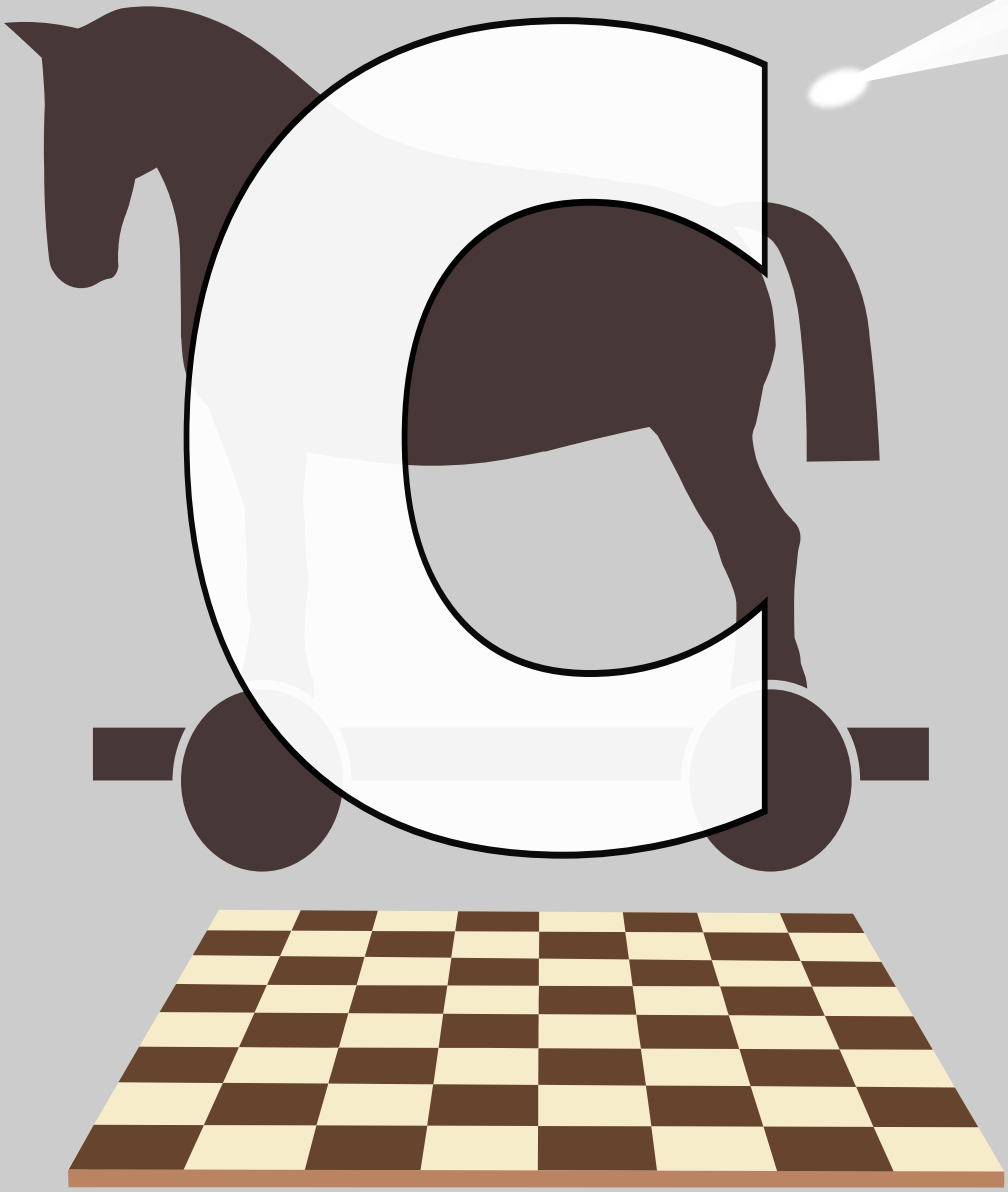
ALIN-ADRIAN ANTON

#include <stdio.h>



# PROGRAMMING TECHNIQUES

LABORATORY ASSIGNMENTS

DR.-ING. ALIN-ADRIAN ANTON

# C PROGRAMMING TECHNIQUES

## LABORATORY ASSIGNMENTS

"STUDENT'S COMMON SENSE" / ACADEMIC USE PRINTING PRESS

PUBLISHED BY "STUDENT'S COMMON SENSE" / ACADEMIC USE PRINTING PRESS

TYPESET WITH TUFTE-LATEX, BOOK COVERS ARE MADE WITH PUBLIC DOMAIN IMAGES



Printed version ISBN 978-973-638-612-1.

*First printing, March 2016*

# *Contents*

# *List of Figures*

# *List of Tables*

# *FOREWORD*

The book titled "Programming Techniques" by dr.ing. Alin-Adrian Anton is a remarkable achievement in its field and has been conceived as a laboratory assignment guide.

Nowadays, programming and its related techniques have become key elements of training not only for specialists, but also for a broad category of computer users. Consequently, there are a lot of manuals and books that relate to this area.

This book, however, presents a modern, exciting and courageous approach for presenting classical concepts of the programming techniques field.

The readers are first introduced to the GNU/Linux Libre universe which is used by the author to support his approach. The book continues with a gradual presentation of the specific concepts within the curricula and is organized into 13 chapters. The laboratory assignments cover subjects such as sorting and searching techniques, composite variables, bit fields, function pointers and variadic functions, text file processing techniques, binary files and dynamic data structures. Algorithm designing techniques like stepwise refinement, divide and conquer, greedy and backtracking are also covered. The final chapter covers the topic of dynamic programming.

Each laboratory assignment is carefully designed using a logical structure which includes the scope of the assignment, the theoretical summary, the assignment breakdown and the methodology, questions and of course home exercises.

Attached to the book the author proposes an interesting and very useful collection of appendices, which on one hand complement the contents, and on the other hand provide practical support and help for the reader. The whole book is carefully written, with a modern and attractive layout and the language belongs to an expert in the field. The author distinguishes himself with proficient English writing skills.

The book is addressed to the students from undergraduate programs in the Information Technology and Computing fields, but it is

equally valuable to a much larger circle of specialists.

Prof. dr. ing. Vladimir-Ioan CREŢU,
Timişoara 19.02.2016

# *How to use this book*

A good practical assignments book is a used book[1]. This book is designed to be used as an instrument for the students to learn fundamental C programming techniques. The book is a companion with practical Laboratory Assignments for the Programming Techniques course lectured by the author at the Polytechnic University of Timişoara, Faculty of Automatic Control and Computing, Department of Computing.

[1] You can find updates at http://academiadesoft.ro

Each Laboratory Assignment is structured in an easy to follow manner:

1. **Scope** – a short statement of what are the objectives of the Laboratory Assignment.
2. **Theoretical Summary** – a usually short theoretical introduction with practical examples.
3. **Assignment Breakdown and Methodology** – a step by step guide for the class time.
4. **Questions** – a list of questions designed to summarize and emphasize what has been learned during the class.
5. **Home Exercises** – a list of exercises for homework and for the students who want to practice more.
6. **Take-Home Idea** – a summary of what has been learned during the class, something to be debated by the personal feedback on the student's own experience.

The book also has a few appendices:

- *Appendix A* – "The Story of Mel", the archetype for the canonical "Real Programmer" as presented by computer science folklore in epic writings.
- *Appendix B* – "The Little Man Computer" (LMC) designed by the Massachusetts Institute of Technology (MIT) to teach students what happens behind the scene of a computer program with a few examples.

- *Appendix C* – a summary of the C language including the C11 standard.
- *Appendix D* – a few useful GNU/Linux manual pages to help the students become familiar with the documentation system and with GNU/Linux.
- *Appendix E* – some health and ergonomic advice on how to use the computer workstation responsibly and what to expect as a professional programmer.

The Bibliography contains easy to read key materials and most of them are available for free.

Any practical assignments book is just for learning the rudiments of something. You know that you understand something when you are capable of explaining it to your colleagues and when you are able to create similar problems and exercises for someone else[2]. This book is just a little bit more than an alphabet. If you want to know more you have to practice, search, research, explore and experiment with what you find. Your greatest advantage is that of living in an information era which has been created by students and people just like you.

It is up to you to keep up the good work and prevent this advantage from being abused.

[2] "Give a man a fish, and you feed him for a day. Teach a man how to use the fishing net, and you feed him for a lifetime."

*Feeding the multitudes, Indian proverb*

# *Laboratory Assignment 1: Introduction to the GNU/Linux tool chain*

### *Scope*

The objective of this laboratory assignment is for the students to familiarize themselves with the GNU/Linux command line environment and with the GNU/Linux tool chain.


[image is based on http://gcc.gnu.org]
The GNU wildebeest is a symbol for Free/Libre Software.

[3] http://parabola.nu

[4] http://fsf.org

### *Theoretical Summary*

The Parabola GNU/Linux-libre distribution[3] is one of the free operating systems recommended by the Free Software Foundation (FSF)[4]. The term "free" is used in the context of:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

The filesystem on GNU/Linux is organized in files and directories:

```
root@parabolaiso / # ls -la
total 60
drwxr-xr-x+  17 1003 1003  4096 Sep 10 18:25 .
drwxr-xr-x+  17 1003 1003  4096 Sep 10 18:25 ..
lrwxrwxrwx    1 root root     7 Oct  7 03:05 bin -> usr/bin
```

```
drwxr-xr-x+   4 root root  4096 Nov 11 05:13 boot
drwxr-xr-x   17 root root  3100 Nov 18 17:37 dev
drwxr-xr-x+  78 1003 1003  4096 Nov 18 17:37 etc
drwxr-xr-x+   3 root root  4096 Nov 11 05:05 home
lrwxrwxrwx    1 root root     7 Oct  7 03:05 lib -> usr/lib
lrwxrwxrwx    1 root root     7 Oct  7 03:05 lib64 -> usr/lib
drwx------    2 root root 16384 Nov 11 05:13 lost+found
drwxr-xr-x+   2 root root  4096 Oct  7 03:05 mnt
drwxr-xr-x+   2 root root  4096 Oct  7 03:05 opt
dr-xr-xr-x  112 root root     0 Nov 18 17:06 proc
drwxr-xr-x+  12 1003 1003  4096 Nov 18 17:38 root
drwxr-xr-x   21 root root   520 Nov 18 17:36 run
lrwxrwxrwx    1 root root     7 Oct  7 03:05 sbin -> usr/bin
drwxr-xr-x+   4 root root  4096 Oct  7 03:05 srv
dr-xr-xr-x   13 root root     0 Nov 18 17:06 sys
drwxrwxrwt    7 root root   200 Nov 18 17:37 tmp
drwxr-xr-x+   8 root root  4096 Nov 11 05:01 usr
drwxr-xr-x+  12 root root  4096 Nov 18 17:07 var
root@parabolaiso / #
```

The symbolic links are shown with arrows, and the first column printed by the "ls" command lists the read, write and execute permissions for the owner, for the group and for everybody else as provided by the GNU/Linux security mechanism.

Special devices are located in the "/dev" directory, including stderr, stdout and stdin. In GNU/Linux, one who can master file input and output can do everything:

```
root@parabolaiso / # ls -l /dev
total 0
crw------- 1 root root     10, 235 Nov 18 17:07 autofs
drwxr-xr-x 2 root root         300 Nov 18 17:06 block
drwxr-xr-x 2 root root          80 Nov 18 17:06 bsg
crw------- 1 root root     10, 234 Nov 18 17:06 btrfs-control
drwxr-xr-x 3 root root          60 Nov 18 17:06 bus
lrwxrwxrwx 1 root root           3 Nov 18 17:07 cdrom -> sr0
drwxr-xr-x 2 root root        2640 Nov 18 17:37 char
crw------- 1 root root      5,   1 Nov 18 17:07 console
lrwxrwxrwx 1 root root          11 Nov 18 17:06 core -> /proc/kcore
drwxr-xr-x 2 root root          60 Nov 18 17:06 cpu
crw------- 1 root root     10,  62 Nov 18 17:07 cpu_dma_latency
crw------- 1 root root     10, 203 Nov 18 17:06 cuse
drwxr-xr-x 6 root root         120 Nov 18 17:06 disk
brw-rw---- 1 root disk    254,   0 Nov 18 17:07 dm-0
lrwxrwxrwx 1 root root          13 Nov 18 17:06 fd -> /proc/self/fd
```

```
crw-rw-rw- 1 root root      1,   7 Nov 18 17:07 full
crw-rw-rw- 1 root root     10, 229 Nov 18 17:07 fuse
crw------- 1 root root    249,   0 Nov 18 17:07 hidraw0
crw-rw---- 1 root audio    10, 228 Nov 18 17:07 hpet
drwxr-xr-x 2 root root           0 Nov 18 17:06 hugepages
lrwxrwxrwx 1 root root          25 Nov 18 17:06 initctl -> /run/systemd/initctl/fifo
drwxr-xr-x 4 root root         300 Nov 18 17:07 input
crw-r--r-- 1 root root      1,  11 Nov 18 17:07 kmsg
lrwxrwxrwx 1 root root          28 Nov 18 17:06 log -> /run/systemd/journal/dev-log
brw-rw---- 1 root disk      7,   0 Nov 18 17:07 loop0
brw-rw---- 1 root disk      7,   1 Nov 18 17:07 loop1
brw-rw---- 1 root disk      7,   2 Nov 18 17:07 loop2
brw-rw---- 1 root disk      7,   3 Nov 18 17:07 loop3
brw-rw---- 1 root disk      7,   4 Nov 18 17:07 loop4
brw-rw---- 1 root disk      7,   5 Nov 18 17:07 loop5
brw-rw---- 1 root disk      7,   6 Nov 18 17:07 loop6
brw-rw---- 1 root disk      7,   7 Nov 18 17:07 loop7
crw-rw---- 1 root disk     10, 237 Nov 18 17:07 loop-control
drwxr-xr-x 2 root root          80 Nov 18 17:06 mapper
crw------- 1 root root     10, 227 Nov 18 17:07 mcelog
crw-r----- 1 root kmem      1,   1 Nov 18 17:07 mem
crw------- 1 root root     10,  59 Nov 18 17:07 memory_bandwidth
drwxrwxrwt 2 root root          40 Nov 18 17:06 mqueue
crw------- 1 root root    254,   0 Nov 18 17:07 ndctl0
drwxr-xr-x 2 root root          60 Nov 18 17:06 net
crw------- 1 root root     10,  61 Nov 18 17:07 network_latency
crw------- 1 root root     10,  60 Nov 18 17:07 network_throughput
crw-rw-rw- 1 root root      1,   3 Nov 18 17:07 null
crw-r----- 1 root kmem      1,   4 Nov 18 17:07 port
crw------- 1 root root    108,   0 Nov 18 17:06 ppp
crw------- 1 root root     10,   1 Nov 18 17:07 psaux
crw-rw-rw- 1 root tty       5,   2 Nov 18 17:43 ptmx
drwxr-xr-x 2 root root           0 Nov 18 17:06 pts
crw-rw-rw- 1 root root      1,   8 Nov 18 17:07 random
.
.
.
root@parabolaiso / #
```

The "mount" command prints the mounted filesystem with "/dev/sro" the serial ATAPI CDROM with format ISO9660.

```
root@parabolaiso / # mount
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
sys on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
```

```
dev on /dev type devtmpfs (rw,nosuid,relatime,size=501152k,nr_inodes=125288,mode=755)
run on /run type tmpfs (rw,nosuid,nodev,relatime,mode=755)
/dev/sr0 on /run/parabolaiso/bootmnt type iso9660 (ro,relatime)
cowspace on /run/parabolaiso/cowspace type tmpfs (rw,relatime,size=766332k,mode=755)
/dev/loop0 on /run/parabolaiso/sfs/root-image type squashfs (ro,relatime)
/dev/mapper/parabola_root-image on / type ext4 (rw,relatime)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup \
    (rw,nosuid,nodev,noexec,relatime,xattr, \
    release_agent=/usr/lib/systemd/systemd-cgroups-agent,name=systemd)
pstore on /sys/fs/pstore type pstore (rw,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
.
.
.
root@parabolaiso / #
```

For editing files we have many options. A decent solution is to install Atom[5] the text editor from github.

[5] http://atom.io

In order to check the documentation for a C function or a program for working on the command line, we can use the manual pages:

```
root@parabolaiso / # man mkdir

MKDIR(1)                          User Commands                          MKDIR(1)

NAME
mkdir - make directories

SYNOPSIS
mkdir [OPTION]... DIRECTORY...

DESCRIPTION
Create the DIRECTORIES, if they do not already exist.

Mandatory  arguments  to  long  options are mandatory for short options
too...
.
.
.
```

Now in order to create a directory, we must make sure we understand the difference between relative paths and absolute paths. The

following "mkdir" commands are usually equivalent:

```
mkdir ~/dir1
rmdir ~/dir1
mkdir $HOME/dir1
rmdir $HOME/dir1
mkdir /home/user/dir1
rmdir /home/user/dir1
cd $HOME; mkdir dir1
ls -l ~/
rm dir1
rmdir dir1
mkdir -p /home/user/dir1
```

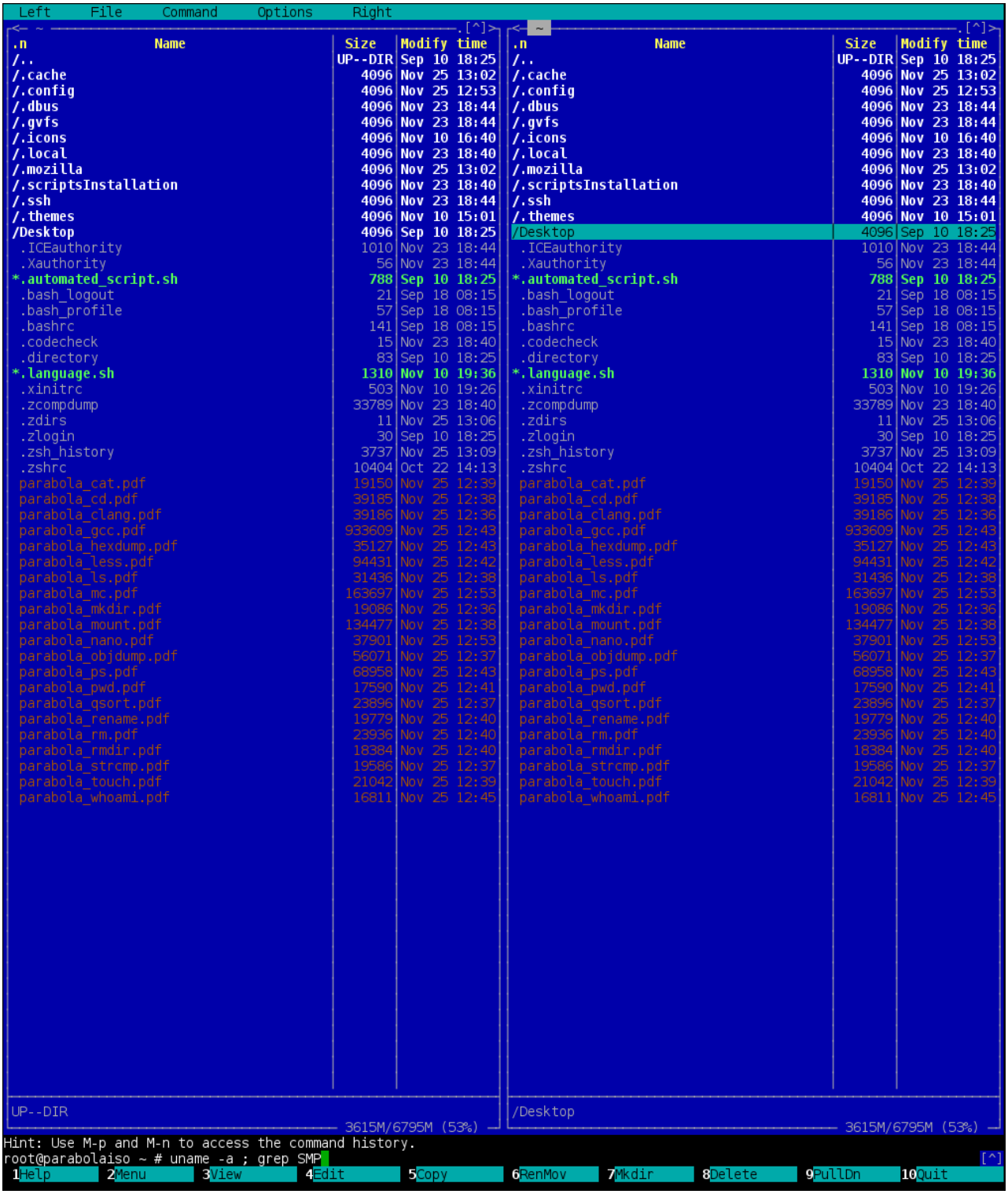
The user may be the superuser also known as root or any other user in the system.

