Tutorials for High Performance Scientific Computing

Victor Eijkhout

2017

Preface

The field of high performance scientific computing requires, in addition to a broad of scientific knowledge and 'coputing folkore', a number of practical skills. Call it the 'carpentry' aspect of the craft of scientific computing.

As a companion to the book 'Introduction to High Performance Scientific Computing', which covers background knowledge, here is then a set of tutorials on those practical skills that are important to becoming a successful high performance practitioner.

The tutorials should be done while sitting at a computer. Given the practice of scientific computing, they have a clear Unix bias.

Public draft This book is open for comments. What is missing or incomplete or unclear? Is material presented in the wrong sequence? Kindly mail me with any comments you may have.

You may have found this book in any of a number of places; the authoritative download location is `http://www.tacc.utexas.edu/~eijkhout/istc/istc.html`. That page also links to `lulu.com` where you can get a nicely printed copy.

Victor Eijkhout `eijkhout@tacc.utexas.edu`
Research Scientist
Texas Advanced Computing Center
The University of Texas at Austin

Contents

In the theory part of this book you learned mathematical models can be translated to algorithms that can be realized efficiently on modern hardware. You learned how data structures and coding decisions influence the performance of your code. In other words, you should now have all the tools to write programs that solve scientific problems.

This would be all you would need to know, if there was any guarantee that a correctly derived algorithm and well designed data structure could immediately be turned into a correct program. Unfortunately, there is more to programming than that. This collection of tutorials will give you the tools to be an effective scientific programmer.

The vast majority of scientific programming is done on the Unix platform so we start out with a tutorial on Unix in chapter 1, followed by an explanation of the how your code is handled by compilers and linkers and such in chapter 3.

Next you will learn about some tools that will increase your productivity and effectiveness:

- The *Make* utility is used for managing the building of projects; chapter 4.
- Source control systems store your code in such a way that you can undo changes, or maintain multiple versions; in chapter 5 you will see the *subversion* software.
- Storing and exchanging scientific data becomes an important matter once your program starts to produce results; in chapter 6 you will learn the use of *HDF5*.
- A lot of functionality that you need has been coded already by other people; in chapter **??** you will learn about some of the scientific libraries that are available to you.
- Visual output of program data is important, but too wide a topic to discuss here in great detail; chapter 7 teaches you the basics of the *gnuplot* package, which is suitable for simple data plotting.

We also consider the activity of program development itself: chapter 8 considers how to code to prevent errors, and chapter 9 teaches you to debug code with *gdb*. Chapter 11 discusses measuring the performance of code. Chapter 12 contains some information on how to write a program that uses more than one programming language.

Finally, chapter 13 teaches you about the LaTeX document system, so that you can report on your work in beautifully typeset articles.

Many of the tutorials are very hands-on. Do them while sitting at a computer!

Chapter 1

Unix intro

Unix is an *Operating System (OS)*, that is, a layer of software between the user or a user program and the hardware. It takes care of files and screen output, and it makes sure that many processes can exist side by side on one system. However, it is not immediately visible to the user. Most of the time that you use Unix, you are typing commands which are executed by an interpreter called the *shell*. The shell makes the actual OS calls. There are a few possible Unix shells available, but in this tutorial we will assume that you are using the *sh* or *bash* shell, although many commands are common to the various shells in existence.

This short tutorial will get you going; if you want to learn more about Unix and shell scripting, see for instance `http://www.tldp.org/guides.html`. Most of this tutorial will work on any Unix-like platform, including *Cygwin* on Windows. However, there is not just one Unix:

- Traditionally there are a few major flavours of Unix: ATT and BSD. Apple has Darwin which is close to BSD; IBM and HP have their own versions of Unix, and Linux is yet another variant. The differences between these are deep down and if you are taking this tutorial you probably won't see them for quite a while.
- Within Linux there are various *Linux distributions* such as *Red Hat* or *Ubuntu*. These mainly differ in the organization of system files and again you probably need not worry about them.
- As mentioned just now, there are different shells, and they do differ considerably. Here you will learn the bash shell, which for a variety of reasons is to be preferred over the *csh* or *tcsh* shell. Other shells are the *ksh* and *zsh*.