The man page for the "strcmp()" C library function interfaced in "string.h" is obtained by:

```
root@parabolaiso / # man strcmp
```

or to be sure

```
root@parabolaiso / # man 3 strcmp

STRCMP(3)                  Linux Programmer's Manual                 STRCMP(3)

NAME
strcmp, strncmp - compare two strings

SYNOPSIS
#include <string.h>

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

DESCRIPTION
The  strcmp()  function compares the two strings s1 and s2.  It returns
an integer less than, equal to, or greater than zero if  s1  is  found,
respectively, to be less than, to match, or be greater than s2...
.
.
.
```




because the C library functions are usually located on Section 3 of the GNU/Linux manual.

Obviously it is much easier to use the Midnight Commander[6]

[6] http://midnight-commander.org/

program for file manipulation.

The superuser usually has a "#" shell prompt and the regular user just a "$" ending shell prompt.

Other useful commands can be found in Appendix C.

With a small trick of output redirection we can use "cat" to create a small C program:

```
root@parabolaiso / # cat > test0.c
#include <stdio.h>

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

Ctrl-D (EOF)
root@parabolaiso / # cat test0.c
root@parabolaiso / # less test0.c
```

so now we can compile it using either "clang" or "gcc":

```
root@parabolaiso / # clang -O2 -Wall -o test0.x test0.c
root@parabolaiso / # ./test0.x
Hello student!
root@parabolaiso / #
```

### *Assignment Breakdown and Methodology*

Create the following hierarchy of directories in your home directory:

```
~/test/test1/test11
~/test/test1/test12
~/test/test2/test21
~/test/test2/test22
```

1. with the "cd" command
2. without using the "cd" command
3. delete the directory test21, having as current directory the one called test, by using a relative path and an absolute path
4. list the contents of the directory test and its subdirectories
5. delete the directory test with all its subdirectories
6. do everything again from "mc"

7. Write a C function to check the validity of ISBN-13 numbers. The last digit of an ISBN-13 number is obtained from the weighted sum of all the other digits. For instance, for ISBN 9781435704572 the sum is $1 \times 9 + 3 \times 7 + 1 \times 8 + 3 \times 1 + 1 \times 4 + 3 \times 3 + 1 \times 5 + 3 \times 7 + 1 \times 0 + 3 \times 4 + 1 \times 5 + 3 \times 7 = 118$. The last digit is obtained from $2 = 10 - (118 \mod 10)$ which means the input ISBN number is correct.

8. A 10 digit ISBN uses the sum $x_{10} = (\sum_{i=1}^{9} ix_i) \mod 11$ where $x_{10}$ is the check digit. If the last digit is 10, it is replaced with a roman "X". For example ISBN 0521396549 has the check digit 9. Write a C function to check the validity of ISBN-10.

9. ISBN-13 and ISBN-10 use different error detection schemes for the final, checksum digit. Write a C function to convert from ISBN-13 to ISBN-10 and another function to convert from ISBN-10 to ISBN-13. The first 3 digits of ISBN-13 are always 978.

10. Write a C program that tells the user to think about a number between 1 and 100 and then tries to guess the number by asking questions of the form "Is your number greater than ...?" and "Is your number smaller than ...?". Save the program in a file called "guess.c". Compile and link the file with GCC(1). Run the program.

11. Study the man page for the CLANG compiler and recompile the program using CLANG. Always use optimization flags and set the warning level to the maximum. Compare the results and the size of the two executables, the one produced with GCC and the one produced with CLANG. Use HEXDUMP(1) to read the binary executable files. Use OBJDUMP(1) to disassemble the two program files.

12. Carefully study the man page for the quicksort function, QSORT(3). Write a program to sort an array of numbers given as input and print it back on the screen.

13. The GNU Debugger is called GDB(1). Recompile your sorting program using DEBUG flags and trace it step by step to study how it works.

## *Questions*

- What is "free" or Libre software?
- Is "free" software free?
- What does the "ls" command do?

- What is an absolute path?
- What is a relative path?

### *Home Exercises*

Install GNU/Parabola Linux, PC-BSD[7] and OpenSUSE[8]. These can be installed natively, on the entire hard disk or in separate virtual machines. Make sure the GNU and the LLVM/CLANG tool chains are installed.

Rework the Laboratory Assignment for those three operating systems installed. Study how they work and notice the subtle differences.

When finished, experiment with ReactOS[9] by installing the tool chain and compiling the assignments.

[7] http://pcbsd.org
[8] http://opensuse.org
[9] http://reactos.org

### *Take-Home Idea*

Free software is a social movement. The four free software freedoms have nothing to do with software price. A software is free if it guarantees:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

The Cathedral[10] and the Bazaar by Eric S. Raymond is a good read on the topic.

[10] "Give a man a fish..."

# *Laboratory Assignment 2: Independent compilation*

## *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with the preprocessor, independent compilation and command line arguments. Students will learn how to create and use libraries and pass parameters from the command line to their programs.

Thus spake the Master Programmer: "Though a program be but three lines long, someday it will have to be maintained." , *The Tao Of Programming*, Book 5.

## *Theoretical Summary*

The C compilation model is shown in Fig.1.

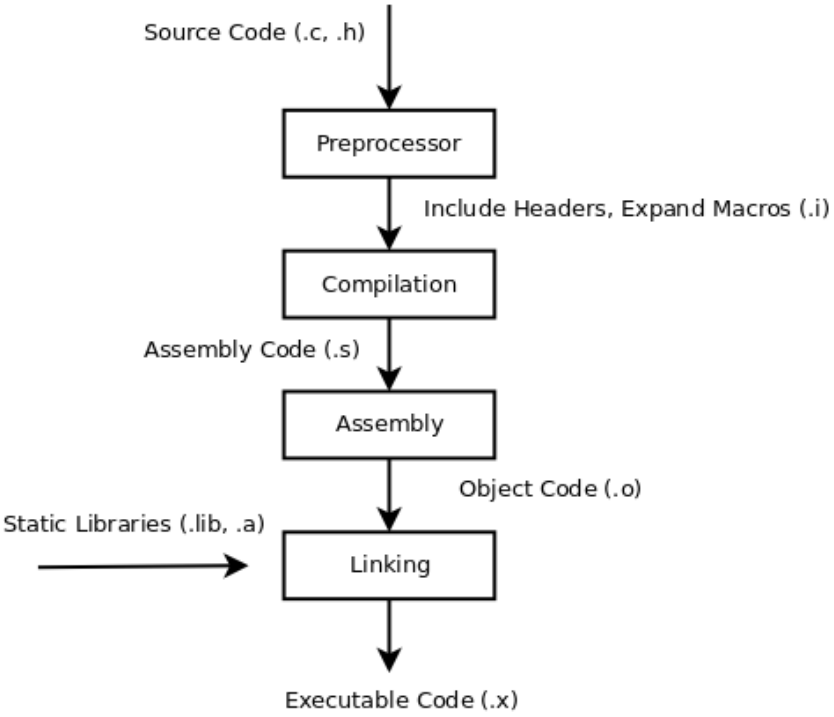



Figure 1: Compilation process in C

Thus spake the Master Programmer: "The Tao gave birth to machine language. Machine language gave birth to the assembler.

The assembler gave birth to the compiler. Now there are ten thousand languages.

Each language has its purpose, however humble. Each language expresses the Yin and Yang of software. Each language has its place within the Tao.

But do not program in COBOL if you can avoid it. ", *The Tao Of Programming*, Book 1 The Silent Void.

### *Command Line Arguments*

Command line arguments in C are passed by specifying

```
int
main (int argc, char *argv[]) {}
```

to the main() function. This means that the main() requires a number of **int** argc parameters in the char * array **char** *argv[]. **char** *argv[] and **char** **argv are the same. The first parameter from argv[] is a pointer to the first string from the command line which contains the name of the binary program.

In order to test if the number of command line arguments are correct, we have to use the argc variable which is the argument counter:

```
#include <stdio.h>

int
main (int argc, char **argv)
{
  printf ("Hellow world\n");
  return printf("%d argument(s), argv[0] = %s\n", argc, argv[0]);
}
```

The **return** instruction returns the value stored by the last call to the printf() library function. According to the manual, upon successful return, the printf() functions return the number of characters printed excluding the zero character.

It is mandatory to return something if main() is defined to return something different than void. If there is a chain of programs or commands running in the command line it is important to return meaningful values to the calling process.

### *The Preprocessor*

The preprocessor first removes comments from the source code and replaces "#" preprocessing indications with something useful before compilation.

For example

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

is replaced with the actual contents of the stdio.h file which is usually located somewhere inside "/usr/include" directory. If the "<" and ">" triangular brackets are replaced with normal quotation marks the file is searched for in the current directory.

```
#include "myheader.h"
```

The preprocessor replaces all macro definitions throughout the entire source code before compilation:

```
#define MAX_SIZE 2048
#define MIN(a,b) (((a)<(b))?(a):(b))
#define min(a,b) ( (a)<(b) ?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))
```

However the replacements can cause compilation problems if taken without care. For example min() used as

```
k = min (x + y, abs(z));
```

will expand as

```
k = ((x + y) < (abs(z)) ? (x + y) : (abs(z)));
```

where x + y has been substituted for a and abs(z) for b.

There are also predefined macros like __FILE__ and __LINE__ which are expanded before compilation to the file name and the line number position in the sources.

The **#ifdef** and **#ifndef** macros can be used to enable the specific operating system and debug helper code:

```
#ifdef DEBUG
#define DEBUG_TEST 1
#else
#define DEBUG_TEST 0
#endif

void main (void) {
  if (DEBUG_TEST) fprintf(stderr, "[%s][%d] doing something\n", __FILE__, __LINE__);
  /* ... code ... */
}
```

### *The Compiler*

The compiler translates the source code prepared by the preprocessor and converts it into a list of assembly language instructions. It is always important to ask the compiler to work harder, optimize the code and print all possible warnings. This will help detect subsequent problems.

```
gcc -Wall -O3 -o file.x file.c
clang -Wall -O3 -o file.x file.c
```

```
bu@firefly:~> file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked \
  (uses shared libs), for GNU/Linux 2.6.32, \
  BuildID[sha1]=0x681930b397195658349eac85d3c3cf8b75589b95, not stripped
```

### *The Assembler*

The assembler converts the assembly instructions into binary, object code.

### *The Linker*

The link editor combines the main() function with object code functions referred by the source code from your own program or from outside libraries in order to produce a working executable. External variable references are also resolved here.

### *The Makefile*

The Make or GNU Make utilities are tools which control the generation of executables and other non-source files of a program from the program's source files.

Make gets its knowledge of how to build your program from a file called the makefile, which lists each of the non-source files, and how to compute it from other files. When you write a program, you should write a makefile for it, in order to make it possible to use Make for the building of the program and for the installation process.[11]

[11] GNU Make documentation

The basic makefile is called "makefile" or "Makefile" and is composed of:

```
target: dependencies
[tab] system command
```

A simple makefile example is

```
all:
  [tab]  gcc -Wall -O3 -ansi -pedantic main.c hello.o string.o -o hello.x
```

which also links the object code from "hello.o" and "string.o" into the main routine. When the "make" program is called from the command line, the "all" target is executed by default:

```
bu@firefly:~> make
gcc -Wall -O3 -ansi -pedantic hello.c -o hello.x
bu@firefly:~> ls -l hello.x
-rwxr-xr-x 1 bu users 12421 Dec  5 17:39 hello.x
bu@firefly:~>
```

A more complex example which can be customized is given by

```
# the compiler: gcc for C program, define as g++ for C++
CC = gcc

# compiler flags:
#  -g    adds debugging information to the executable file
#  -Wall turns on most, but not all, compiler warnings
CFLAGS  = -g -Wall -O3 -ansi -pedantic

# the build target executable:
TARGET = hello

all: $(TARGET)

$(TARGET):
  $(CC) $(CFLAGS) -c string.c
  $(CC) $(CFLAGS) -c main.c
  $(CC) $(CFLAGS) -o $(TARGET).x $(TARGET).c string.o main.o

clean:
  $(RM) $(TARGET).x
```

### *Creating a Library*

Multiple object files can be archived into a library. Let us create a library called "libhello":

```
/* libhello.c */

#include <stdio.h>

void hello(void) {
  printf("Hello, library world.\n");
}
```

The header file for "libhello" is just an interface telling us how to call hello():

```
/* libhello.h */

void hello(void);
```

Libraries can be statically linked into the main executable, can be called as shared collections of object code or can be dynamically loaded during the runtime using dlopen().

```
/* main.c */

#include "libhello.h"

void main(void) {
  hello();
}
```

We can compile "main.c" independent of the other files using the "-c" flag from the compiler. However, in order to produce a working executable called "main.x" one needs to link with the object code from "libhello.c".

Let us create a static library called "libhello.a" using the "ar" archiver and run the program:

```
gcc -Wall -O3 -pedantic -c -o libhello.o libhello.c
gcc -Wall -O3 -pedantic -c main.c -o main.o
ar rcs libhello.a libhello.o
gcc -Wall -O3 -pedantic -o main.x main.o -L. -lhello
./main.x
```

The ".a" file can contain one or more object files produced by the compiler. By default the standard C library functions are included when linking is done in a shared manner. If we include <string.h> the object code for the string manipulation functions like strcmp() is linked by references directly into the library without including it physically into the "main.x" executable. Static linking clones the physical library objects into the final executable so the file is larger but the program is faster.

Dynamic linking is only used when new object code is required during the runtime.

The C language provides storage classes for all functions and variables: **extern**, **static**, **auto** and **register**. A header file is an interface which might contain private portions for internal use: if a variable or function in the ".h" file is declared as being "static" it will not be accessible from outside the header file. If it is declared "extern" it means it is defined somewhere else.

## *Assignment Breakdown and Methodology*

1. Run the examples from Theoretical Summary and study how they work. Ask questions and understand them well.
2. Create a string manipulation library with your own static implementation of the strcmp(), strcpy() and strncpy() functions using two steps:

   (a) wrap the calls directly to the original functions from <string.h>

   (b) replace the wrapper calls with your own solutions and remove any references to <string.h>

3. use a Makefile to compile a proof of concept program and show that your static library is working correctly.

Hint: remember to use the [Tab] key in the Makefile. For bonus points also try to call the library using dlopen() as shown in the man pages.

### *Questions*

1. What does the Preprocessor do?
2. What does the Compiler do?
3. What does the Assembler do?
4. What does the Linker do?
5. What does the Makefile do?
6. What does the make utility do?
7. Which is faster: a static library, a shared library or a dynamically loaded library at runtime?
8. Which is slower: a static library, a shared library or a dynamically loaded library at runtime?
9. Which produces the biggest executable and why: a static library, a shared library or a dynamically loaded library at runtime?
10. How many objects can be stored in a library archive with objects?
11. When do we use dynamically loaded libraries?
12. What are the storage classes used for in C?

### *Home Exercises*

1. Check the <string.h> header file and implement your own version for every function interfaced by it. Your own object code should be compatible with the original and behave as specified by the man pages of string.h. Try to improve the speed of strlen() and strstr(). Add strtoupper() and strtolower() for uppercase and lowercase conversion which is now handled by <ctype.h> with toupper() and tolower().

2. Package your <string.h> code in a library "libcstring.a" and use it to link a program which uses strncat(), strncpy() and strncmp() functions. Stay away of bound-ignorant string functions like strcat(), strcmp() and strcpy() for safety reasons. You should have at least the following files prepared:

   ```
   Makefile
   cstring.h
   cstring.c
   main.c
   demo.c
   README
   ```

   where "demo.c" is a short example of how to use your library and "README" is a text file having the same structure as a manual page[12].

[12] If your homework is ready and you really want it, you can try to find out how to create a man page by reading "man man", "man man-pages" and "man mdoc".

A simple man page named "demo.1" in "roff" format may look like:

```
.TH DEMO 1
.SH NAME
demo.x \- concatenate strings by removing vowels
.SH SYNOPSIS
.B demo.x
[\fB\-n\fR \fIVOWELS\fR]
[\fB\-\-vowels\fR \fIVOWELS\fR]
.IR file ...
.SH DESCRIPTION
.B demo.x
concatenates lines read from file by removing the vowels
.SH OPTIONS
.TP
.BR \-n ", " \-\-vowels =\fIVOWELS\fR
Set the number of vowels to remove.
Default behavior is that all the vowels are removed.
```

and you can read it with "man ./demo.1". The README file should contain the output format, not the source.

## *Take-Home Idea*

Source code files can be compiled independently using a Makefile. This allows for software teams to work together by sharing contract header ".h" files and individually compiled object code or libraries.

# *Laboratory Assignment 3: Sorting and searching*

## *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with fundamental searching and sorting algorithms. Sorting, searching and string processing are fundamental operations performed with a computer.[13]

[13] Remember: sorting, searching and string processing are frequent operations performed with a computer.

## *Theoretical Summary*

Sorting and searching are essential operations performed with a computer[14]. We are going to study Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort. The algorithm finds subsets of data are already ordered, and uses that knowledge to sort the rest of it more efficiently. We are also going to study binary search, which is implemented in the standard C library by bsearch().

[14] Firewalls use string processing in order to inspect and sort network packets.

### *Sorting*

Sorting is a fundamental operation performed with a computer. There are many ways to sort a sequence of elements, say integers. For example, in order to sort the sequence of numbers:

```
3 7 6 4 2 1 8 5
```

we can start picking numbers from the left and comparing them two by two to detect if the relation of precedence is violated:

```
3 7 [6] 4 2 1 8 5
```

3 has no precedent number; 7 is greater than 3 so there is no infringement; 6 **is lower** than 7 so we select it because it violates the order of precedence. We have to reinsert [6] before [7] where it belongs:

```
3 |6| 7 [4] 2 1 8 5
```

Thus spake the Master Programmer: The Grand Master Turing once dreamed that he was a machine. When he awoke, he exclaimed:
"I don't know whether I am Turing dreaming that I am a machine, or a machine dreaming that I am Turing!", *The Tao Of Programming*, Book 2.

The next number selected is [4] because it **is greater** than [7] and we reinsert it between [3] and [6] where it belongs:

```
3 |4| 6 7 [2] 1 8 5
```

[2] **is greater** than [7] so we remove it from the sequence and reinsert it before [3].

```
|2| 3 4 6 7 [1] 8 5
```

Again [1] is breaching the order so it is removed and reinserted before [2].

```
|1| 2 3 4 6 7 8 [5]
```

8 is greater than 7 so there is no infringement, but [5] has to be removed and reinserted between [4] and [6] where it belongs. The final sequence is sorted:

```
1 2 3 4 |5| 6 7 8
```

Another approach is to recursively split the sequence of numbers into halves and pairs, and collect them back together according to their order of precedence.

For example

```
4 2 1 7
```

can be split into

```
[4 2] and [1 7]
```

and furthermore into

```
[4] [2]  and  [1] [7]
```

When collecting them back, the order of precedence is imposed:

```
[2 4] and [1 7]
```

and then again

```
[1 2 4 7]
```

So we have two very simple techniques for sorting an array of elements. The same can be done for strings by using strcmp() from <string.h>.

Timsort combines these two techniques by adapting and deciding when to sort by insertion sort and when to sort by the merging back of the numbers.

We must first detect subsequences of numbers which are ordered either by ascending or descending order. If we need to sort them by ascending order the subsequences which we find to be in descending order can be easily swapped to reverse their order from the margins to the middle of the subsequence. Such subsequences are called *runs* and are based on the fact that it is natural to expect that at least a number of elements to be sorted are already conforming to the sorting rule.

A natural run is a sub-array that is already ordered. Natural runs in real-world data may be of varied lengths. Timsort chooses a sorting technique depending on the length of the run. If the natural run is smaller than *minrun* it means Timsort will use the insertion method for sorting.

For an array of fewer than 64 elements, *minrun* is the size of the array, reducing Timsort to an insertion sort. For larger arrays, *minrun* is chosen from the range 32 to 64 inclusive, such that the size of the array, divided by *minrun*, is equal to, or slightly smaller than, a power of two.

Now that we understand how sorting is done, we will have to test our abilities using both numerical values and strings, as well as one of the two methods discussed.

### *Searching*

Binary searching works with already sorted data or with ordered data. It converges to a solution in logarithmic time, because it reduces the data set to be worked upon by a factor of 2 during each iteration. In other words, the data that is being processed is always reduced by half after each iteration.

A simple example is a game one can play against an opponent or against a computer: a challenge for number guessing. The human player choses a number between 1 and 100. The computer starts asking questions about the size of the number comparing it to different helper values. The human player answers with yes or no:

```
Did you pick a number between 1 and 100? [y/N]Y
Is the number lower than 50? [y/N]Y
Is the number greater than 25? [y/N]Y
Is the number lower than 37? [y/N]N
Is the number greater than 43 [y/N]N
Is the number lower than 40? [y/N]N
Is the number greater than 41 [y/N]N
Is the number greater than 40 [y/N]Y
Your number is 41!
```

Searching for a substring in a string or for a character in a string can be straightforward using one or more loops, but can also be improved by using specially crafted algorithms.

## *Assignment Breakdown and Methodology*

1. The rand() function declared in the <stdlib.h> interface and explained in the GNU/Linux man pages can be used to generate a random sequence of numbers and store them into a vector array. Write a program to populate such an array of given size N and sort the numbers using any of the two methods that are part of Timsort.

2. Write a program to show the top students from a class given their names and grades. The program should read the names of the students and their grades from the keyboard. The program will display a menu and allow the user to sort the students by alphabetic order or by grades. For example the menu can look like this:

```
0. Exit program
1. Give N, number of students
2. Enter the students
3. Display class in alphabetic order
4. Display class creating a top based on grades
5. Display the first 3 students according to their grades
```

   but feel free to improve.

3. Write a program for number guessing and carefully test the example given for binary search.

4. Write a program to store the first N prime numbers into a vector array. N is given as input from the keyboard. Write a function to test, using binary search, if a given number X is present or not into the array of prime numbers. You should test if X is prime before searching the array.

## *Questions*

1. Why do you think Timsort adapts and uses either insertion sort either merge sort?

2. Elaborate about the performance, and which one you think is faster and why: merge sort or insertion sort?

3. Why do you think small *runs* Timsort is being reduced to insertion sort?

4. Can binary search be used on an array of unsorted numbers?

5. Can binary search be used to locate a character into a string?

6. Does binary search and merge sort have anything in common, that looks familiar to both techniques?

7. Do you think the fact that, when binary search is used, after each iteration the remainder of the data to be searched is reduced by half has an impact on the performance of the algorithm?

8. Do you think that taking into account the distribution of the numbers in the sequence to be searched is important if the distribution is non-uniform? ("e" for example is the most frequent letter in English language).

9. Can you sort floating point numbers using the same techniques?

10. Is it more appropriate to sort data using pointers and indexes instead of the real thing? Explain.

### *Home Exercises*

1. Write a program to test the performance of the two sorting techniques, insertion sort and merge sort. Populate a vector array with random numbers using the rand() function from <stdlib.h>. Write two functions for sorting a local clone of the array called insertion_sort() and merge_sort(). In C function parameters are passed by locally cloning their values. Sort the array previously populated with random numbers using the two techniques and compare the swap counters for the two methods.

2. Write a program to test the performance of the binary search technique. Populate a vector array of $2^N$ elements with random numbers using the rand() function from <stdlib.h>. N is given as input, with N=10 being a good starting point. $2^{10} = 1024$. Search for the presence of a given number X in the sequence. Before sorting you can use linear search, reading the array element by element from 0 to N-1 or from N-1 down to 0. Sort the array and search again using binary search. Count the number of steps during each search attempt. Compare the final counters and display the results: was the given X number found? Repeat the steps for different values of N.

## *Take-Home Idea*

Sorting, searching, string processing and arithmetic are the most frequent operations performed by a computer. Binary search uses sorted data to look for an element in a row in logarithmic time. The remaining data is always halved using base 2. Timsort is not related to Timişoara, but the TimS computer is[15].

[15] The TimS Plus computer was the last desktop PC designed by the Computing Department of the Polytechnic University of Timişoara.

# *Laboratory Assignment 4: Composite variables. Bit fields*

## *Scope*

The objective of this laboratory assignment is for the students to familiarize themselves with the C features of type definition, enumeration, structure, union and the bit field.

## *Theoretical Summary*

The C language provides the keywords **struct** and **union** for grouping variables together inside a C block. The difference between the two is how memory is reserved inside the block. For **struct** nothing happens to the space required by the variables in terms of memory. For **union**, however, all contained variables are overlapped into a single memory space which has to be large enough to store any of the variables but not all of them at once.

For example the size of the following structure

```
struct snode {
        char name[10];
        char address[20];
        float income;
}
```

is given by **sizeof** to be the sum of the variables inside the structure, which in this verse[16] is 10+20+4 = 34 bytes = 272 bits[17].

On the contrary if we replace **struct** with **union** we get:

```
union unode {
        char name[10];
        char address[20];
        float income;
}
```

with **sizeof**(**union** unode) being 20, since this is the size of the largest variable in the group.

Remember the space inside a **union** is overlapped so if we set a variable inside the group all the others are overwritten. For example:

[16] Just a quote from the Firefly series.

[17] How many 32-bit registers are necessary to hold 272 bits? What about 64-bit?

```
union unode u1, *u2;
u1.income=3.14;
```

will surely destroy what was previously stored in name and address. Displaying those will display garbage even if we did initialize them, which of course we did not.

Did you know that $\pi$ seconds equal a nanocentury?

What about u2? Well, u2->income=3.14 is not going to work with the u2 variable not properly initialized or if it has been set to NULL. The u2 variable requires some dynamically allocated memory so:

```
u2 = (union unode *) malloc(sizeof(struct union unode));
if (u2 == NULL) printf("Failed");
else {
        u2->income = 3.14:
        u2->name = 0x0;
        u2->address = 0;
}

free(u2);
```

Remember we use -> instead of . for dynamically allocated structures and unions so that we easily remember if we need to call malloc() and free.

Now, instead of repeating **struct** snode or **union** unode we can define our own data types with **typedef**:

```
typedef struct _snode {
        char name[10];
        char address[20];
        float income;
} snode;

typedef union unode theUnion;

snode snode1, *snode2;

theUnion u1, *u2;

typedef unsigned char BYTE;

u1.income = 3.14;
```

In this case _snode the private name of our structure may be omitted.

The **enum** type is sort of a structure used for enumeration. The internal variables are given numbers behind the scenes which we don't care about if what we need is a simple enumeration:

```
enum cardsuit {
        CLUBS,
        DIAMONDS,
        HEARTS,
        SPADES
} deck[52];
```

or one can simply specify the internal values:

```
enum cardsuit {
        CLUBS    = 1,
        DIAMONDS = 2,
        HEARTS   = 3,
        SPADES   = 4
} custom_deck[52];
```

Bit fields can be used to work with data types lower than 1 byte or simply to use a compact stream of 1 and 0 to encode our information. Bit fields are usually unsigned int, but when necessary they can also be defined using a signed int type.

For example:

```
struct {
        unsigned int age : 3;
        unsigned int dayWeek: 3;
        unsigned int flagValid: 1;
} Info;
```

The 3 bits for the day of the week can encode up to $2^3 = 8$ values, from 000 to 111. If the type is unsigned, the leftmost bit also known as the most significant is used to signal if the number has a minus sign or not.

Bit fields can be assigned values just as any other variables. Correct practice requires that the programmer is careful to provide values only within the bit field range. For example:

```
Info.flagValid = 999;
```

is trusted by the compiler to be correct but program behavior may be undefined.

When the bit field is initialized it is up to the programmer to decide how to encode the information within. For instance, Info.dayWeek=010 can be assigned to Monday but there is no reason why Monday should not be associated with Info.dayWeek=111 instead. The C compiler trusts the user.

The standard C input functions from <stdio.h> do not know how to read bit fields directly. For this reason an auxiliary variable must be used:

```
unsigned int aux;
scanf ("%u", &aux);

if (aux == 0) Info.flagValid = 0;
else Info.flagValid = 1;

if (aux > 1) fprintf(stderr, "Sir, You should be more careful with your input!\n");
```

### *Assignment Breakdown and Methodology*

1. Write programs to include the examples in the theoretical summary and run them.

2. Write a program that memorizes a list of geometric objects given as input by the user in 2D Cartesian coordinates. The objects can be circles, triangles and rectangles. Pack the information as much as possible using **struct**, **union** and bit fields and use your own data types. Print the list of geometric objects on the screen. Use the **enum** keyword to differentiate between the object types and group pairs of (x,y) coordinates together.

### *Questions*

1. What is the difference between **struct** and **union**?

2. What data types can be used to define a bit field?

3. Is it useful to define a bit field over a floating point type?

4. Can bit fields be defined over single precision and double precision floating point types?

5. What is the relation between a **struct** and an **enum**?

6. Is it possible to use custom values for an **enum**?

7. Is it possible to use floating point values for an **enum**?

8. How many bits are in a bit field?

9. What is the maximum number of bits in a bit field?

10. What is the difference between signed and unsigned bit fields?

## *Home Exercises*

1. A simple classification of stars is as follows:

   (a) Main Sequence Stars characterized by: age category (an integer value between 1 and 4) and color (a character array of at most 12 characters)

   (b) Binary Stars rotating around a mass center characterized by: rotation radius for the first and the second star

   (c) Variable Light Stars characterized by: lowest luminosity, highest luminosity and light periodicity measured in Earth days

   All stars have a name of arbitrary length. Use composite variables and bit fields and write a program that reads data about a star and then prints it back on the screen.

2. Animals can be classified into insects, birds, mammals and fishes. Write a program making the best use of composite variables and bit fields to manage a database of animal entries. The system must be able to add, delete, replace and lookup entries into the memory database. The animal classes are described by:

   (a) number of legs and life span for **insects**

   (b) flight speed, wing length and migration habbits for **birds**

   (c) weight, height and food type for **mammals**

   (d) weight, swimming depth and water salt percentage for **fishes**

## *Take-Home Idea*

Composite variables allow for grouping variables into a block of code. Bit fields can be defined as internal integer types. The difference between **struct** and **union** variable groups is that **union** uses a single memory block for all the variables inside. Use **union** when the internal variables are mutually exclusive (ie. do not use **union** for (x,y,z, and t) coordinates).

# *Laboratory Assignment 5: The Stepwise Refinement Method*

## *Scope*

The scope of this Laboratory Assignment is for the students to develop systematic iterative programming skills by reducing the problem down to simpler objectives and only adding complexity in a step by step manner and only after the simplified model is functional. Stepwise refinement is a powerful technique for developing a complex program from a simple program by adding features incrementally.

## *Theoretical Summary*

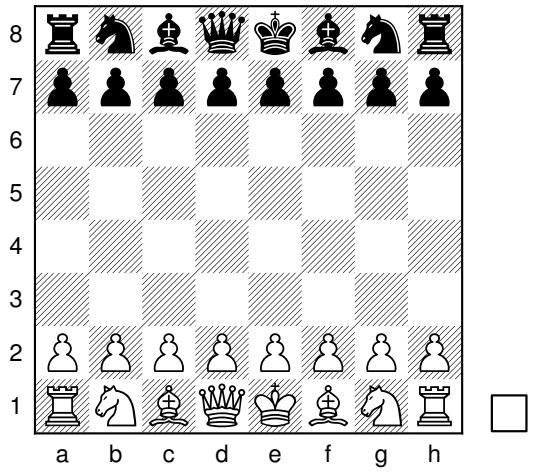
The Stepwise Refinement Method means that we begin with a very simple model for the initial request. Then, we implement something that provides raw feedback on the screen and only after passing each step like this the model is refined by adding layers of complexity to it.

Thus spake the Master Programmer: " In the beginning was the Tao. The Tao gave birth to Space and Time.

Therefore Space and Time are the Yin and Yang of programming.

Programmers that do not comprehend the Tao are always running out of time and space for their programs. Programmers that comprehend the Tao always have enough time and space to accomplish their goals.

How could it be otherwise? " , *The Tao Of Programming*, Book 1 "The Silent Void".




Let us consider a game of chess. A board configuration can be graded using the evaluation function:

```
f(board[8x8]) = 200(Kw-Kb)
  + 9(Qw-Qb)
  + 5(Rw-Rb)
  + 3(Bw-Bb + Nw-Nb)
  + 1(Pw-Pb)
  - 0.5(Dw-Db + Sw-Sb + Iw-Ib)
  + 0.1(Mw-Mb)
```

where KQRBNP = number of kings, queens, rooks, bishops, knights and pawns, and D,S,I = doubled, blocked and isolated pawns, and M = Mobility (the total number of legal moves). White and black pieces are taken into account.

The board has 8 x 8 = 64 positions and each player has 16 pieces of which 8 are pawns. Some pieces can be placed upon any of the 64 squares. The function evaluates the board from white perspective: the larger the grade, the more advantage for the white player. Before making any move, the white player evaluates its best options according to what the table would look like after every possible legal move.

Stepwise refinement is a powerful paradigm for developing a complex program from a simple program by adding features incrementally[18].

[18] IEEE Transactions on Software Engineering

We will start out by creating a program skeleton with a main() function and with some auxiliary functions to be used from main(). We'll fill in the code later. Think of the chess playing engine as this:

```
#include <stdio.h>

int main (void) {
        init_board();
        print_board();
        play();
}
```

Basic stuff, but we need to add the functions:

```
#include <stdio.h>

void init_board();
void print_board();
void play();

int main (void) {
        init_board();
        print_board();
        play();
```

```
}
```

If you want to see some action *add some feedback*[19] on the screen:

[19] The best way to create Lab Assignments is to create easy to follow tutorials, just like with reverse engineering. Learning should be fun and rewarding. Otherwise there is no Tao. There exist people who would like to destroy the Tao in you.

```
#include <stdio.h>

void init_board() {
        printf("The␣board␣has␣been␣configured.\n");
}

void print_board() {
        printf("The␣board␣is␣printed.\n");
}

void play() {
        printf("Checkmate!␣I'm␣still␣playing..\n");
}

int main (void) {
        init_board();
        print_board();
        play();
}
```

Now according to f() we need to know the KQRBNP values for white and black pieces. So we should provide some functions for computing them:

```
#include <stdio.h>

#define LEVEL   100 // how many arbitrary moves to evaluate?

int _Kw; // number of king pieces
int _Kb;

int _Qw; // number of queen pieces
int _Qb;

int _Rw; // number of rocks
int _Rb;

int _Nw; // number of knights
int _Nb;

int _Pw; // number of pawns
int _Pb;
```

```
int _Dw(); // doubled pieces
int _Db();

int _Sw(); // static (blocked) pieces
int _Sb();

int _Iw(); // isolated pieces
int _Ib();

int _Mw(); /* mobility or freedom coefficient,
            * the total number of possible moves
            */
int _Mb();

void init_board() {
        printf("The board has been configured.\n");
}

void print_board() {
        printf("The board is printed.\n");
}

void play() {
        printf("Checkmate! I'm still playing..\n");
}

int main (void) {
        init_board();
        print_board();
        play();
}
```

Should consider how to encode the chessboard, now that everything is set? The easiest thing to think of is to define an 8 by 8 matrix and use numbers to encode the type of each piece. For example, negative numbers should belong to black pieces and positive numbers to the white pieces. Take the following encoding table as an example inspired by f():

| Piece Type | White | Black |
|---|---|---|
| King | 200 | -200 |
| Queen | 9 | -9 |
| Rock | 5 | -5 |
| Bishop | 3 | -3 |
| Knight | 2 | -2 |
| Pawn | 1 | -1 |

so the initial setup can be encoded as:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -5 | -2 | -3 | -9 | -200 | -3 | -2 | -5 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 2 | 3 | 9 | 200 | 3 | 2 | 5 |

For humans, moves are encoded using piece type and their new position on the board. The position on the board is given by numbers from 1 to 8 starting with the leftmost lower corner and with letters from a to h. So the initial position for the kings are Ke1 Ke8. The leftmost lower corner is always a black square and bishops are bound to move within the colors of their flags. In a C matrix the kings are located at [0,4] and [7,4] with [0,0] being the leftmost upper white corner.

Now the only remaining thing is to encode the movements for each of the 6 piece types. One idea is to encode the move as an index displacement relative to the current [i,j] position on the board. Moves are only valid if they fall within free squares on the table or upon opposite color pieces. This is easy for the King because it has only 8 possible displacements. But for the Queen you have to write a snippet to generate the table of moves.

White pawns go up and black pawns go down so we have to decide how to say that in C. Let us add some **#define**s for the piece type values and we'll decide how to move and where later.

```
#include <stdio.h>

#define KING    200
#define QUEEN   9
#define ROCK    5
#define BISHOP  3
#define KNIGHT  2
#define PAWN    1

#define LEVEL   100 // how many arbitrary moves to look ahead?

int _Kw; // number of king pieces
```

```
int _Kb;

int _Qw; // number of queen pieces
int _Qb;

int _Rw; // number of rocks
int _Rb;

int _Nw; // number of knights
int _Nb;

int _Pw; // number of pawns
int _Pb;

int _Dw(); // doubled pieces
int _Db();

int _Sw(); // static (blocked) pieces
int _Sb();

int _Iw(); // isolated pieces
int _Ib();

int _Mw(); /* mobility or freedom coefficient,
            * the total number of possible moves for white
            */
int _Mb();

void init_board() {
        printf("The board has been configured.\n");
}

void print_board() {
        printf("The board is printed.\n");
}

void play() {
        printf("Checkmate! I'm still playing..\n");
}

int main (void) {
        init_board();
        print_board();
        play();
```

}

We added **#define** LEVEL 100 to configure how many moves to evaluate before actually moving. An idea is to actually scan the matrix for pieces with our flag color and pick random valid moves to decide which is best. We have to do the same for the opponent pieces because the evaluation function requires both flags to decide. We are going to use rand() from <stdlib.h> to pick something from the chessboard and a **switch**() directive to decide how to move it based on the piece type.

```
#include <stdio.h>

#define KING    200
#define QUEEN   9
#define ROCK    5
#define BISHOP  3
#define KNIGHT  2
#define PAWN    1

#define LEVEL   100 // how many arbitrary moves to look ahead?

int _Kw; // number of king pieces
int _Kb;

int _Qw; // number of queen pieces
int _Qb;

int _Rw; // number of rocks
int _Rb;

int _Nw; // number of knights
int _Nb;

int _Pw; // number of pawns
int _Pb;

int _Dw(); // doubled pieces
int _Db();

int _Sw(); // static (blocked) pieces
int _Sb();

int _Iw(); // isolated pieces
int _Ib();
```

```
int _Mw(); /* mobility or freedom coefficient,
            * the total number of possible moves for white
            */
int _Mb();

int f_eval ( ) { // the f ( ) eval function

}

int random_move (int color) {
        if (color > 0) { // we are white
                // random pick a white pice with random i, j from table
        } else if ( color < 0) { // all blacks here
                // random pick a black piece with random i, j from table
        }

        return f_eval (); // evaluate the board and add to score
}

void init_board() {
        printf("The board has been configured.\n");
}

void print_board() {
        printf("The board is printed.\n");
}

void play() {
        printf("Checkmate! I'm still playing..\n");
}

int main (void) {
        init_board();
        print_board();
        play();
}
```

As we have understood so far, a better way to encode the piece positions after randomly picking them and after evaluating arbitrary moves on the table is to use arrays for each of the 16 pieces of a player. A piece set to 0 is a missing piece. So let us add the arrays:

```
int white_pieces[16];
int black_pieces[16];
```

```
int white_positions[16][2];
int black_positions[16][2];
```

## *Assignment Breakdown and Methodology*

Stepwise refinement is a programming principle and method by which a programmer is writing the source code in an iterative manner adding layers of complexity step by step. The first and simplest layer should provide some feedback on the screen and dummy functions for all the functionality imagined.

1. A classic tic tac toe game in a 3-by-3 square table is a 2 player game with "X" and "0". This is an example where "X" wins:

   | | | |
   |---|---|---|
   | X | 0 | 0 |
   | X | X | - |
   | X | - | 0 |

   The first player "X" has 9 possible places where to start from. After the first move, 8 positions remain for "0". The total number of game sequences is $9! = 362880$. The winning player is the first to have a row, column or diagonal drawn with his own pieces.

2. Say the computer plays with "0" and always has the second move. Write a dummy program to play tic tac toe using SWR. Declare some dummy functions and print something on the screen.

3. Add one layer of complexity using SWR. Use a function to display the board configuration on the screen with "X", "0" and empty squares.

4. Add another layer of complexity using SWR by doing something more useful with the functions: enhance the program to play dumb tic tac toe but with valid moves.

5. Add some signs of intelligence: instruct the program to block any winning position of the "X" player.

## *Questions*

1. Why is the Stepwise Refinement Method called like that?

2. Is it important to provide feedback on the screen starting with the first and simplest layer of complexity?

3. What if we forget some needed functions and we realize that only while programming other layers of complexity?

4. Is it necessary to add more feedback on the screen during each and every layer of complexity ?
5. Do you think it is useful to print on the screen the current layer of complexity?
6. The SWR prevents from creating one big main.c file full of global variables and one huge main()) function. Why ?
7. Is SWR different from intuitive common sense programming where programs are developed step by step? How?
8. Is SWR a good method for small programs?
9. Is SWR useful for large software?
10. Can SWR be used with independent compilation?