For further reading:

- 'The Linux Command Line' by William Shotts `gd.tuwien.ac.at/linuxcommand.org/` discusses life on the command line plus shell scripting.
- 'Bash Guide for Beginners' by Machtelt Garrels `tille.garrels.be/training/bash/` focuses on shell scripting in great detail.

1.1 Files and such

Purpose. In this section you will learn about the Unix file system, which consists of *directories* that store *files*. You will learn about *executable* files and commands for displaying data files.

1.1.1 Looking at files

Purpose. In this section you will learn commands for displaying file contents.

The `ls` command gives you a listing of files that are in your present location.

Exercise. Type `ls`. Does anything show up?

Expected outcome. If there are files in your directory, they will be listed; if there are none, no output will be given. This is standard Unix behaviour: no output does not mean that something went wrong, it only means that there is nothing to report.

The `cat` command is often used to display files, but it can also be used to create some simple content.

Exercise. Type `cat > newfilename` (where you can pick any filename) and type some text. Conclude with `Control-d` on a line by itself[1]. Now use `cat` to view the contents of that file: `cat newfilename`.

Expected outcome. In the first use of `cat`, text was concatenated from the terminal to a file; in the second the file was cat'ed to the terminal output. You should see on your screen precisely what you typed into the file.

Caveats. Be sure to type `Control-d` as the first thing on the last line of input. If you really get stuck, `Control-c` will usually get you out. Try this: start creating a file with `cat > filename` and hit `Control-c` in the middle of a line. What are the contents of your file?

Remark 1 *Instead of* `Control-d` *you will often see the notation* `^D`*. The capital letter is for historic reasons: you use the control key and the lowercase letter.*

Above you used `ls` to get a directory listing. You can also use the `ls` command on specific files:

Exercise. Do `ls newfilename` with the file that you created above; also do `ls nosuchfile` with a file name that does not exist.

Expected outcome. For an existing file you get the file name on your screen; for a non-existing file you get an error message.

The `ls` command can give you all sorts of information.

Exercise. Read the man page of the `ls` command: `man ls`. Find out the size and the time/date date of the last change to some files, for instance the file you just created.

Expected outcome. Did you find the `ls -s` and `ls -l` options? The first one lists the size of each file, usually in kilobytes, the other gives all sorts of information about a file, including things you will learn about later.

Caveats. The `man` command puts you in a mode where you can view long text documents. This viewer is common on Unix systems (it is available as the `more` or `less` system command), so memorize the following ways of navigating: Use the space bar to go forward and the `u` key to go back up. Use `g` to go to the beginning fo the text, and `G` for the end. Use `q` to exit the viewer. If you really get stuck, `Control-c` will get you out.

1. Press the `Control` and hold it while you press the `d` key.

Remark 2 *There are several dates associated with a file, corresponding to changes in content, changes in permissions, and access of any sort. The* `stat` *command gives all of them.*

Remark 3 *If you already know what command you're looking for, you can use man to get online information about it. If you forget the name of a command,* `man -k keyword` *can help you find it.*

The *touch* command creates an empty file, or updates the timestamp of a file if it already exists. Use `ls -l` to confirm this behaviour.

Three more useful commands for files are: *cp* for copying, *mv* (short for 'move') for renaming, and *rm* ('remove') for deleting. Experiment with them.

There are more commands for displaying a file, parts of a file, or information about a file.

Exercise. Do `ls /usr/share/words` or `ls /usr/share/dict/words` to confirm that a file with words exists on your system. Now experiment with the commands `head`, `tail`, `more`, and `wc` using that file.

> *Expected outcome.* *head* displays the first couple of lines of a file, *tail* the last, and *more* uses the same viewer that is used for man pages. Read the man pages for these commands and experiment with increasing and decreasing the amount of output. The *wc* ('word count') command reports the number of words, characters, and lines in a file.

Another useful command is *which*: it tells you what type of file you are dealing with. See what it tells you about one of the text files you just created.

1.1.2 Directories

> **Purpose.** Here you will learn about the Unix directory tree, how to manipulate it and how to move around in it.