### *Home Exercises*

1. Carefully test the chess examples from the Theoretical Summary and use SWR to add layers of complexity until you can play against a dumb computer which produces arbitrary but valid moves. Add another layer of complexity in the spirit of SWR by actually using the board evaluation function f(): choose the best configuration from a pool of board configurations which can be obtained from a given initial setup. If the pool is sufficiently large, your program will show signs of intelligence. You will have to study the basic chess rules.
2. Tic tac toe can be played in 3D boards. Write a 3D board game of tic tac toe by using SWR and independent compilation in order to create a 3D tic tac toe playing engine. Enumerate the complexity layers. At the final layer of complexity the engine should block the opponent's winning positions but it should also try to win. The easiest way to display the 3D tic tac toe cube is to print 3 separate tables for each cube level.

### *Take-Home Idea*

The Stepwise Refinement Method is a strategy for writing source code with dummy functions. The central main() function contains descriptive calls to as many dummy functions as necessary. Complexity and functionality is added step by step by developing the dummy functions into functional components.

# *Laboratory Assignment 6: Function pointers. Variadic functions*

### *Scope*

The scope of this Laboratory Assignment is to learn how to use function pointers correctly and how to write functions that receive a variable number of arguments.

### *Theoretical Summary*

Pointers can be used to address functions just as well as they address variables. Both variables and functions are concepts originating in mathematics. Being expanded into a set of instructions and data that were blended together they form a program for the deterministic machine called "computer".

The "computer" is just a computing device which sometimes calls for instructions or data from a remote system or library. A function is defined as

```
return_type function_name(argument list);
```

and a pointer to a function is defined as

```
return_type (*function_pointer_name)(argument list);
```

The C programming language provides the pointer mechanism for both variables and functions. Function pointers can be used in order to incorporate intelligence into composite variables by associating proper functions to the ordinary data inside the structures or unions.

For example, let us write a program with geometric figures which are intelligent enough to decide what functions to call in order to manage their internal data.

```
#include <stdio.h>

#define N 5
```

```
struct gshape {
        int type; /* 0/1/2 */
        int ID;
        union
        {     struct
              {
                  float radius;
                  float x,y;
              } circle;
              struct
              {
                  float x1,y1, x2,y2, x3,y3;
              } triangle;
              struct {
                  float x1,y1; /* upper left */
                  float x4,y4; /* right bottom */
         } rectangle;
    } gshape;
    /* function pointers in struct provide contained intelligence */
    void (*read_gshape)(struct gshape *gs);
    /* like an object having both data and code */
    void (*write_gshape)(struct gshape *gs);
};

void read_circle(struct gshape *gs) {
    printf("radius? ");
    scanf("%f", &(gs->gshape.circle.radius));
    printf("X center? ");
    scanf("%f", &(gs->gshape.circle.x));
    printf("Y center? ");
    scanf("%f", &(gs->gshape.circle.x));
}

void write_circle (struct gshape *gs) {
    printf("Printing circle ID=%d:\n", gs->ID);
}

void read_triangle (struct gshape *gs) {
    printf("Reading triangle:\n");
}

void write_triangle (struct gshape *gs) {
    printf("Triangle (%f,%f) (%f,%f), (%f,%f)\n", gs->gshape.triangle.x1,
```

```
        gs->gshape.triangle.y1, gs->gshape.triangle.x2,
        gs->gshape.triangle.y2, gs->gshape.triangle.x3,
        gs->gshape.triangle.y3);
}

void read_rectangle (struct gshape *gs) {
    printf("Reading␣rectangle:\n");
}

void write_rectangle (struct gshape *gs) {
    printf("Printing␣rectangle:\n");
}

int main (void) {
    int i, type;
    struct gshape gs[N];

    printf("Initializing␣the␣geometric␣shapes␣collection:\n");
    for (i = 0; i < N; i++) {
        do {
            printf("Shape␣ID␣%d:␣", i);
            printf("What␣type␣of␣shape␣is␣it?␣(0=circle,␣1=triangle,␣2=rectangle)␣");
            scanf("%d", &type);
            gs[i].type = type;
            gs[i].ID = i;
            switch (gs[i].type) {
                    case 0:
                        gs[i].read_gshape = &read_circle;
                        gs[i].write_gshape = &write_circle;
                        break;
                    case 1:
                        gs[i].read_gshape = &read_triangle;
                        gs[i].write_gshape = &write_triangle;
                        break;
                    case 2:
                        gs[i].read_gshape = &read_rectangle;
                        gs[i].write_gshape = &write_rectangle;
                        break;
                    default:
                        printf("Yikes!\n");
            }
        } while ((gs[i].type != 0) && (gs[i].type != 1) && (gs[i].type != 2));
    }
```

```
    printf("Printing␣all␣the␣circles␣from␣our␣collection:\n");
    for (i=0; i<N; i++) if (gs[i].type == 0) gs[i].write_gshape(&gs[i]);

    return 1;
}
```

It is sometimes desirable to implement functions that require an arbitrary or unknown number of parameters. For example a maxval(n1, n2, n3, ...) function for determining the maximum value from a list of parameters is a good candidate. The following program uses the arithmetic_mean(**unsigned int** n, ...) function in order to compute the arithmetic mean of the n values given as parameters.

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

/* float is automatically promoted to double via ...
*  because of the stack memory requirements
*/
double arithmetic_mean(unsigned int n, ...) {

   va_list valist;
   float sum = 0.0;
   unsigned int i;

   /* initialize valist for n arguments */
   va_start(valist, n);

   /* parse n arguments of type float from valist */
   for (i = 0; i < n; i++) {
      sum += va_arg(valist, double);
   }

   /* clean valist */
   va_end(valist);

   return sum/n;
}

int main() {
   printf("Average␣of␣1.2,␣3.14,␣4,␣5␣=␣%f\n",
      arithmetic_mean(4, 1.2, 3.14, 4.0, 5.0));
   printf("Average␣of␣1,␣2,␣3␣=␣%f\n",
      arithmetic_mean(3, 1.0, 2.0, 3.0));
```

```
    return 1;
}
```

Macro definitions can be defined as variadic just like functions are: **#define** errprintf (...) fprintf (stderr, __VA_ARGS__). Function pointers can also point somewhere in the data region:

```
#include <stdio.h>

/* gcc -fno-stack-protector -z execstack -o exit exit.c ;
printf "\x48\x31\xFF\x48\x31\xC0\xB0\x3C\x0F\x05" > a.out ;
objdump -D -b binary -m i386:x86-64 a.out
0000000000000000 <.data>:
   0:   48 31 ff                xor    %rdi,%rdi
   3:   48 31 c0                xor    %rax,%rax
   6:   b0 3c                   mov    $0x3c,%al
   8:   0f 05                   syscall
*/

char code64[]="\x48\x31\xFF\x48\x31\xC0\xB0\x3C\x0F\x05";

int main (void) {
        int (*func)(); /* function pointer for a function */
                       /* with no arguments but with a return value */

        func = (int (*)()) code64; /* cast from char array to function pointer */

        (int)(*func)(); /* calls exit(0), Intel 64 bit CPU */

        printf("Should exit(0) gracefully before getting here!\n");

        return;
}
```

A function pointer is just a regular pointer. If it points the data region the computer will try to read executable instructions from its variables and run them. For this to work the program has to be compiled on 64 bit Intel CPU using the flags -fno-stack-protector for disabling the default stack protection and -z execstack in order to enable the stack to become executable. The program should exit graciously instead of crashing. The actual instruction from code64[] can be disassembled by:

```
printf "\x48\x31\xFF\x48\x31\xC0\xB0\x3C\x0F\x05" > a.out
objdump -D -b binary -m i386:x86-64 a.out
0000000000000000 <.data>:
```

```
0: 48 31 ff             xor    %rdi,%rdi
3: 48 31 c0             xor    %rax,%rax
6: b0 3c                mov    $0x3c,%al
8: 0f 05                syscall
```

where 0x3c or decimal 60 is the code for the exit() GNU/Linux 64 bit system call and RDI and RAX are cleared to zero (AL is part of RAX).

## *Assignment Breakdown and Methodology*

1. Carefully study and experiment with the examples from the Theoretical Summary section.
2. Using the examples from the Theoretical Summary section write a program which makes use of intelligent structures in order to index a collection of geometric objects (circles, triangles and rectangles) capable of computing their own area. Print the area for every rectangle from the database.
3. Using the examples from the Theoretical Summary section write a program which takes an arbitrary number of arguments and determines the minimum and the maximum value from the list.
4. Write a program which generates an array with function pointers pointing to the following functions from <math.h>: sin(), cos(), ceil(), floor(), fabs(), log() and sqrt(). Generate a secondary array with the names of these functions and allow the user to select which function to call using a menu. Read a number and apply the selected function to it, then print the result on the screen.

## *Questions*

1. What is a function pointer?
2. Can a regular pointer point to a function?
3. Can a function pointer point to a data variable?
4. Is it possible with proper casting to exchange addresses between a function pointer and a data pointer and viceversa?
5. Can a function pointer be NULL?
6. Can a variadic function be converted in a regular function?
7. Can a regular function be converted in a variadic function?
8. Is it possible to define variadic macros?

9. What is the difference between a variadic macro definition and a variadic function?

10. Is it better to have one qsort() function accepting function pointers to an argument comparison function, or would it be better to have \qsort_string(), \qsort_float(), \qsort_int() functions for the multiple data types provided by the C language? Please explain.

## *Home Exercises*

1. Write a program with a custom_printf() variadic function emulating the real printf() function prototyped in <stdio.h>. Use a format string parameter for determining the number of arguments from va_list. You will have to manually parse the format string.

2. The following program uses qsort() from the standard C library in order to sort a list of arguments given as parameters from the command line:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int cmpstringp(const void *p1, const void *p2) {
/* The actual arguments to this function are "pointers to
 pointers to char", but strcmp(3) arguments are "pointers
 to char", hence the following cast plus dereference */
return strcmp(* (char * const *) p1, * (char * const *) p2);
}

int main(int argc, char *argv[]) {
   int j;
   if (argc < 2) {
      fprintf(stderr, "Usage: %s <string>...\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   qsort(&argv[1], argc - 1, sizeof(char *), cmpstringp);
   for (j = 1; j < argc; j++) puts(argv[j]);

   return 1;
}
```

Extend the program so that it also works with single and double precision floating point numbers given in exponential format.

### *Take-Home Idea*

Pointers can point to functions just like they point to data. By using function pointers one can embed intelligence into a data structure and flip its behavior at runtime. Function pointers can be passed as parameters to other functions, as qsort() does with the comparison function.

# *Laboratory Assignment 7: Working with text files*

## *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with text files and the way C handles text files.

## *Theoretical Summary*

In GNU/Linux everything is a file, including memory components and devices[20]. Text files are files that only contain printable characters. Not all characters are printable, some are audible or produce unprintable results, for example the Bell character. The characters starting from decimal 32 (SPACE) up to 127 (DEL) inclusive are printable:

[20] hard disks and disk partitions, memory sticks and cards, the sound card, the graphic card, the network card, they are all files.

```
root@parabolaiso / # man ascii
       Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
       ------------------------------------------------------------------
       000   0     00    NUL '\0'                    100   64    40    @
       001   1     01    SOH (start of heading)      101   65    41    A
       002   2     02    STX (start of text)         102   66    42    B
       003   3     03    ETX (end of text)           103   67    43    C
       004   4     04    EOT (end of transmission)   104   68    44    D
       005   5     05    ENQ (enquiry)               105   69    45    E
       006   6     06    ACK (acknowledge)           106   70    46    F
       007   7     07    BEL '\a' (bell)             107   71    47    G
       010   8     08    BS  '\b' (backspace)        110   72    48    H
       011   9     09    HT  '\t' (horizontal tab)   111   73    49    I
       012   10    0A    LF  '\n' (new line)         112   74    4A    J
       013   11    0B    VT  '\v' (vertical tab)     113   75    4B    K
       014   12    0C    FF  '\f' (form feed)        114   76    4C    L
       015   13    0D    CR  '\r' (carriage ret)     115   77    4D    M
       016   14    0E    SO  (shift out)             116   78    4E    N
       017   15    0F    SI  (shift in)              117   79    4F    O
       020   16    10    DLE (data link escape)      120   80    50    P
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 021 | 17 | 11 | DC1 (device control 1) | 121 | 81 | 51 | Q |
| 022 | 18 | 12 | DC2 (device control 2) | 122 | 82 | 52 | R |
| 023 | 19 | 13 | DC3 (device control 3) | 123 | 83 | 53 | S |
| 024 | 20 | 14 | DC4 (device control 4) | 124 | 84 | 54 | T |
| 025 | 21 | 15 | NAK (negative ack.) | 125 | 85 | 55 | U |
| 026 | 22 | 16 | SYN (synchronous idle) | 126 | 86 | 56 | V |
| 027 | 23 | 17 | ETB (end of trans. blk) | 127 | 87 | 57 | W |
| 030 | 24 | 18 | CAN (cancel) | 130 | 88 | 58 | X |
| 031 | 25 | 19 | EM  (end of medium) | 131 | 89 | 59 | Y |
| 032 | 26 | 1A | SUB (substitute) | 132 | 90 | 5A | Z |
| 033 | 27 | 1B | ESC (escape) | 133 | 91 | 5B | [ |
| 034 | 28 | 1C | FS  (file separator) | 134 | 92 | 5C | \ '\\' |
| 035 | 29 | 1D | GS  (group separator) | 135 | 93 | 5D | ] |
| 036 | 30 | 1E | RS  (record separator) | 136 | 94 | 5E | ^ |
| 037 | 31 | 1F | US  (unit separator) | 137 | 95 | 5F | _ |
| 040 | 32 | 20 | SPACE | 140 | 96 | 60 | ‘ |
| 041 | 33 | 21 | ! | 141 | 97 | 61 | a |
| 042 | 34 | 22 | " | 142 | 98 | 62 | b |
| 043 | 35 | 23 | # | 143 | 99 | 63 | c |
| 044 | 36 | 24 | $ | 144 | 100 | 64 | d |
| 045 | 37 | 25 | % | 145 | 101 | 65 | e |
| 046 | 38 | 26 | & | 146 | 102 | 66 | f |
| 047 | 39 | 27 | ’ | 147 | 103 | 67 | g |
| 050 | 40 | 28 | ( | 150 | 104 | 68 | h |
| 051 | 41 | 29 | ) | 151 | 105 | 69 | i |
| 052 | 42 | 2A | * | 152 | 106 | 6A | j |
| 053 | 43 | 2B | + | 153 | 107 | 6B | k |
| 054 | 44 | 2C | , | 154 | 108 | 6C | l |
| 055 | 45 | 2D | - | 155 | 109 | 6D | m |
| 056 | 46 | 2E | . | 156 | 110 | 6E | n |
| 057 | 47 | 2F | / | 157 | 111 | 6F | o |
| 060 | 48 | 30 | 0 | 160 | 112 | 70 | p |
| 061 | 49 | 31 | 1 | 161 | 113 | 71 | q |
| 062 | 50 | 32 | 2 | 162 | 114 | 72 | r |
| 063 | 51 | 33 | 3 | 163 | 115 | 73 | s |
| 064 | 52 | 34 | 4 | 164 | 116 | 74 | t |
| 065 | 53 | 35 | 5 | 165 | 117 | 75 | u |
| 066 | 54 | 36 | 6 | 166 | 118 | 76 | v |
| 067 | 55 | 37 | 7 | 167 | 119 | 77 | w |
| 070 | 56 | 38 | 8 | 170 | 120 | 78 | x |
| 071 | 57 | 39 | 9 | 171 | 121 | 79 | y |
| 072 | 58 | 3A | : | 172 | 122 | 7A | z |
| 073 | 59 | 3B | ; | 173 | 123 | 7B | { |
| 074 | 60 | 3C | < | 174 | 124 | 7C | \| |

```
075   61    3D    =                          175   125   7D    }
076   62    3E    >                          176   126   7E    ~
077   63    3F    ?                          177   127   7F    DEL
```

The <stdio.h> interface provides functions for working with text files:

```
#include <stdio.h>

extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;

FILE *fopen(const char *path, const char *mode);
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int fclose(FILE *fp);
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

The input, usually the keyboard is predefined as the stdin file stream. The stdout normally is the default display and stderr the device for displaying errors (similar to stdout). The following example counts the number of characters read from stdin:

```
#include <stdio.h>

int main (void) {
        unsigned int n=0;
        char c;

        while ((c=fgetc(stdin)) != EOF) n++;

        fprintf(stdout, "Just␣read␣%u␣chars␣from␣the␣input␣file\n", n);

        return n;
}
```

The EOF character is encoded as Ctrl-D in GNU/Linux and ESC as Ctrl-C.

```
root@parabolaiso / # ./charcount
Typed from the keyboard
and so on..
Just read 36 chars from the input file
root@parabolaiso / #
```

Alternatively the input can be redirected from a file:

```
root@parabolaiso / # ./charcount < charcount.c
Just read 185 chars from the input file
root@parabolaiso / #
```

The fprintf() and fscanf() calls work just like regular printf() and scanf(). A file opened with fopen() must be closed with fclose():

```
#include <stdio.h>

int main (void) {
        unsigned int n=0;
        char c;
        FILE *fp;

        fp = fopen("charcount.c", "r");
        if (fp == NULL) { perror("fopen()"); return -1; }

        while ((c=fgetc(fp)) != EOF) n++;

        fclose(fp);

        fprintf(stdout, "Just␣read␣%u␣chars␣from␣the␣input␣file\n", n);

        return n;
}
```

```
root@parabolaiso / # ./charcount2
Just read 291 chars from the input file
root@parabolaiso / # cat charcount2.c | wc -c
291
root@parabolaiso / #
```

Notice that the file has been opened for reading only. The fopen() function has different modes for opening a file:

Table 1: fopen() modes **for** opening a file

| | |
|---|---|
| "r" | Open text file for reading. |
| "r+" | Open for reading and writing. |
| "w" | Truncate file to zero length or create text file for writing. |
| "w+" | Open for reading and writing. |
| "a" | Open for appending (writing at end of file). The file is created if it does not exist. |
| "a+" | Open for reading (at the beginning of the file) and appending (writing at end of file). |

## *Assignment Breakdown and Methodology*

1. Carefully run and study the examples given in the Theoretical Summary.

2. Write a program to read the standard input file stdin until EOF is signaled and count the frequencies of appearances for all the characters met. Use an array to retain the frequencies and print it on the standard output stdout when finished. You may reuse the code snippets from the Theoretical Summary.

3. Write a tool to convert from UNIX text files to DOS text files. UNIX uses newlines encoded by only one character "\n", and DOS uses two characters "\r\n". Use the stdin and stdout file streams.

4. Modify the previous tool so that if any arguments are given in the command line they should be treated as files ready to be processed. The "*.c" argument will be expanded by the shell into a list with all the source files in the working directory.

5. The SubRip file format is one of the most basic of all subtitle formats. A text file with ".srt" extension consists of several entries of the form:

   (a) A numeric counter

   (b) The time that the subtitle should appear on the screen, followed by "–>" and the time it should disappear

   (c) Subtitle text itself on one or more lines

   (d) A blank line containing no text, indicating the end of this subtitle

   The following example uses HTML tags to mark *italic text*, "**bold**" can be used for **bold** and "<u>text</u>" for underlining:

```
1
00:00:30,447 --> 00:00:31,740
<i>Earth-That-Was...</i>

2
00:00:31,782 --> 00:00:34,910
<i>...could no longer sustain
our numbers, we were so many.</i>

3
00:00:38,455 --> 00:00:40,624
<i>We found a new solar system.</i>
```

```
4
00:00:41,124 --> 00:00:43,919
<i>Dozens of planets
and hundreds of moons.</i>
```

Write a program to parse the ".srt" text file, remove the text formatting tags and shift the timings with a number of seconds given as command line arguments. Use fscanf() in order to parse the time.

## *Questions*

1. What is the difference between text files and other files?
2. Name an example of ASCII code which does not have printable functionality.
3. Why would anyone use non-printable ASCII codes?
4. Which file streams are opened by default by <stdio.h>?
5. A file opened by fopen() is automatically closed by the operating system when the program exits. Why bother using fclose()?
6. Why does fclose() return a value?
7. There is a limit of how many files can be opened simultaneously. What happens if a program has a loop for opening files but forgets to close them accordingly?
8. How is fscanf() different than scanf() and sscanf()?
9. How is fprintf() different than printf() and sprintf()?
10. What is the difference between the "r+" and "a+" modes for fopen()?

## *Home Exercises*

1. Write a program to replace all lowercase letters from a file containing uppercase letters.
2. Write a program to print a table with single precision values for the trigonometric functions sin(), cos(), tan() and $\tan()^{-1}$ from the math library interfaced by <math.h>. There is no reason to use double precision unless you really need it. Single precision is faster. When linking the executable you need to specify the "m" library or use your own implementation. Use fprintf().

```
#include <math.h>

double tan(double x);
float tanf(float x);
long double tanl(long double x);

Link with -lm.
```

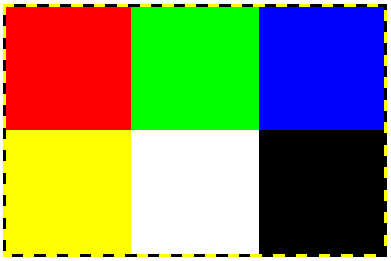
The file should have the following format:

```
================================================================================
Degrees    Radians        sin(x)         cos(x)         tan(x)          ctan(x)
================================================================================
0.0          0.0        0.0             1.0           0.0                -
0.1          0.175      0.175           0.9998        0.0175      57.2900
```

To convert from degrees to radians use $radians = degress \times PI/180$.
It is up to you how you define $\pi = 3.14$ but if you search the
<math.h> header you will find:

```
root@parabolaiso / # grep PI /usr/include/math.h
# define M_PI 3.14159265358979323846 /* pi */
# define M_PI_2 1.57079632679489661923 /* pi/2 */
# define M_PI_4 0.78539816339744830962 /* pi/4 */
# define M_1_PI 0.31830988618379067154 /* 1/pi */
# define M_2_PI 0.63661977236758134308 /* 2/pi */
# define M_2_SQRTPI 1.12837916709551257390 /* 2/sqrt(pi) */
# define M_PIl 3.14159265358979323846264338327950288 4L /* pi */
# define M_PI_2l 1.57079632679489661923132169163975144 2L /* pi/2 */
# define M_PI_4l 0.78539816339744830961566084581987572 1L /* pi/4 */
# define M_1_PIl 0.31830988618379067153776752674502872 4L /* 1/pi */
# define M_2_PIl 0.63661977236758134307553505349005744 8L /* 2/pi */
# define M_2_SQRTPIl 1.12837916709551257389615890312154517 2L /* 2/sqrt(pi) */
root@parabolaiso / #
```

3. The NetPBM P3 file format is a text file format of the form given by:



```
P3
3 2
255
255   0   0     0 255   0     0   0 255
255 255   0   255 255 255     0   0   0
```

where P3 means ASCII contents, "3" and "2" represent the number of columns and the number of rows from the picture, "255" is the

maximum value for a color level and pixels are encoded one by one using Red, Green and Blue color levels for each. For example "255 0 0" means Red is full and Green and Blue are missing. White is obtained by setting all three color components to their maximum level.

The file extension is ".ppm" and it shows a bitmap image with 6 pixels, 3 pixels for each of the 2 rows (3×2): the colors from the upper left corner, row by row, to the lower right corner are: red, green, blue and yellow, white, black.

(a) Write a program which produces a 1000×1000 pixels picture showing a vertical gradient for black, from dark (left) to white (right).

(b) Modify the program to produce a violet gradient, knowing that violet is made from equal portions of red and blue.

(c) All values from 0 to 255 are stored in one unsigned byte, using a total of 8 bits. The rightmost bit is the least significant bit. The leftmost bit is the most significant bit. Any ASCII code can be encoded using 8 bits. Write a program to hide a text message in the violet gradient using the least significant bit from 8 pixels from the Green component for every **char** value. Your program must be able to extract the message from any ".ppm" file provided as command line argument. Use bit level operations for bit manipulation.

4. Mimesis criticism[21] is a method for detecting imitations in literature. The Hamming distance between two strings of equal length is the number of positions at which the corresponding letters are different. Write a program for computing the Hamming distance between two text files. The filenames are to be given as command line arguments. Can this be used for text files containing source code? What about approximate string matching?

## *Take-Home Idea*

Text files only contain printable ASCII codes but any file can be processed using character input and output with fgetc(), regardless of the opening mode. Text mode "t" is the default file opening mode for fopen(). Files can also be processed without using the FILE **struct** from <stdio.h> by directly using the GNU/Linux system call wrappers open(), read(), write() and close().

[21] Virgil's Aeneid is an imitation of Homer's Iliad and Odyssey, but Virgil rewrites the Greek legend in important ways. The first six books of the Aeneid, describing Aeneas's journey to the future site of Rome, are an imitation of the Odyssey. The second six, describing the battle for the region in which Rome is destined to be situated, imitate the Iliad. Virgil thus reverses Homer's chronology: the Homeric story is about leaving home and returning home; Virgil's story is about leaving home and making a new and better home. Roman culture is finally on pair or even superior to the dominant Greek culture. The Aeneid comprises 9,896 lines in dactylic hexameter and becomes the national Roman epic about the founding myth of Rome. By carefully imitating both Homer and Virgil, using a list with recorded teachings and rhetorolects from the epistles, the Gospel authors portrayed their hero as superior to the worldly kingdom of the Augustan Golden Age. Their hero also had to be much more appealing than the Greek super-heroes, because their precepts ought to be superior. They also directly quote from older Sanskrit texts, without being ignorant of human genetics and the veiled language of mythology. Some versions resemble intertextual similarities with the Indian King Ashoka's missionary communities in Egypt. Others have been partially written in the $5^{th}$ century. Epic works are no more historical than Jewish folklore from the Bronze Age period; see Dennis Ronald MacDonald on epics writing.

# *Laboratory Assignment 8: Working with binary files*

### *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with binary files and the way C handles files in binary mode.

### *Theoretical Summary*

All files on binary computers are binary files, but some may be restricted to printable ASCII codes and designated as text files. Binary files must be opened with the binary flag added to the mode string. For example

```
#include <stdio.h>

FILE *fp;

fp = fopen("somefile.bin", "rb");
if (fp == NULL) { perror("fopen()"); return -1; }
```

opens the file "somefile.bin" in binary mode. This means reading and writing has to be done with fread() and fwrite() having the following interface:

```
root@parabolaiso / # man fread
       #include <stdio.h>

       size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
       size_t fwrite(const void *ptr, size_t size, size_t nmemb,
       FILE *stream);

       int fseek(FILE *stream, long offset, int whence);
       long ftell(FILE *stream);
       void rewind(FILE *stream);
       int fgetpos(FILE *stream, fpos_t *pos);
```

```
int fsetpos(FILE *stream, fpos_t *pos);
```

The fread() and fwrite() functions read and write nmemb memory blocks of a given size. For example in order to write 2 blocks of 64 bits (8 bytes) one would call

```
unsigned long long buf[2] = { 0x6c61766975716520 , 0x7469206564616d20 };
fwrite(buf, sizeof(unsigned long long), 2, fpdest);
```

File positioning can be operated with fseek(), ftell (), rewind(), fgetpos() and fsetpos(). These functions return or set the file position in terms of byte index either in absolute value or relative to a predefined position in the file. The whence can be set to SEEK_SET, SEEK_CUR, or SEEK_END for the start of the file, the current position indicator, or for the end of the file. Without the whence file, positioning is an index relative to the beginning of the file. The start of the file is pos = 0.

A file edited in "a+b" mode is opened in binary mode and positioned at the start of the file for reading, and at the end of the file for writing (appending). For example

```
#include <stdio.h>

int main (void) {
        FILE *fp;
        unsigned long pos;

        fp=fopen("somefile.bin", "a+b");
        if (fp == NULL) { perror("fopen()"); return -1; }

        pos = ftell(fp); printf("File position is %lu\n", pos);
        fwrite(&a, sizeof(int), 1, fp);
        pos = ftell(fp); printf("File position is %lu\n", pos);
        fread(&a, sizeof(int), 1, fp);
        pos = ftell(fp); printf("File position is %lu\n", pos);

        // pos_t in fgetpos() is a structure, it is not meant for arithmetic
        fclose(fp); return 1;
}
```

with output

```
root@parabolaiso / # ./filepos
File position is 0
File position is 4
File position is 4
root@parabolaiso / #
```

records and prints the file positioning index in bytes before and after the fread() and fwrite() calls.

The following example records and plays back 5 seconds of audio using the "/dev/dsp" device file which can be emulated by modern GNU/Linux sound drivers by requesting "sudo modprobe snd-pcm-oss":

```
#include <stdio.h>
#include <unistd.h>
/* how many seconds  */
#define DURATION 5
/* the sampling rate */
#define RATE 8000
/* sample size: 8 bits*/
#define SIZE 8
/* 1 = mono */
#define CHANNELS 1

/* this buffer holds the digitized audio */
unsigned char
    buf[DURATION*RATE*SIZE*CHANNELS/8];

int main (int argc, char *argv[]) {
    FILE *soundcard;
    int n, filenum;

    /* sudo modprobe snd-pcm-oss creates the /dev/dsp device file */
    if (argc == 2) soundcard = fopen(argv[1], "r+b");
    else soundcard = fopen("/dev/dsp", "r+b");
    if (soundcard == NULL) {
        perror("Can␣not␣talk␣to␣soundcard␣:X..");
        return (1); /* main() returns 0 if OK */
    }

    /* obtain struct FILE._file (short) */
    filenum = fileno(soundcard);
    printf("Recording␣for␣%d␣seconds␣(mic):",
        DURATION);

    /* record sound */
    n = read(filenum, buf, sizeof(buf));
    if (n != sizeof(buf)) perror("wrong␣size");

    printf("\nYou␣said:\n"); /* play it back */
    n = write(filenum, buf, sizeof(buf));
```

```
    if (n != sizeof(buf)) perror("wrong size");

    fclose(soundcard);

    return 1;
}
```

### *Assignment Breakdown and Methodology*

1. Carefully study and experiment with the examples from the Theoretical Summary.
2. The following example uses NSA published source code for their SPECK cryptographic cipher.

```
#include <x86intrin.h>
#include <stdio.h>
#include <string.h>

#define LCS _lrotl //left circular shift
#define RCS _lrotr //right circular shift
#define u64 unsigned long long
#define R(x,y,k) (x=RCS(x,8), x+=y, x^=k, y=LCS(y,3), y^=x)
#define INV_R(x,y,k) (y^=x, y=RCS(y,3), x^=k, x-=y, x=LCS(x,8))

void Speck128ExpandKeyAndEncrypt(u64 pt[], u64 ct[], u64 K[])
{
        u64 i,B=K[1],A=K[0];
        ct[0]=pt[0]; ct[1]=pt[1];
        for(i=0; i<32; i++) {R(ct[1], ct[0], A); R(B, A, i);}
}

void Speck128Encrypt(u64 pt[], u64 ct[], u64 k[])
{
        u64 i;
        ct[0]=pt[0]; ct[1]=pt[1];
        for(i=0; i<32; i++) R(ct[1], ct[0], k[i]);
}

void Speck128Decrypt(u64 pt[], u64 ct[], u64 K[])
{
        long long i;
        for(i=31; i>=0; i--) { INV_R(ct[1], ct[0], K[i]); }
}
```

```
static void speck_keyexpand(u64 K[]) // OK
{
        u64 tmp[32], *p;
        memcpy(tmp,K,sizeof tmp);
        // K[0] stays the same
        u64 i;
        for (i=0; i<(31);) {
                p = tmp + (1+(i%(2-1)));
                R(*p,tmp[0],i);
                ++i;
                K[i] = tmp[0];
        }
}

int main (void) {
        u64 k[32];
        k[1] = 0x0f0e0d0c0b0a0908;
        k[0] = 0x0706050403020100;

        u64 pt[2];
        pt[1] = 0x6c61766975716520;
        pt[0] = 0x7469206564616d20;
        u64 ctverify[2];
        ctverify[1] = 0xa65d985179783265;
        ctverify[0] = 0x7860fedf5c570d18;
        u64 ct[] = {0x0, 0x0};

        //Speck128ExpandKeyAndEncrypt(pt, ct, k);
        speck_keyexpand(k);
        Speck128Encrypt(pt, ct, k);

        printf("ct[] = %llx %llx\n", ct[0], ct[1]);

        if ((ct[1] == ctverify[1]) && (ct[0] == ctverify[0]))
                printf("Encryption Test Vector OK\n");

        Speck128Decrypt(pt, ct, k);

        printf("ct[] = %llx %llx\n", ct[0], ct[1]);

        if ((ct[1] == pt[1]) && (ct[0] == pt[0]))
                printf("Decryption Test Vector OK\n");

        return 0;
```

}

Modify the example using fread() and fwrite() to encrypt a file given as parameter in the command line. Read the password from the keyboard and convert it into a 128 bit key made of two **unsigned long long** pieces. Use a command line switch to decide if to encrypt or decrypt the file.

3. A mobile phone repairing shop uses a small database for keeping track of the business. The database has the following structure:

| i | Repair type | IMEI | Price | Investment | Profit | Total profit | Total influx |
|---|---|---|---|---|---|---|---|
| 1 | Display replacement | 123456789012345 | 45 | 35 | 10 | 20 | 60 |
| 2 | Battery replacement | 123456789012347 | 15 | 5 | 10 | - | - |

The IMEI number identifies the phone and the producer and can be blacklisted. Use proper composite variables and a binary file with ".bdb" extension in order to implement the database. Display a menu on the screen for the user to select the desired operation. The user must be able to add entries to the database, delete entries from the database, display all entries belonging to an IMEI and mark or unmark a phone as blacklisted. Use fread() and fwrite() for reading and writing the database file and file positioning functions to move around and locate entries (fseek(), ftell(), fgetpos(), fsetpos()). The "Total profit" and "Total influx" fields must be automatically updated.

## *Questions*

1. What is the difference between a binary file and a text file?
2. Why do you think C has different approaches for binary mode and text mode file operations?
3. How is binary file mode different than text file mode in terms of fopen()?
4. How is binary file mode different than text file mode in terms of reading and writing?
5. What is the fundamental difference between fread() and fprintf()?
6. What is the fundamental difference between fwrite() and fscanf()?
7. Why is it not recommended to print pos_t values?
8. How can we print the position in a file without using the pos_t structure from fgetpos()?

9. Can rewind() be expressed in terms of fseek()?

10. How is "a+b" mode different than "r+b" mode?

## *Home Exercises*

1. Write a program to record 1 minute of conversation, encrypt it and save it in a ".snd" sound file. Add a command line switch to the program to enable it to play the file. You can reuse the examples from the Assignment Breakdown and Methodology.

2. Enhance the phone repairing database program in order to be able to enable and disable database encryption. Use the examples from the Assignment Breakdown and Methodology and independent compilation to organize your source code files.

3. Write a file archiving program driven by command line arguments. The program must be able to concatenate an arbitrary number of binary files in a single ".arh" file. The user must be able to extract any file from the archive and list the archive contents (the file names). When given the "*" from the command line the GNU/Linux shell will expand the list of files from the working directory in a string containing their names separated by spaces and provide that string as argument to your program. Basically, all the file names from the working directory will be command line arguments to your program helping you archive the entire working directory.

   (a) write the program using binary file mode with fread(), fwrite() and fseek()-like functions.

   (b) write the program using text file mode with fgetc() and fputc() and other functions if necessary.

```
root@parabolaiso / # man fputc
        #include <stdio.h>
        int fputc(int c, FILE *stream);
```

## *Take-Home Idea*

All files are binary files because binary hardware is cheaper and more reliable. Working with files in binary "b" mode allows for reading and writing byte blocks at byte level. Since a disk sector usually has 512 or more bytes it is a good idea to read and write binary files in terms of sector sizes.

# *Laboratory Assignment 9: Divide and Conquer*

### *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with recursion and the divide and conquer programming technique.

### *Theoretical Summary*

A function is recursive if it contains a call to itself:

```
int recursive_add(int n) {
        if (n == 1) return 1;
        else return n + recursive_add(n-1);
}
```

Recursion can be used to implement mathematical functions that are naturally recursive. For example the code above corresponds to the formula in (1):

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

Revisiting the Fibonacci formula in (4) $F(N) = 0, 1, 1, 2, 3, 5, 8, \ldots$

$$f(N) = \begin{cases} 0 & N = 0 \\ 1 & N = 1 \\ f(N-2) + f(N-1) & N \geq 2 \end{cases} \tag{2}$$

the function can be implemented by the recursive function:

```
int recursive_fib(int n) {
        if (n <= 0) return 0; /* skip errors for n < 0 */
        if (n == 1) return 1;
        if (n > 1) return (recursive_fib(n-1) + recursive_fib(n-2));
}
```

having the recursive call tree for recursive_fib(5):

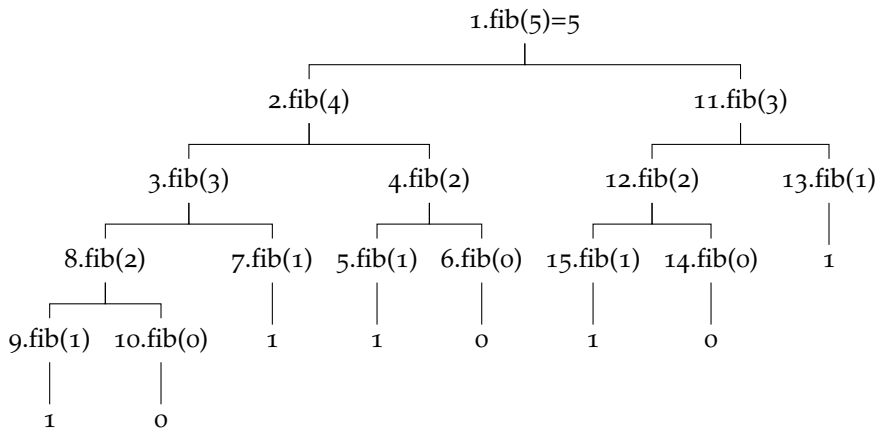



Figure 2: Recursion tree for the recursive_fib(5) recursive Fibonacci function implementation

The whole idea of divide and conquer is to recursively split a problem into smaller subproblems. By doing so, repeating the same strategy for solving the small subparts, the global solution can be reconstructed based on the smaller, conquered partial solutions.

For example, in order to determine the maximum and the minimum value from a set of numbers, a recursive divide and conquer implementation would require that the set of numbers is recursively split into halves and the maximum and the minimum values are isolated within the halves themselves. The global minimum and maximum can be obtained during the problem reconstruction process by selecting the right value from the left and right candidates.

The linear method for finding the minimum value from an given array is shown in:

```
#include <stdio.h>

#define N 10

int main (void) {
        int elements[N] = {2, 7, 3, 4, 1, 9, 12, 6, 5, 0};

        int i;
        for (i=1; i<N-1; i++)
                if ((elements[i-1] >= elements[i]) && (elements[i] <= elements[i+1]))
                printf("Local minimum: %d\n", elements[i]);
}
```

A divide a conquer method for finding a local minimum from the array is shown in:

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

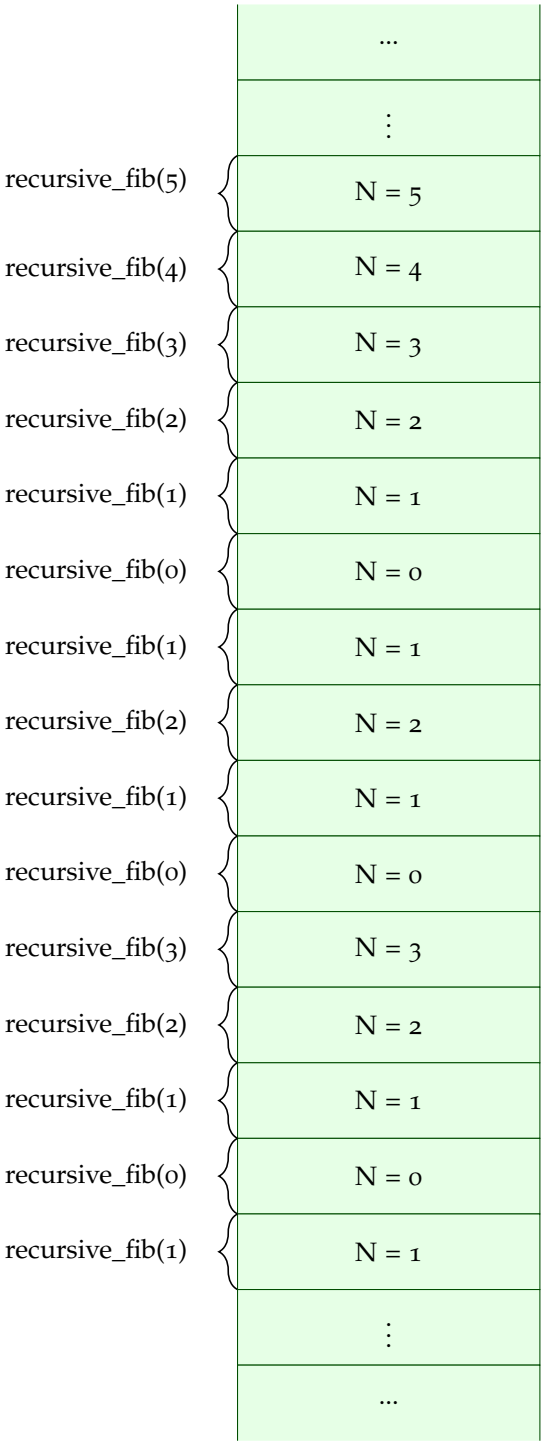



Figure 3: Stack parameters pushed onto the stack with every stack frame activation record created for the call to recursive_fib(N)

```
#define N 10

int localmin(int elements[N], int i, int j) {
        int m = (i+j)/2;
        if ((elements[m-1] >= elements[m]) &&
                (elements[m] <= elements[m+1])) {
                        printf("Local minimum: %d\n", elements[m]);
                        return m;
        }
        else if (elements[m-1] < elements[m]) return localmin(elements, i, m-1);
        else if (elements[m] > elements[m+1]) return localmin(elements, m+1, j);
}

int main (void) {
        int elements[N] = {2, 7, 3, 4, 1, 9, 12, 6, 5, 0};

        localmin(elements, 0, N);
}
```

The local minimum is a minimum which finds a minimum but the detected value is not necessary the global value for the minimum. If the integer values are topological heights, the algorithm finds a valley which is not necessary the deepest one from the array.