A unix file system is a tree of directories, where a directory is a container for files or more directories. We will display directories as follows:

```
/ ..... The root of the directory tree
├─ bin ..... Binary programs
└─ home ..... Location of users directories
```

The root of the Unix directory tree is indicated with a slash. Do `ls /` to see what the files and directories there are in the root. Note that the root is not the location where you start when you reboot your personal machine, or when you log in to a server.

Exercise. The command to find out your current working directory is *pwd*. Your home directory is your working directory immediately when you log in. Find out your home directory.

> *Expected outcome.* You will typically see something like `/home/yourname` or `/Users/yourname`. This is system dependent.

Do `ls` to see the contents of the working directory. In the displays in this section, directory names will be followed by a slash: `dir/` but this character is not part of their name. You can get this output by using `ls -F`, and you can tell your shell to use this output consistently by stating `alias ls=ls -F` at the start of your session. Example:

```
/home/you/
├── adirectory/
└── afile
```

The command for making a new directory is *`mkdir`*.

Exercise. Make a new directory with *`mkdir`* `newdir` and view the current directory with `ls`

Expected outcome. You should see this structure:

```
/home/you/
└── newdir/ ..... the new directory
```

The command for going into another directory, that is, making it your working directory, is *`cd`* ('change directory'). It can be used in the following ways:

- `cd` Without any arguments, `cd` takes you to your home directory.
- `cd <absolute path>` An absolute path starts at the root of the directory tree, that is, starts with `/`. The `cd` command takes you to that location.
- `cd <relative path>` A relative path is one that does not start at the root. This form of the `cd` command takes you to `<yourcurrentdir>/<relative path>`.

Exercise. Do `cd newdir` and find out where you are in the directory tree with `pwd`. Confirm with `ls` that the directory is empty. How would you get to this location using an absolute path?

Expected outcome. `pwd` should tell you `/home/you/newdir`, and `ls` then has no output, meaning there is nothing to list. The absolute path is `/home/you/newdir`.

Exercise. Let's quickly create a file in this directory: *`touch`* `onefile`, and another directory: `mkdir otherdir`. Do `ls` and confirm that there are a new file and directory.

Expected outcome. You should now have:

```
/home/you/
└── newdir/ ..... you are here
    ├── onefile
    └── otherdir/
```

The `ls` command has a very useful option: with `ls -a` you see your regular files and hidden files, which have a name that starts with a dot. Doing `ls -a` in your new directory should tell you that there are the following files:

```
/home/you/
└─ newdir/.....................................you are here
   ├─ .
   ├─ ..
   ├─ onefile
   └─ otherdir/
```

The single dot is the current directory, and the double dot is the directory one level back.

Exercise. Predict where you will be after `cd ./otherdir/..` and check to see if you were right.

> *Expected outcome.* The single dot sends you to the current directory, so that does not change anything. The `otherdir` part makes that subdirectory your current working directory. Finally, `..` goes one level back. In other words, this command puts your right back where you started.

Since your home directory is a special place, there are shortcuts for `cd`'ing to it: `cd` without arguments, `cd ~`, and `cd $HOME` all get you back to your home.

Go to your home directory, and from there do `ls newdir` to check the contents of the first directory you created, without having to go there.

Exercise. What does `ls ..` do?

> *Expected outcome.* Recall that `..` denotes the directory one level up in the tree: you should see your own home directory, plus the directories of any other users.

Exercise. Can you use `ls` to see the contents of someone else's home directory? In the previous exercise you saw whether other users exist on your system. If so, do `ls ../thatotheruser`.

> *Expected outcome.* If this is your private computer, you can probably view the contents of the other user's directory. If this is a university computer or so, the other directory may very well be protected – permissions are discussed in the next section – and you get `ls: ../otheruser: Permission denied`.

Make an attempt to move into someone else's home directory with `cd`. Does it work?

You can make copies of a directory with `cp`, but you need to add a flag to indicate that you recursively copy the contents: `cp -r`. Make another directory `somedir` in your home so that you have

```
/home/you/
├─ newdir/.............................you have been working in this one
└─ somedir/.....................................you just created this one
```

What is the difference between `cp -r newdir somedir` and `cp -r newdir thirddir` where `thirddir` is not an existing directory name?

1.1.3 Permissions

> **Purpose.** In this section you will learn about how to give various users on your system permission to do (or not to do) various things with your files.

Unix files, including directories, have permissions, indicating 'who can do what with this file'. Actions that can be performed on a file fall into three categories:

- reading `r`: any access to a file (displaying, getting information on it) that does not change the file;
- writing `w`: access to a file that changes its content, or even its metadata such as 'date modified';
- executing `x`: if the file is executable, to run it; if it is a directory, to enter it.

The people who can potentially access a file are divided into three classes too:

- the user `u`: the person owning the file;
- the group `g`: a group of users to which the owner belongs;
- other `o`: everyone else.

These nine permissions are rendered in sequence

user	*group*	*other*
rwx	*rwx*	*rwx*

For instance `rw-r--r--` means that the owner can read and write a file, the owner's group and everyone else can only read.

Permissions are also rendered numerically in groups of three bits, by letting `r` $= 4$, `w` $= 2$, `x` $= 1$:

rwx
421

Common codes are $7 =$ `rwx` and $6 =$ `rw`. You will find many files that have permissions 755 which stands for an executable that everyone can run, but only the owner can change, or 644 which stands for a data file that everyone can see but again only the owner can alter. You can set permissions by the *chmod* command:

```
chmod <permissions> file         # just one file
chmod -R <permissions> directory # directory, recursively
```

Examples:

```
chmod 766 file  # set to rwxrw-rw-
chmod g+w file  # give group write permission
chmod g=rx file # set group permissions
chod o-w  file  # take away write permission from others
chmod o=  file  # take away all permissions from others.
chmod g+r,o-x file # give group read permission
                   # remove other execute permission
```

The man page gives all options.

Exercise. Make a file `foo` and do `chmod u-r foo`. Can you now inspect its contents? Make the file readable again, this time using a numeric code. Now make the file readable to your classmates. Check by having one of them read the contents.

Expected outcome. When you've made the file 'unreadable' by yourself, you can still `ls` it, but not `cat` it: that will give a 'permission denied' message.

Make a file `com` with the following contents:

```
#!/bin/sh
echo "Hello world!"
```

This is a legitimate shell script. What happens when you type `./com`? Can you make the script executable?

In the three permission categories it is who 'you' and 'others' refer to. How about 'group'? The command *groups* tells you all the groups you are in, and `ls -l` tells you what group a file belongs to. Analogous to `chmod`, you can use *chgrp* to change the group to which a file belongs, to share it with a user who is also in that group. Adding a user to a group sometimes needs system priviliges.

1.1.4 Wildcards

You already saw that `ls filename` gives you information about that one file, and `ls` gives you all files in the current directory. To see files with certain conditions on their names, the *wildcard* mechanism exists. The following wildcards exist:

- `*` any number of characters.
- `?` any character.

Example:

```
%% ls
s       sk      ski     skiing  skill
%% ls ski*
ski     skiing  skill
```

The second option lists all files whose name start with `ski`, followed by any number of other characters'; below you will see that in different contexts `ski*` means '`sk` followed by any number of `i` characters'. Confusing, but that's the way it is.

1.2 Text searching and regular expressions

Purpose. In this section you will learn how to search for text in files.

For this section you need at least one file that contains some amount of text. You can for instance get random text from http://www.lipsum.com/feed/html.

The *grep* command can be used to search for a text expression in a file.

Exercise. Search for the letter `q` in your text file with `grep q yourfile` and search for it in all files in your directory with `grep q *`. Try some other searches.

Expected outcome. In the first case, you get a listing of all lines that contain a `q`; in the second case, `grep` also reports what file name the match was found in: `qfile:this line has q in it`.

Caveats. If the string you are looking for does not occur, `grep` will simply not output anything. Remember that this is standard behaviour for Unix commands if there is nothing to report.

In addition to searching for literal strings, you can look for more general expressions.