A divide and conquer algorithm for finding the global minimum is given in:

```
#include <stdio.h>

#define N 10

int globalmin(int elements[N], int i, int j) {
        int a, b, m;

        if (i==j) return elements[i];
        else if (i==j-1) {
                if (elements[i] < elements[j]) return elements[i];
                else return elements[j];
        } else {
                m = (i + j)/2;

                a=globalmin(elements, i, m);
                b=globalmin(elements, m+1, j);

                if (a > b) return b;
```

```
                else return a;
        }
}

int main (void) {
        int elements[N] = {2, 7, 3, 4, 1, 9, 12, 6, 5, 0};

        printf("Global minimum is: %d\n", globalmin(elements, 0, N));
}
```

The Tower of Hanoi problem for 2 disks is given in Fig.4 with the example for 3 disks in Fig.5.

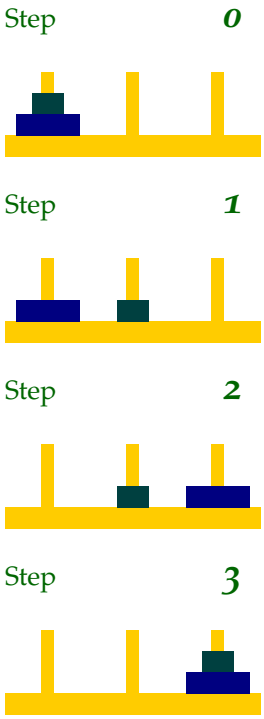



Figure 4: Tower of Hanoi for 3 pegs and 2 disks

The disks must be moved from the first peg to the last peg without ever inserting a larger disk over a smaller disk. In order to conquer the problem the challenge must always be reduced to the case with 2 disks and 1 disk. This is shown in the solution:

```
#include <stdio.h>

#define N 3

void movedisks(int ndisks, char fromPeg, char toPeg, char auxPeg) {
        if (ndisks == 1) printf("Move disk 1 from [%c] to [%c]\n", fromPeg, toPeg);
```

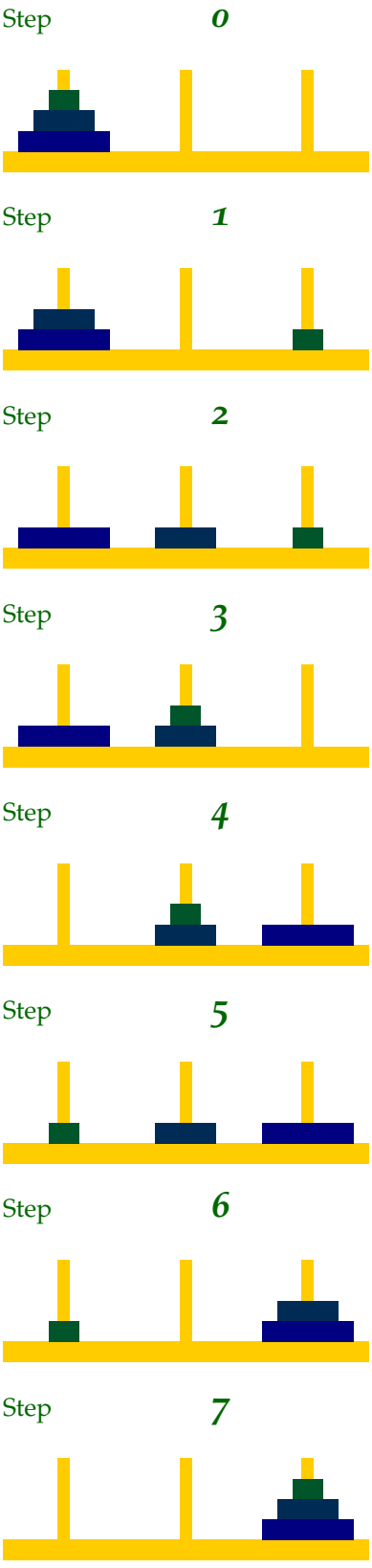



Figure 5: Tower of Hanoi for 3 pegs and 3 disks

```
        else {
                movedisks(ndisks - 1, fromPeg, auxPeg, toPeg);
                printf("Move disk %d from [%c] to [%c]\n", ndisks, fromPeg, toPeg);
                movedisks(ndisks - 1, auxPeg, toPeg, fromPeg);
        }
}

int main(void) {
        movedisks(N, 'A', 'C', 'B');
}
```

## *Assignment Breakdown and Methodology*

1. Carefully study and experiment with the examples from the Theoretical Summary.

2. The SVG file format is a text file of the form:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
<rect x="25" y="25" width="200" height="200" fill="lime" stroke-width="4" stroke="pink" />
<circle cx="125" cy="125" r="75" fill="orange" />
<polyline points="50,150 50,200 200,200 200,100" stroke="red" stroke-width="4" fill="none" />
<line x1="50" y1="50" x2="200" y2="200" stroke="blue" stroke-width="4" />
</svg>
```

   Write a C program based on the Hanoi Tower example from the Theoretical Summary which draws the 3 peg examples for 2, 3 and 4 disks in SVG format, using the "<rect x=...y=... width=... height=...>" SVG directive. Use a web browser or an image viewer to render the SVG on the screen.

3. Write a recursive C program to generate the Koch snowflake. The number of iterations are to be given from the command line as arguments. The Koch fractal of order 0 is a straight line. Use three Koch fractals of order 0 to draw an equilateral triangle. After 1 iteration, every triangle segment is divided into 3 subsegments and a new equilateral triangle 3 times smaller is drawn on the outside of the original triangle in the middle of the original triangle sides. Output the result in a SVG file.

   (a) Start by understanding the requirements. You need a recursive function to compute Koch coordinates of a given order.

   (b) Start by drawing a straight line. Draw a Koch fractal[22] of order 0 in a SVG file as a simple line segment parallel with the X axis.

   (c) Test the Koch fractal drawing function for higher orders.

[22] If you want to see a 3D fractal made up from cones arranged in spirals check the broccoli romanesco plant.

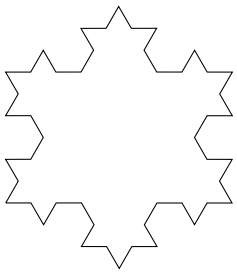


Figure 6: 3 Koch fractals of order 2, also known as the Koch snowflake of order 2

(d) Use the function to draw an equilateral triangle in SVG format and thoroughly test it.

## *Questions*

1. The core strategy of the Divide and Conquer technique is to reduce the problem to one or few more simple cases. Can this also be expressed in an iterative manner, without using recursion?

2. Any recursive function can be rewritten in iterative manner. What is the preferred approach for Divide and Conquer? Please explain thoroughly.

3. Is it easier to describe a Divide and Conquer problem in an iterative manner? Why?

4. How many leaves does a recursive call tree for the factorial function have?

5. When is Divide and Conquer preferred over the linear approach?

6. When is the linear approach preferred over the Divide and Conquer technique?

7. When is the recursive implementation preferred over the iterative Divide and Conquer implementation?

8. When is the iterative Divide and Conquer implementation preferred over the recursive implementation?

9. In prefix notation 3 + 4 is written as + 3 4. The operators prefix the operands. How many nodes does a recursive call tree for computing the result of a prefix format expression have?

10. For prefix format which is faster: a recursive implementation or an iterative non-recursive one?

## *Home Exercises*

1. The construction of the Sierpinski carpet begins with a square. The square is cut into 9 congruent subsquares in a 3-by-3 grid, and the central subsquare is removed. The same procedure is then applied recursively to the remaining 8 subsquares, ad infinitum. Write a C program to draw the Sierpinski carpet in a SVG file. The number of iterations are to be given as command line arguments.

2. Use divide and conquer to find the peak values from a 2D array of given size populated with random values. The random() function is accessible via <stdlib.h>.

3. Using Divide and Conquer write a non-recursive iterative solution to the Towers of Hanoi problem.

4. Using the source code from the Theoretical Summary write a C program to draw the recursive call tree for the Tower of Hanoi problem in a SVG file.

## *Take-Home Idea*

Divide and Conquer is a pattern for recursively breaking down a problem into two or more subproblems of the same type, until they become simple enough to be conquered directly. By repeating the same strategy to solve the small subparts, the global solution can be reconstructed based on the partial solutions. The Tower of Hanoi logical challenge or finding the peak value from a list of numbers are classic examples.

# *Laboratory Assignment 10: Greedy*

### *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with the Greedy programming technique and problem solving strategy.

### *Theoretical Summary*

The Greedy technique is a trade-off between functionality and speed. Greedy allows us to find a globally suboptimal solution for a given problem. For example, given an array of numbers, we are asked to find the peak value from the list. An optimal solution would find the exact maximum value from the given array of numbers. Greedy, however, allows us to select a number which is large enough, but not necessary the biggest. For instance, the number may be a maximum within the left half of the array, but regarding the right half, we still remain uncertain.

Let us revisit the previous chapter's example of minimum finding:

```
#include <stdio.h>

#define N 10

int localmin(int elements[N], int i, int j) {
        int m = (i+j)/2;
        if ((elements[m-1] >= elements[m]) &&
                (elements[m] <= elements[m+1])) {
                        printf("Local minimum: %d\n", elements[m]);
                        return m;
        }
        else if (elements[m-1] < elements[m]) return localmin(elements, i, m-1);
        else if (elements[m] > elements[m+1]) return localmin(elements, m+1, j);
}
```

```
int main (void) {
        int elements[N] = {2, 7, 3, 4, 1, 9, 12, 6, 5, 0};

        localmin(elements, 0, N);
}
```

This finds 3 to be the local minimum. But it is obvious that 1 is also a valley with a local minimum; and, 1, is lower than 3. So Greedy allows us to find a close to optimal solution, but there is no guarantee that the solution is also global. So, instead of trying to look first for a minimum in the left half, we could have tried locamin(elements, 0, N−1) and find 1 as the local minimum! The gain is that it finds a minimum faster, and that it might be the global one.

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? This is called the Travelling Salesman Problem or TSP. Let us use a matrix to encode the links between the cities, their direction and their length.

| - | Timişoara | Oradea | Cluj-Napoca | Iaşi | Bucureşti |
|---|---|---|---|---|---|
| Timişoara | - | 174 | 315 | 634 | 544 |
| Oradea | 174 | - | 152 | - | 595 |
| Cluj-Napoca | 315 | 152 | - | 393 | - |
| Iaşi | 634 | 544 | 393 | - | - |
| Bucureşti | 544 | 595 | - | 388 | - |

Table 2: The adjacency matrix for town to town connectivity in Romania

The fact that there is a link from Timişoara to Oradea does not necessarily imply that there is a link between Oradea and Timişoara. The links are not automatically bidirectional unless specified so in the connectivity matrix. A collection of nodes and arches that link them is called a graph. Because there are weight values associated with the distance between the towns, the graph is called a weighted graph, and because the links are not automatically bidirectional, the graph is an oriented graph.

The oriented weighted graph in Tab.2 has a connection from Bucureşti to Iaşi measuring 388 km but there is no connection between Iaşi and Bucureşti and this is marked with "-" which can be encoded by 0 or a negative value in the C source code.

Having Timişoara as the starting point in Tab.2, what is the shortest possible route that visits each city exactly once and returns back to Timişoara? A Greedy solution simplifies the problem of choosing of the shortest nearby route. So, from Timişoara, the next visited City would be Oradea because it is only 174 km away. From Oradea the shortest route is to Cluj-Napoca. From Cluj-Napoca we skip the

previously visited cities in order to get to Iaşi via the 393 km route. We cannot go from Iaşi to Bucureşti so either we give up or be less Greedy by going back one city and choosing a different route. But from Cluj-Napoca there is no new route to unvisited cities so we resume from Oradea and choose Iaşi instead of Cluj-Napoca. From Iaşi we can go to Cluj-Napoca and from there to Bucureşti.

In order for Greedy to work we remove the orientation from the graph and we consider that any link between two cities is bidirectional. So we produce Tab.3.

| - | Timişoara | Oradea | Cluj-Napoca | Iaşi | Bucureşti |
|---|---|---|---|---|---|
| Timişoara | - | 174 | 315 | 634 | 544 |
| Oradea | 174 | - | 152 | 544 | 595 |
| Cluj-Napoca | 315 | 152 | - | 393 | 449 |
| Iaşi | 634 | 544 | 393 | - | 388 |
| Bucureşti | 544 | 595 | 449 | 388 | - |

Table 3: The bidirectional adjacency matrix for town to town connectivity in Romania

Now the simple Greedy technique will always work: we start in Timişoara, then move to Oradea, from Oradea we move to Cluj-Napoca, from there to Iaşi and from Iaşi to Bucureşti. There is no guarantee that this is the global shortest path, but it is a short path and it does satisfy the requirements. So Greedy is a shortcut to quickly find a solution which may or may not coincide with the best version. A responsive GPS interface may resort to Greedy to quickly spot a solution while doing more extensive calculations in the background. It is always pleasant for the software user to naturally interact with the system without having to wait.

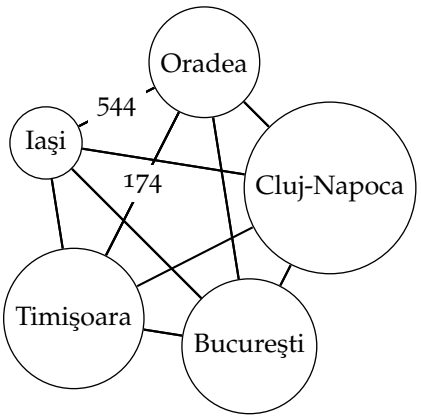



Figure 7: A partially weighted bidirectional graph for Tab.5. An oriented graph has arrow-pointed arches.

### *Assignment Breakdown and Methodology*

1. Carefully study and experiment with the examples from the Theoretical Summary.
2. Write a program to find the shortest path from Oradea to all the other cities without visiting any city twice, using Greedy and Tab.3. Read the adjacency connectivity matrix from a text file given as command line argument. Print the total distance traveled.
3. Modify the program to print all possible routes and check if the Greedy solution was the best.
4. Use Greedy and Divide and Conquer to find the peak values from a list of numbers in a 1D array.

### *Questions*

1. What is the fundamental principle behind the Greedy tactics?
2. Why is Greedy preferred over extensive calculus?
3. When is Greedy preferred over extensive calculus?
4. Is Greedy necessary for the examples in the Theoretical Summary?
5. What is the difference between a local optimum and a global optimum?
6. Is a global optimum less optimal than a local optimum?
7. Is a local optimum more optimal than a global optimum?
8. How many global optimums can a problem have?
9. How many local optimums can a problem have?
10. Can a global optimum also be a local optimum? When?

### *Home Exercises*

1. Solve the Tab.2 oriented graph problem with Greedy using the remarks from the Theoretical Summary.
2. Use Greedy to determine the minimum number of coins to give while making change. Your currency supports coins with values 1, 5, 10, 20. The change value is given as input. Print the solution on the screen.

3. Write a program to determine the optimal compression scheme for a text file. Count the frequency of the characters that appear in the file and sort them in a table. Build the Huffman encoding tree starting with the 2 lowest frequency letters, arranging them as leaves in a parent node containing the sum of their frequencies. Continue creating the tree from the bottom frequencies to the most frequent letters by inserting parent nodes and leaves using the same scheme. When reading the encoding from the root to a leaf, print a 0 for a left branch and a 1 for a right branch. Use the obtained encoding to compress the text file: the most frequent characters are going to be encoded with fewer binary digits. You need to save the encoding in a separate file in order to be able to decompress the original text.

## *Take-Home Idea*

The Greedy algorithm works by always picking the best local value in order to build a global solution. It provides an easy way to quickly build good but suboptimal solutions. A GPS screen can use Greedy to print a suboptimal route and do more intensive computing in the background while the user is being pleased by a responsive touch screen and a suboptimal solution.

# *Laboratory Assignment 11: Backtracking*

## *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with the backtracking technique.

## *Theoretical Summary*

The fundamental principle behind the backtracking strategy is that when a dead-end is detected it is possible to resume by going back to the previous choice. It fits quite natural for recursive implementations being a depth-first tree search of possible (partial) solutions. Each choice is a node in the tree. Everyone who ever got disoriented in an unfamiliar town or forest is familiar with the technique[23]: you retrace your path by backtracking when you make a wrong turn or reach a dead-end.

[23] nist.gov

The solution space for backtracking is a cartesian product $S = S_0 \times S_1 \times S_2 \times \cdots \times S_{n-1}$ of the form:

$$S = \begin{bmatrix} x_0^0 \\ x_1^0 \\ \vdots \\ x_{n-1}^0 \end{bmatrix} \times \begin{bmatrix} x_0^1 \\ x_1^1 \\ \vdots \\ x_{n-1}^1 \end{bmatrix} \times \cdots \times \begin{bmatrix} x_0^{n-1} \\ x_1^{n-1} \\ \vdots \\ x_{n-1}^{n-1} \end{bmatrix}$$

Not all $x \in S$ are valid solutions. Function bool isvalidsolution(x) returns true only if $x = [x_0, x_1, x_2, \ldots x_{n-1}]$ is a valid solution where $x_i \in S_i$ and $0 < i < n$. It is up to us to implement the bool isvalidsolution(x) function depending on the problem requirements.

So revisiting the recursive call tree, the pseudocode for backtracking is given by:

```
bool solve(pick x[i] from S[i]) {  /* starts with some initial guess */
    if x[i] is a leaf node {       /* if stuck or ready, depending on problem */
        if the leaf isvalidsolution(x[0], x[1], ... x[i])==true return true;
        else return false;
    } else {                       /* we can explore more partial solutions */
```

```
        for each child x of x[i] {
            if solve(x) succeeds, return true;
        }
      return false;
    }
}
```

## *Assignment Breakdown and Methodology*

1. Tab.4 revisits the TSP problem from the previous chapter.

| - | Timişoara | Oradea | Cluj-Napoca | Iaşi | Bucureşti |
|---|---|---|---|---|---|
| Timişoara | - | 174 | 315 | 634 | 544 |
| Oradea | 174 | - | 152 | - | 595 |
| Cluj-Napoca | 315 | 152 | - | 393 | - |
| Iaşi | 634 | 544 | 393 | - | - |
| Bucureşti | 544 | 595 | - | 388 | - |

Table 4: The adjacency matrix for town to town connectivity in Romania revisited

   Write a program which finds the shortest route starting from a given city and visiting all the other cities from Tab.4. The adjacency matrix is given in a text file with the order of the matrix written on the first line.

2. The chess Knight always moves in an L-shaped fashion in any direction. The 8 possible moves are formed by changing one of the x or y coordinates with 2 units and the other one with one unit in a 2-by-1 L shape on the board. A famous challenge is to move the Knight from a given position on the empty board and cover the battlefield by touching all the squares from the board only once. A solution starting with position 0 is given in Tab.5.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 59 | 38 | 33 | 30 | 17 | 8 | 63 |
| 37 | 34 | 31 | 60 | 9 | 62 | 29 | 16 |
| 58 | 1 | 36 | 39 | 32 | 27 | 18 | 7 |
| 35 | 48 | 41 | 26 | 61 | 10 | 15 | 28 |
| 42 | 57 | 2 | 49 | 40 | 23 | 6 | 19 |
| 47 | 50 | 45 | 54 | 25 | 20 | 11 | 14 |
| 56 | 43 | 52 | 3 | 22 | 13 | 24 | 5 |
| 51 | 46 | 55 | 44 | 53 | 4 | 21 | 12 |

Table 5: A solution to the Knight jump problem beginning with position square 0 at the top left white corner

   Write a program which solves the Knight jump problem starting with any given position on the chessboard. You may encode the 8 Knight possible moves using deltaX and deltaY values.

   (a) Use a recursive backtracking implementation.

   (b) Use a non-recursive backtracking implementation.

## *Questions*

1. What is the fundamental strategy behind backtracking?
2. What is the difference between recursive and non-recursive backtracking implementations?
3. Which one is faster? Explain.
4. Can you improve backtracking?
5. Is it necessary to erase previous steps when backtracking?
6. Can backtracking and greedy be blended together?
7. Write the text for a backtracking problem where backtracking takes too long.
8. Write the text for a backtracking problem. Make use of the solution space format $S$ described in the Theoretical Summary.
9. Rewrite the text for the previous backtracking problem without using the solution space format $S$. Explain.
10. Is it possible to prune the recursive call tree during the runtime? Explain.

## *Home Exercises*

1. A text file contains the map of a maze encoded in a 2D array such that walls are printed with "#" and spaces with ".". Given a starting point $(i, j)$ find the shortest path to exit the maze, if possible.
   (a) Print all possible solutions as a list of $(i, j)$ coordinates.
   (b) Print all possible solutions by marking the route with "x" in the maze.
2. The maze problem can be extended to 3D with N levels and places to go up and down one level. Solve the maze in a 3D box and print the result in $(i, j, k)$ format where $k$ is the floor level.
3. Place 8 queens on the chessboard such that they do not threat each other. Show all possible unique solutions.
4. Given an amount of money and a set of N coins and their values, print all possible ways to make the change.

### *Take-Home Idea*

Backtracking is a strategy for constructing solutions piece by piece. It is powerful when all possible solutions or a global optimum is required. Solutions are constructed by pruning the non-promising routes and by using the ability to resume from a previous track. Placing 8 queens on a chessboard is a good example.

# *Laboratory Assignment 12: Dynamic data structures*

### *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with dynamic data structures.

### *Theoretical Summary*

Dynamically allocated memory can be requested with the following standard library functions:

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

Dynamically allocated memory must be manually released using free(). Whenever malloc() or callloc () are called free() must immediately follow. Releasing the same memory pointer twice or even allocating memory in a wrong way can lead to security breaches and realloc() is no exception. So whenever memory is reserved, the free() function must be carefully prepared in advance.

Dynamically allocated memory in contrast to statically declared memory can make use of the entire RAM. The statically declared memory is prepared by the compiler before the runtime.

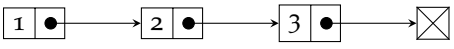



Figure 8: Simple linked list, NULL terminated (non-circular)

The simple linked list in Fig.8 has dynamically allocated nodes of the form:

```
struct lnode { /* simple linked list, may be circular */
        int key;
        struct lnode *next;
```

```
}

void main (void) {
        struct lnode *root;
}
```

where the last element has the pointer "next" set to NULL. Another option is to create a simple circular linked list by setting the final "next" pointer to the address of the first element, called the root or head of the list: current->next = root where "current" is the last element from the list with the key value 3.

In a double linked list the nodes also provide a pointer to the previous element allowing for bidirectional traversal of the list:

```
struct llnode { /* double linked list, may be circular */
        int key;
        struct llnode *previous, *next;
}

void main (void) {
        struct llnode *root;
}
```

Inserting and deleting elements in a linked list is pretty much straightforward provided that the pointers are corrected accordingly. One can search for an element in the list of nodes using:

```
struct lnode {
        int key;
        struct lnode *next;
}

void main (void) {
        struct lnode *root, *crt;

        crt = root;
        while (crt != NULL) {
                if (crt->key == searchedkey) { printf("Found!\n"); break; }
                else crt=crt->next;
        }

        free(...);
}
```

If the "next" and "previous" pointers from a double linked list are renamed with "left" and "right", one easily obtains the node structure for a binary search tree:

```
struct bstnode { /* binary search tree */
        int key;
        struct bstnode *left, *right;
}

void main (void) {
        struct bstnode *root;
}
```

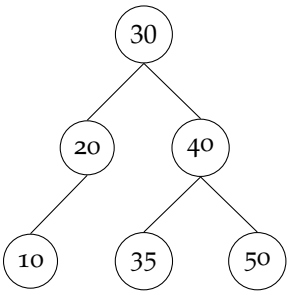



Figure 9: A simple binary search tree (BST) with 6 entries

Fig.9 shows a simple binary search tree with 6 elements having the node with key==30 as the root, being the first element inserted. Nodes with higher values are inserted to the right, and those with lower key values to the left of the root. One needs to search the right place for the next insertion using the "left" and "right" pointers until a leaf – a "NULL" is found. If the nodes are inserted with constantly increasing key values the tree degenerates into a linked list with the last "next" set to "NULL".

## *Assignment Breakdown and Methodology*

1. Write a function to construct (and return) a double linked list that is the reverse of a given original double linked list without changing the original list provided as parameter.

2. Write a function to split a simple linked list into two double linked lists at a specific point in the original list. The specific point is provided by the key value.

3. Write a function to remove the middle element in a circular simple linked list.

4. Write a function to add unique floating point values in a binary search tree. Use eps = 0.001 for comparing the values or read it from the command line.

## *Questions*

1. Is it easier to delete an element from a linked list, or from a 1D array?
2. What is a circular linked list?
3. Is it easier to search for an element into a 1D array, or into a linked list?
4. What is the most important advantage of a binary search tree over a linked list?
5. Can a binary search tree loose this advantage?
6. What happens if you forget to call free() when deleting a node from a dynamic data structure?
7. What is the disadvantage of a binary search tree as compared to a linked list?
8. What is the advantage of a simple linked list over a double linked list?
9. What is the advantage of a double linked list over a simple linked list?
10. Which one is better: logarithmic or linear performance?

## *Home Exercises*

1. Carefully study and experiment with the examples from the Theoretical Summary.
2. Write the text for a problem with dynamic data structures.
3. Solve the problem.
4. A circular double linked list can be used to dial a simple game.
5. Some kids decide to play a game. They sit in a circle, so that their names come in alphabetical order (considered clockwise). Then they choose a direction, either clockwise, either counter-clockwise. They also choose a number, N. If the direction is clockwise, they start from the kid with the first name in alphabetical order. Otherwise they start from the kid with the last name. They count the kids in the chosen direction, and they eliminate every N-th kid, until there is nobody left in the circle. Write a program that simulates this game, using doubly linked ordered circular lists.

The names of the kids are read from a text file, each name on a separate line. The number N and the direction are read from the keyboard. Print out to the screen how the game evolves (who is skipped and who is eliminated).

For example, supposing the file with the names contains:

```
Jacob
Emma
Michael
Isabella
Emily
```

The kids will be arranged in the following order: Emily Emma Isabella Jacob Michael.

If N is 3 and the direction is counter-clockwise, then the program can print out something like this:

```
Skipping Michael
Skipping Jacob
Eliminating Isabella.
Skipping Emma
Skipping Emily
Eliminating Michael.
Skipping Jacob
Skipping Emma
Eliminating Emily.
Skipping Jacob
Skipping Emma
Eliminating Jacob.
Skipping Emma
Skipping Emma
Eliminating Emma.
Game over.
```

## *Take-Home Idea*

Dynamic data structures are built during the runtime depending on user interaction. Sorted binary trees can be used to improve the access speed for data and double linked lists to allow for bidirectional searches. Variations are allowed, like forests or lists of lists. A ring network can be modeled using a circular linked list and message delivery probabilities.

# *Laboratory Assignment 13: Dynamic programming*

### *Scope*

The scope of this Laboratory Assignment is for the students to familiarize themselves with the dynamic programming technique.

### *Theoretical Summary*

Dynamic programming is a method to improve the performance of the algorithm when the solution contains overlapping partial solutions. Revisiting the Fibonacci recursive call tree:

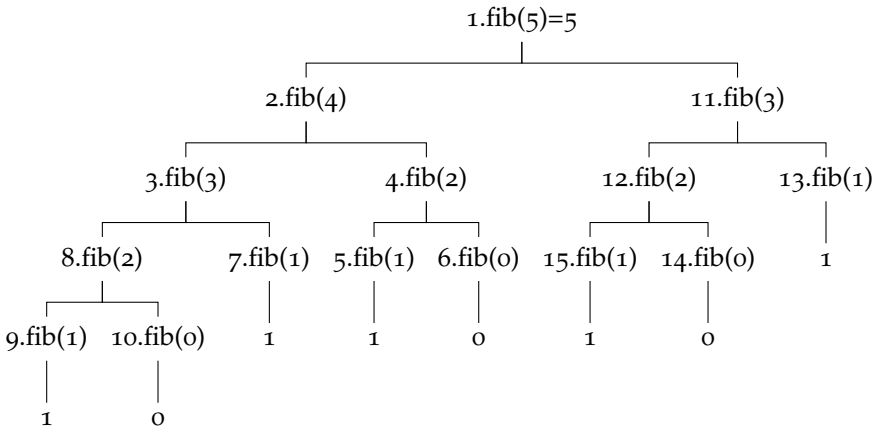



Figure 10: Recursion tree for the Fibonacci function

with the known formula:

$F(N) = 0, 1, 1, 2, 3, 5, 8, \ldots$

$$f(N) = \begin{cases} 0 & N = 0 \\ 1 & N = 1 \\ f(N-2) + f(N-1) & N \geq 2 \end{cases} \tag{3}$$

it is easy to spot that $f(N)$ is obtained from the partial solutions $f(N-1)$ and $f(N-2)$. More importantly, quite a lot of work is being done to calculate the already known values for the previous terms.

The question is why shouldn't we store those values in an array and reuse them? Provided that fib(4)=3 and fib(3)=2 are already known, fib(5) = 3 + 2 is obtained in an instant:

```
#include <stdio.h>

unsigned long int fib(unsigned long int n) {
        long int f[n+1];
        unsigned int i;

        f[0] = 0;
        f[1] = 1;

        for (i = 2; i<=n; i++) f[i] = -1;

        if (n > 2) {
                for (i = 2; i <= n; i++) {
                        if (f[i]<0) f[i] = f[i-1] + f[i-2];
                }
        }

        return (unsigned long int ) f[n];
}

int main (void) {
        printf("fib(10) = %lu\n", fib(10));
        printf("fib(3) = %lu\n", fib(3));
        printf("fib(5) = %lu\n", fib(5));
        printf("fib(11) = %lu\n", fib(11));

        return 0;
}
```

## *Assignment Breakdown and Methodology*

1. Carefully study and experiment with the example from the Theoretical Summary.

2. Rewrite the example from the Theoretical Summary such that the $f(N)$ precomputed values are stored in a global array and print a message on the screen whenever a value from the array is being reused.

3. Revisiting the coin make change problem, use dynamic programming to count in how many ways can a certain amount of money

be expressed in terms of a given set of coin values. The order of the coins is not important. Use a 2D array to store the partial solutions.

## *Questions*

1. What is the difference between divide and conquer and dynamic programming?
2. What is the difference between backtracking and dynamic programming?
3. What is the fundamental principle behind dynamic programming?
4. What are the prerequisites for applying dynamic programming?
5. Can dynamic programming be used to improve the calculus of $e^x$ and make it faster each time the program is called?

## *Home Exercises*

1. Write the text for a problem to be solved with dynamic programming (DP).
2. Solve the problem.
3. Given a rod of length n meters and a list of prices $p_i$, $i = 1, 2, \ldots n$ write an algorithm to find the maximum revenue obtainable by cutting up the rod and selling the pieces.

## *Take-Home Idea*

Dynamic programming is a strategy where partial solutions are recorded in order to speed-up the search for a new result by reusing the known states. Fibonacci terms can be computed by resuming the sequence starting with a previously recorded solution. Dynamic programming trades memory storage for processor cycles and can be easily combined with other techniques.

# *Appendix A: The Story of Mel Kaye free verse epic version*

A science or a community becomes sufficiently mature when grown up people start writing epics about the subject singing the deeds, the battles and the brave victories or attitudes of their heroes. The same pattern took place in the history of Europe and the Church[24]. The real Mel from the story has been recently identified as Mel Kaye, an engineer transferred to the Royal McBee payroll mentioned in a 1956 newspaper along with Bud Hazlett and Jack Behr. "The Story of Mel" is an archetypal piece of computer programming folklore. Its subject, Mel Kaye, is the canonical Real Programmer. The events happened in the days when the first complete compiler was built through an 18 years-person effort. A photo of the real Mel is available on the 1956 front page. It is now clear that the narrative was composed long after Mel was parted with Royal McBee, because it made such a lasting impression.

[24] see Dennis Ronald MacDonald on epics writing; also the Annals of Applied Statistics, Vol. 7, No. 4, p.2081.

This is the story of Mel Kaye, by Ed Nather, Professor Emeritus at the University of Texas[25] also known as "The Story of Mel". May 21, 1983. Modified free verse epic version.

[25] Professor Emeritus Edward Nather like Rod M. Jenkins, the discoverer of the star of the magi[26], was an astronomer. He pioneered the field of asteroseismology of white dwarfs.

[26] Journal of the British Astronomical Association, Vol. 114, No. 6, p.336.

```
Real Programmers write in FORTRAN.

Maybe they do now,
in this decadent era of
Lite beer, hand calculators, and "user-friendly" software
but back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums and vacuum tubes,
Real Programmers wrote in machine code.
Not FORTRAN.  Not RATFOR.  Not, even, assembly language.
Machine Code.
Raw, unadorned, inscrutable hexadecimal numbers.
Directly.
```

```
Lest a whole new generation of programmers
grow up in ignorance of this glorious past,
I feel duty-bound to describe,
as best I can through the generation gap,
how a Real Programmer wrote code.
I'll call him Mel,
because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp.,
a now-defunct subsidiary of the typewriter company.
The firm manufactured the LGP-30,
a small, cheap (by the standards of the day)
drum-memory computer,
and had just started to manufacture
the RPC-4000, a much-improved,
bigger, better, faster --- drum-memory computer.
Cores cost too much,
and weren't here to stay, anyway.
(That's why you haven't heard of the company,
or the computer.)

I had been hired to write a FORTRAN compiler
for this new marvel and Mel was my guide to its wonders.
Mel didn't approve of compilers.

"If a program can't rewrite its own code",
he asked, "what good is it?"

Mel had written,
in hexadecimal,
the most popular computer program the company owned.
It ran on the LGP-30
and played blackjack with potential customers
at computer shows.
Its effect was always dramatic.
The LGP-30 booth was packed at every show,
and the IBM salesmen stood around
talking to each other.
Whether or not this actually sold computers
was a question we never discussed.

Mel's job was to re-write
the blackjack program for the RPC-4000.
(Port?  What does that mean?)
```

The new computer had a one-plus-one
addressing scheme,
in which each machine instruction,
in addition to the operation code
and the address of the needed operand,
had a second address that indicated where, on the revolving drum,
the next instruction was located.

In modern parlance,
every single instruction was followed by a GO TO!
Put *that* in Pascal's pipe and smoke it.

Mel loved the RPC-4000
because he could optimize his code:
that is, locate instructions on the drum
so that just as one finished its job,
the next would be just arriving at the "read head"
and available for immediate execution.
There was a program to do that job,
an "optimizing assembler",
but Mel refused to use it.

"You never know where it's going to put things",
he explained, "so you'd have to use separate constants".

It was a long time before I understood that remark.
Since Mel knew the numerical value
of every operation code,
and assigned his own drum addresses,
every instruction he wrote could also be considered
a numerical constant.
He could pick up an earlier "add" instruction, say,
and multiply by it,
if it had the right numeric value.
His code was not easy for someone else to modify.

I compared Mel's hand-optimized programs
with the same code massaged by the optimizing assembler program,
and Mel's always ran faster.
That was because the "top-down" method of program design
hadn't been invented yet,
and Mel wouldn't have used it anyway.
He wrote the innermost parts of his program loops first,
so they would get first choice

```
of the optimum address locations on the drum.
The optimizing assembler wasn't smart enough to do it that way.

Mel never wrote time-delay loops, either,
even when the balky Flexowriter
required a delay between output characters to work right.
He just located instructions on the drum
so each successive one was just *past* the read head
when it was needed;
the drum had to execute another complete revolution
to find the next instruction.
He coined an unforgettable term for this procedure.
Although "optimum" is an absolute term,
like "unique", it became common verbal practice
to make it relative:
"not quite optimum" or "less optimum"
or "not very optimum".
Mel called the maximum time-delay locations
the "most pessimum".

After he finished the blackjack program
and got it to run
("Even the initializer is optimized",
he said proudly),
he got a Change Request from the sales department.
The program used an elegant (optimized)
random number generator
to shuffle the "cards" and deal from the "deck",
and some of the salesmen felt it was too fair,
since sometimes the customers lost.
They wanted Mel to modify the program
so, at the setting of a sense switch on the console,
they could change the odds and let the customer win.

Mel balked.
He felt this was patently dishonest,
which it was,
and that it impinged on his personal integrity as a programmer,
which it did,
so he refused to do it.
The Head Salesman talked to Mel,
as did the Big Boss and, at the boss's urging,
a few Fellow Programmers.
Mel finally gave in and wrote the code,
```

but he got the test backwards,
and, when the sense switch was turned on,
the program would cheat, winning every time.
Mel was delighted with this,
claiming his subconscious was uncontrollably ethical,
and adamantly refused to fix it.

After Mel had left the company for greener pa$ture$,
the Big Boss asked me to look at the code
and see if I could find the test and reverse it.
Somewhat reluctantly, I agreed to look.
Tracking Mel's code was a real adventure.

I have often felt that programming is an art form,
whose real value can only be appreciated
by another versed in the same arcane art;
there are lovely gems and brilliant coups
hidden from human view and admiration, sometimes forever,
by the very nature of the process.
You can learn a lot about an individual
just by reading through his code,
even in hexadecimal.
Mel was, I think, an unsung genius.

Perhaps my greatest shock came
when I found an innocent loop that had no test in it.
No test.  *None*.
Common sense said it had to be a closed loop,
where the program would circle, forever, endlessly.
Program control passed right through it, however,
and safely out the other side.
It took me two weeks to figure it out.

The RPC-4000 computer had a really modern facility
called an index register.
It allowed the programmer to write a program loop
that used an indexed instruction inside;
each time through,
the number in the index register
was added to the address of that instruction,
so it would refer
to the next datum in a series.
He had only to increment the index register
each time through.

```
Mel never used it.

Instead, he would pull the instruction into a machine register,
add one to its address,
and store it back.
He would then execute the modified instruction
right from the register.
The loop was written so this additional execution time
was taken into account ---
just as this instruction finished,
the next one was right under the drum's read head,
ready to go.
But the loop had no test in it.

The vital clue came when I noticed
the index register bit,
the bit that lay between the address
and the operation code in the instruction word,
was turned on ---
yet Mel never used the index register,
leaving it zero all the time.
When the light went on it nearly blinded me.

He had located the data he was working on
near the top of memory ---
the largest locations the instructions could address ---
so, after the last datum was handled,
incrementing the instruction address
would make it overflow.
The carry would add one to the
operation code, changing it to the next one in the instruction set:
a jump instruction.
Sure enough, the next program instruction was
in address location zero,
and the program went happily on its way.

I haven't kept in touch with Mel,
so I don't know if he ever gave in to the flood of
change that has washed over programming techniques
since those long-gone days.
I like to think he didn't.
In any event,
I was impressed enough that I quit looking for the
offending test,
```

telling the Big Boss I couldn't find it.
He didn't seem surprised.

When I left the company,
the blackjack program would still cheat
if you turned on the right sense switch,
and I think that's how it should be.
I didn't feel comfortable
hacking up the code of a Real Programmer.

# *Appendix B: The Little Man Computer*

The "Little Man Computer" (LMC) is a computer with a small dwarf inside[27] invented by the MIT for teaching purposes. The dwarf has some tools at its disposal:

- 100 mailboxes indexed from 00 to 99
- 1 inbox for input
- 1 outbox for output
- a counter
- a panel with 3 digit instructions (Tab. 6)
- an agenda

An LMC instruction cycle is always performed according to the following steps:

1. read the counter which holds a mailbox number from 00 to 99
2. read the 3 digit OpCode from the mailbox pointed to by the counter
3. increment the counter with +1 so that it will point to the next mailbox where to fetch the instruction from
4. follow the instruction as specified by the panel in (Tab. 6)

In order to start the computer, the program has to be loaded starting with address 00. The program contains at most 100 instructions.

The following sample program computes the value of the $N^{th}$ Fibonacci term. N is read from the inbox, the value of $F_N$ is given at the outbox. Tab.7 shows the code listing.

$$f(N) = \begin{cases} 0 & N = 0 \\ 1 & N = 1 \\ f(N-2) + f(N-1) & N \geq 2 \end{cases} \tag{4}$$

$F(N) = 0, 1, 1, 2, 3, 5, 8, \ldots$

[27] The first programmable robot was built by Heron of Alexandria in order to play theater. It was programmed using rope knots. He also invented the first coin-based vending machine to prevent the theft of holy water, along with many automatic temple wonders for pilgrims: self-opening doors which opened when a fire was lit to heat up the temple chamber, an automatic drinking bird which consumed water at the feet of Pan when the liquid was poured from the mouth of a small erotes face, a steam engine, a mysterious self-trimming oil lamp and several smart candle devices, a fountain which trickles by the action of the Sun's rays and a vessel in which water and air ascend and descend alternately resembling a karstic estavelle. He is a Founder and Director and the last recorded member of the first Polytechnic School and he is well known for the formula of the area of a triangle from its side lengths.

The Musaeum of Alexandria where Heron had lectured during the reign of the $2^{nd}$ Roman Emperor, Tiberius Caesar and his close associate Publius Sulpicius Quirinius included the ancient Library of Alexandria founded by the Ptolemaic dynasty of Egypt. It is here where the most important books and manuscripts of the world have been translated not only from countries nearby Egypt but even procured by the missionaries of the Indian King Ashoka as recorded by the Greek-Aramaic $13^{th}$ rock edict.

| 3 digit OpCode | Operation | Mnemonics | Description |
|---|---|---|---|
| 1xy | LOAD | LDA | copy the value from mailbox (xy) into the agenda |
| 2xy | STORE | STO | copy the value from the agenda into mailbox (xy) |
| 3xy | ADD | ADD | add the value from mailbox (xy) to the value from the agenda |
| 4xy | SUBSTRACT | SUB | substract the value from mailbox (xy) from the value from the agenda |
| 500 | INPUT | IN | copy the value from the inbox into the agenda |
| 600 | OUTPUT | OUT | copy the value from the agenda on a note and deposit it into the outbox |
| 700 | HALT | HLT | stop until someone presses the RESET button |
| 800 | SKIP_IF_NEG | SKN | if the number from the agenda has a "-" sign jump over an instruction by incrementing the counter |
| 801 | SKIP_IF_ZERO | SKZ | if the number from the agenda is ZERO jump over an instruction by incrementing the counter |
| 802 | SKIP_IF_POS | SKP | if the number from the agenda is positive jump over an instruction by incrementing the counter |
| 803 | SKIP_NON_ZERO | SKNZ | if the number from the agenda is NOT ZERO jump over an instruction by incrementing the counter |
| 9xy | JUMP | JMP | set the counter to the value from mailbox (xy); the next instruction will be read from mailbox (xy) |
| 0xy | SUBRTN_CALL | CALL | see JUMP |
| ??? | DATABASE | DAT | write the digits "???" into the unused mailboxes after HALT |

Table 6: The "LMC" instructions panel

The flowchart for this program is given by Fig.11. The $N^{th}$ term from the Fibonacci sequence is given by the formula in (4). Notice how important it is to declare the variables on the last 6 lines, using the OpCode for DATABASE. For instance, variable "FIRST" which holds the value of $F(n-2)$ is associated with the mailbox address 37 which initially holds the decimal value 000.

**Optional Hack 1**. Write a small LMC program which prints the first N odd natural numbers. N is given into the inbox. Use comments to keep track of what you are doing.

**Optional Hack 2**. Write a LMC program which computes the sum of the first N natural numbers.

**Optional Hack 3**. Write a LMC program to implement the formula given by $\frac{N(N+1)}{2}$ knowing that N is even.

**Optional Hack 4**. A punched film roll contains the following LMC instructions encoded using 10 bits:

0111110100100101100001100101111100100010111000 1..

| Adresă | Program.LMC |
|---|---|
| 00 | IN |
| 01 | SKZ |
| 02 | JMP 05 ; or relative addressing +03 |
| 03 | OUT ; display "0" |
| 04 | HLT |
| 05 | SUB ONE |
| 06 | SKZ |
| 07 | JMP 11 ; absolute address 11 |
| 08 | ADD ONE |
| 09 | OUT ; display "1" |
| 10 | HLT |
| 11 | SUB ONE |
| 12 | SKZ |
| 13 | JMP 17 |
| 14 | ADD ONE |
| 15 | OUT ; display "1" |
| 16 | HLT |
| 17 | ADD TWO |
| 18 | SUB ONE |
| 19 | STO N |
| 20 | SKNZ |
| 21 | JMP 32 |
| 22 | LDA AUX |
| 23 | ADD FIRST |
| 24 | ADD SECOND |
| 25 | STO AUX |
| 26 | LDA SECOND |
| 27 | STO FIRST |
| 28 | LDA AUX |
| 29 | STO SECOND |
| 30 | LDA N |
| 31 | JMP 18 ; relative to mailbox address 31 would be JMP -13 |
| 32 | LDA AUX |
| 33 | OUT ; display f(N) |
| 34 | HLT |
| 35 | DAT ONE 001 |
| 36 | DAT TWO 002 |
| 37 | DAT FIRST 000 |
| 38 | DAT SECOND 001 |
| 39 | DAT N 000 |
| 40 | DAT AUX 000 |

Table 7: Fibonacci Program

Example for F(3), N=3
SUB ONE
SUB ONE
ADD TWO
STO N
N=N-1=2
AUX=FIRST+SECOND=1
FIRST=SECOND=1
SECOND=AUX=1
N=N-1=1
AUX=FIRST+SECOND=2
FIRST=SECOND=1
SECOND=AUX=2
N=N-1=0
OUT AUX=2

..01011100001011010111100000000001.

Reverse engineer the program by disassembling it and by renaming the unknown variables. Explain how it works and what it does.

**Optional Hack 5**. A rectangular punched cardboard contains LMC instructions encoded in base 2 on 12 bits, one instruction for every row of holes. A position with a hole that allows the light to reach an electronic photoreceptor is a "1". An opaque position is a "0". In order to compress the 12 bit numbers these are replaced with hexadecimal values. The digits for base 16 (hexadecimal) have 16 possible values numbered A–F starting with digit 10. Consequently each LMC instruction is compressed on 3 hexadecimal digits because each hexadecimal digit covers 4 bits ($2^4 = 16$).

Try to reverse engineer the following program by following it carefully step by step. Explain what it does and how it works: 1F4 258 38D 199 323 38C 386 2BD 002 07C 147 0D6 147 0D8 067 1A9 0CB 07C 145 0E0 1AA 321 38D 387 003 001 009 064.

Thus spake the Master Programmer: "Time for you to leave." , *The Tao Of Programming*, Book 9.

1. Write a program for converting the string of bits from **Optional**

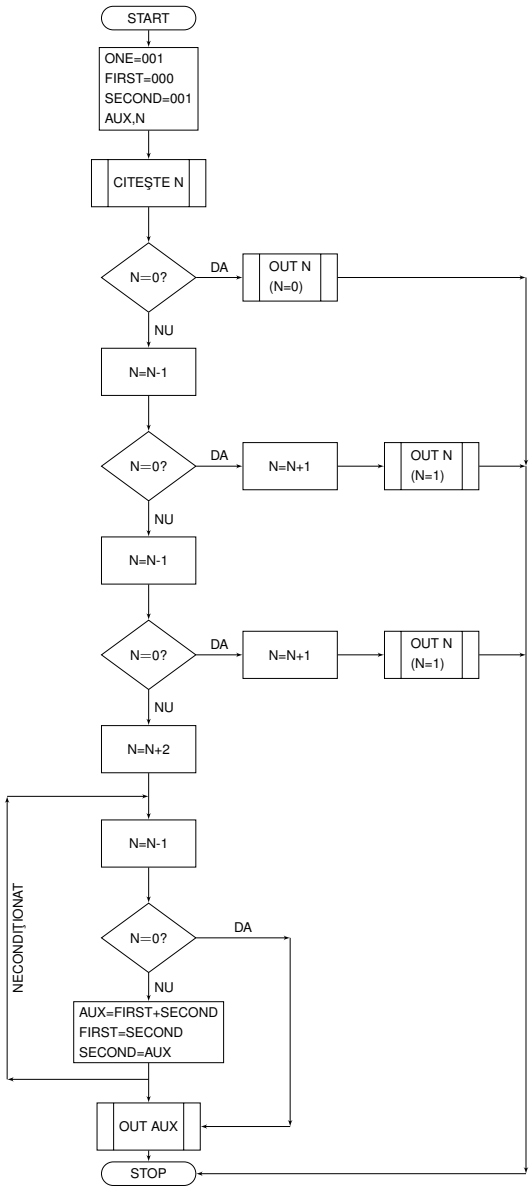



Figure 11: The flowchart for the Fibonacci Program in Tab.2

Example for F(N) with N=2:
N=N-1
N=N-1
N=N+1
OUT N=1

**Hack 4** into a list of instructions and their explanation using the format of Tab.7. Use proper composite variables to encode Tab.7 as elegantly as possible.

2. Do the same for **Optional Hack 5**.

# *Appendix C: C Language Summary*

| | |
|---|---|
| auto | Defines a local variable as having a local lifetime. |
| break | Passes control out of the compound statement. |
| case | Branches control. |
| char | Basic signed integer type. |
| const | Makes variable value or pointer parameter unmodifiable. |
| continue | Passes control to the begining of the loop. |
| default | Branches control. |
| do | Do-while loop. |
| double | Double precision floating point data type. |
| else | Conditional branching statement. |
| enum | Defines a set of constants of type int. |
| extern | Indicates that an identifier is defined elsewhere. |
| float | Single precision floating point data type. |
| for | For loop. |
| goto | Unconditionally transfer control. |
| if | Conditional branching statement. |
| inline | Inline expansion of function calls for speed. |
| int | Basic single precision signed integer data type. |
| long | Type modifier for integers. |
| register | Tells the compiler to store the variable being declared in a CPU register. |
| restrict | Restrict pointers so that they can not be aliased. |
| return | Exits the function. |
| short | Type modifier for integers. |
| signed | Type modifier for integers. |
| sizeof | Returns the size of the expression or type. |
| static | Preserves variable value to survive after its scope ends. |
| struct | Groups variables into a single composite record. |
| switch | Branches control. |
| typedef | Creates a new type. |
| union | Groups variables which share the same storage space. |
| unsigned | Type modifier for integers. |
| void | Empty data type. |
| volatile | Indicates that a variable can be changed by a background routine. |
| while | While loop. |

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | ++ −− | Suffix/postfix increment and decrement | Left-to-right |
| | () | Function call | |
| | [] | Array subscripting | |
| | . | Structure and union member access | |
| | −> | Structure and union member access through pointer | |
| | (type){ list } | Compound literal | |
| 2 | ++ −− | Prefix increment and decrement | Right-to-left |
| | + − | Unary plus and minus | |
| | ! ~ | Logical NOT and bitwise NOT | |
| | (type) | Type cast | |
| | * | Indirection (dereference) | |
| | & | Address-of | |
| | **sizeof** | Size-of | |
| | _Alignof | Alignment requirement | |
| 3 | * / % | Multiplication, division, and remainder | Left-to-right |
| 4 | + − | Addition and subtraction | |
| 5 | << >> | Bitwise left shift and right shift | |
| 6 | < <= | For relational operators $<$ and $\leq$ respectively | |
| | > >= | For relational operators $>$ and $\geq$ respectively | |
| 7 | == != | For relational $=$ and $\neq$ respectively | |
| 8 | & | Bitwise AND | |
| 9 | ^ | Bitwise XOR (exclusive or) | |
| 10 | \| | Bitwise OR (inclusive or) | |
| 11 | && | Logical AND | |
| 12 | \|\| | Logical OR | |
| 13 | ?: | Ternary conditional | Right-to-Left |
| 14 | = | Simple assignment | |
| | += −= | Assignment by sum and difference | |
| | *= /= %= | Assignment by product, quotient, and remainder | |
| | <<= >>= | Assignment by bitwise left shift and right shift | |
| | &= \^= = \| | Assignment by bitwise AND, XOR, and OR | |
| 15 | , | Comma | Left-to-right |

| | |
|---|---|
| <assert.h> | Conditionally compiled macro that compares its argument to zero |
| <complex.h> | Complex number arithmetic |
| <ctype.h> | Functions to determine the type contained in character data |
| <errno.h> | Macros reporting error conditions |
| <fenv.h> | Floating-point environment |
| <float.h> | Limits of float types |
| <inttypes.h> | Format conversion of integer types |
| <iso646.h> | Alternative operator spellings |
| <limits.h> | Sizes of basic types |
| <locale.h> | Localization utilities |
| <math.h> | Common mathematics functions |
| <setjmp.h> | Nonlocal jumps |
| <signal.h> | Signal handling |
| <stdalign.h> | alignas and alignof convenience macros |
| <stdarg.h> | Variable arguments |
| <stdatomic.h> | Atomic types |
| <stdbool.h> | Boolean type |
| <stddef.h> | Common macro definitions |
| <stdint.h> | Fixed-width integer types |
| <stdio.h> | Input/output |
| <stdlib.h> | Memory management, program utilities, string conversions, random numbers |
| <stdnoreturn.h> | noreturn convenience macros |
| <string.h> | String handling |
| <tgmath.h> | Type-generic math (macros wrapping math.h and complex.h) |
| <threads.h> | Thread library |
| <time.h> | Time/date utilities |
| <uchar.h> | UTF-16 and UTF-32 character utilities |
| <wchar.h> | Extended multibyte and wide character utilities |
| <wctype.h> | Wide character classification and mapping utilities |

```
root@parabolaiso / # man stdio.h

stdio.h(0P)               POSIX Programmer's Manual              stdio.h(0P)

PROLOG
This  manual  page is part of the POSIX Programmer's Manual.  The Linux
implementation of this interface may differ (consult the  corresponding
Linux  manual page for details of Linux behavior), or the interface may...
.
.
.
```

# *Appendix D: Some GNU/Linux commands*

You can tell everything about a woman by the way she knows how to receive flowers: like the light breath of weather, she might be able to gently isolate and deepen their joy with the heartbeats of a butterfly – a state of flow which requires mental clarity and mastering.

| | |
|---|---|
| ls | directory listing |
| ls -al | formatted listing with hidden files |
| cd dir | change directory to dir |
| cd | change to home |
| pwd | show current directory |
| mkdir dir | create a directory dir |
| rm file | delete file |
| rm -r dir | delete directory |
| rm -f file | force remove |
| rm -rf dir | force remove directory dir |
| rmdir dir | remove directory |
| cp file1 file2 | copy file1 to file2 |
| cp -r dir1 dir2 | copy dir1 to dir2 |
| mv file1 file2 | rename or move file1 to file2 |
| ln -s file link | create symbolic link to file |
| touch file | create or update file |
| cat > file | places standard input into file |
| more file | output the contents of file |
| less file | output the contents of file |
| head file | output the first 10 lines of file |
| tail file | output the last 10 lines of file |
| tail -f file | continously output the contents of file |
| uptime | show current uptime |
| whoami | show current user |
| man command | show manual page for command |
| nano file | edit file with the nano editor |
| pico file | edit gile with the pico editor |
| rename .htm .html *.htm | fix html file extension |
| man bash | general commands manual |
| man csh | general commands manual for C shell |
| cint | command line C interpreter |
| gdb file | debug file with the GNU Debugger |
| hexdump -C file \| less | display binary file and pipe output to less command |

| | |
|---|---|
| sort file \| more | sort file and pipe output to more command |
| uniq file | display unique lines in file |
| wc -l file | count the number of lines from file |
| find . -name *".c" | find all .c files from current subdirectories |
| ls -l \| grep ^d | display all subdirectories |
| gcc -Wall -O3 -ggdb file.c -o file.x | compile file.x with debug flags for gdb command |
| gcc -Wall -O3 -fopenmp file.c | compile a.out file with **#pragma** omp parallel compiler hints |
| gcc -Wall -O3 -fopenacc file.c | compile a.out file with **#pragma** acc parallel compiler hints |
| clang -Weverything file.c -o file.x | compile file.x with full diagnostics |
| truncate -s 0 file | truncate file to 0 bytes |

```
root@parabolaiso / # man rename
RENAME(1)                       User Commands                       RENAME(1)

NAME
rename - rename files

SYNOPSIS
rename [options] expression replacement file...

DESCRIPTION
rename  will  rename  the specified files by replacing the first occur-
rence of expression in their name by replacement...
.
.
.
```

# *Appendix E: Ergonomic computer workstation usage*

The programmer is not a device for turning caffeine, sugar and junk food into software. There a plenty of standards for almost anything out there and there are many standards for ergonomic computer workstation usage. However, the following checklist should serve as a guide for you to understand how to protect your health and productivity when working at a computer workstation[28]:

1. use a good chair with a dynamic chair back and sit in it
2. top of monitor casing 2-3" (5-8 cm) above eye level
3. no glare on screen, use an optical glass anti-glare filter where needed
4. sit at arms length from monitor
5. feet on floor or stable footrest
6. use a document holder, preferably in-line with the computer screen
7. wrists flat and straight in relation to forearms to use keyboard/-mouse/input device
8. arms and elbows relaxed close to body
9. center monitor and keyboard in front of you
10. use a negative tilt keyboard tray with an upper mouse platform or downward tiltable platform adjacent to keyboard
11. use a stable work surface and stable (no bounce) keyboard tray
12. take frequent short breaks (microbreaks)

Use a **negative tilting for the keyboard**. A custom or third-party keyboard tray can be attached beneath your desk. This will help your hands slightly bend down towards your feet. Use a mechanical keyboard you can sense, without a numerical pad. A sculpted or ergonomic keyboard will not fit the negative tilting and will disorient you when changing workstations.

[28] Cornell University, Professor Alan Hedge and his team

Use at least one **vertical display** for text and source code editing and for reading or simply rotate a regular display with 90 degress. Make sure it is non-glare and does not flicker.

Pick a robust and **ajdustable chair** which allows your skin to breath and avoid cubicles.

Use **powerful reflected light** for correct ambient lightning conditions and natural light when possible.

When you write software your brain will steal glucose from all the other organs dampening your energy and if you do not act accordingly, *it will break your health*. Make sure you have a thorough understanding of the potential psychological and physiological diseases associated with prolonged computer usage and desk sitting[29].

Pornography *does* hardwire your brain and *will destroy* your life. It thrives by consuming the souls, lives, identities, personality, deepest

**Alternate tasks** after 30 minutes of continuous usage and make short breaks. Stand up for printing and calls.

[29] Also, make sure you really grasp the full meaning of professional *burnout*, *brownout* and *karoshi*.

pain and misery of the actors who produce it. There is nothing like classic family and real life friends and real life communities so virtual communities can only serve as artificial extensions to the real component.

The Romanian Chapter of the IEEE Society on the Social Implications of Technology (SSIT)[30] was started in February 2013 in Timişoara at the Polytechnic University and had its first workshop in the $5^{th}$ of June 2013 having the author functioning as interim-chair and chapter organizer. Students are **always welcome** to join our meetings and discuss their problems, and also eventually find out about the latest implications of technology in their society.

[30] A sample topic of SSIT is *agnotology*: the study of willful acts to spread confusion and deceit, the study of culturally induced ignorance or doubt, in particular to sell a product or win favor. Also see http://ieeessit.org/.

# *Bibliography*


Alin Anton, Vladimir Creţu, Albert Ruprecht, and Sebastian Muntean. Traffic replay compression (TRC): A highly efficient method for handling parallel numerical simulation data. *PROCEEDINGS OF THE ROMANIAN ACADEMY – Series A*, 14(4):385–392, 2013. ISSN 1454-9069.

Anonymous author. History of computer developments in Romania. *Annals of the History of Computing, IEEE*, 21(3):58–60, Jul 1999. ISSN 1058-6180.

Alan R. Feuer. *The C Puzzle Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1985. ISBN 0131099264.

Paul Graham. *Hackers and painters - big ideas from the computer age.* O'Reilly, 2004. ISBN 978-0-596-00662-4.

Ciocârlie Horia. *Utilizarea şi programarea calculatoarelor, ediţia a II-a*. Eurostampa, Timişoara, 2012. ISBN 978-606-569-484-2.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. ISBN 0131103709.

B. Kjell. Computer science education and the global outsourcing of software production. *Technology and Society Magazine, IEEE Society on Social Implications of Technology*, 24(3):5–53, Fall 2005. ISSN 0278-0097.

Steve Oualline. *C Elements of Style: The Programmer's Style Manual for Elegant C and C++ Programs*. M & T Books, 1992. ISBN 1558512918.

Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. ISBN 0596001088.

Trevis Rothwell and James Youngman. *The GNU C Reference Manual*. The GNU Project, 2015.

John Walker. *The Hacker's Diet: How to lose weight and hair through stress and poor nutrition*. Fourmilab, 4th edition, 2005.