`^`	the beginning of the line
`$`	the end of the line
`.`	any character
`*`	any number of repetitions
`[xyz]`	any of the characters `xyz`

This looks like the wildcard mechanism you just saw (section 1.1.4) but it's subtly different. Compare the example above with:

```
%% cat s
sk
ski
skill
skiing
%% grep "ski*" s
sk
ski
skill
skiing
```

In the second case you search for a string consisting of `sk` and any number of `i` characters, including zero of them.

Some more examples: you can find

- All lines that contain the letter 'q' with `grep q yourfile`;
- All lines that start with an 'a' with `grep "^a" yourfile` (if your search string contains special characters, it is a good idea to use quote marks to enclose it);
- All lines that end with a digit with `grep "[0-9]$" yourfile`.

Exercise. Construct the search strings for finding

- lines that start with an uppercase character, and
- lines that contain exactly one character.

  Expected outcome. For the first, use the range characters `[]`, for the second use the period to match any character.

Exercise. Add a few lines `x = 1`, `x = 2`, `x = 3` (that is, have different numbers of spaces between `x` and the equals sign) to your test file, and make `grep` commands to search for all assignments to `x`.

The characters in the table above have special meanings. If you want to search that actual character, you have to *escape* it.

Exercise. Make a test file that has both `abc` and `a.c` in it, on separate lines. Try the commands `grep "a.c" file`, `grep a\.c file`, `grep "a\.c" file`.

> *Expected outcome.* You will see that the period needs to be escaped, and the search string needs to be quoted. In the absence of either, you will see that `grep` also finds the `abc` string.

1.2.1 Stream editing with `sed`

Unix has various tools for processing text files on a line-by-line basis. The stream editor *sed* is one example. If you have used the `vi` editor, you are probably used to a syntax like `s/foo/bar/` for making changes. With `sed`, you can do this on the commandline. For instance

```
sed 's/foo/bar/' myfile > mynewfile
```

will apply the substitute command `s/foo/bar/` to every line of `myfile`. The output is shown on your screen so you should capture it in a new file; see section 1.3.2 for more on output *redirection*.

1.2.2 Cutting up lines with `cut`

Another tool for editing lines is *cut*, which will cut up a line and display certain parts of it. For instance,

```
cut -c 2-5 myfile
```

will display the characters in position 2–5 of every line of `myfile`. Make a test file and verify this example.

Maybe more useful, you can give `cut` a delimiter character and have it split a line on occurrences of that delimiter. For instance, your system will mostly likely have a file `/etc/passwd` that contains user information[2], with every line consisting of fields separated by colons. For instance:

```
daemon:*:1:1:System Services:/var/root:/usr/bin/false
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

The seventh and last field is the login shell of the user; `/bin/false` indicates that the user is unable to log in.

You can display users and their login shells with:

```
cut -d ":" -f 1,7 /etc/passwd
```

This tells `cut` to use the colon as delimiter, and to print fields 1 and 7.

2. This is traditionally the case; on Mac OS information about users is kept elsewhere and this file only contains system services.

1.3 Command execution

1.3.1 Search paths

Purpose. In this section you will learn how Unix determines what to do when you type a command name.

If you type a command such as `ls`, the shell does not just rely on a list of commands: it will actually go searching for a program by the name `ls`. This means that you can have multiple different commands with the same name, and which one gets executed depends on which one is found first.

Exercise. What you may think of as 'Unix commands' are often just executable files in a system directory. Do *`which ls`*, and do an `ls -l` on the result

Expected outcome. The location of `ls` is something like `/bin/ls`. If you `ls` that, you will see that it is probably owned by root. Its executable bits are probably set for all users.

The locations where unix searches for commands is the 'search path', which is stored in the *environment variable* (for more details see below) `PATH`.

Exercise. Do `echo $PATH`. Can you find the location of `cd`? Are there other commands in the same location? Is the current directory '`.`' in the path? If not, do `export PATH=".:$PATH"`. Now create an executable file `cd` in the current director (see above for the basics), and do `cd`.

Expected outcome. The path will be a list of colon-separated directories, for instance `/usr/bin:/usr/local/bin:/usr/X11R6/bin`. If the working directory is in the path, it will probably be at the end: `/usr/X11R6/bin:.` but most likely it will not be there. If you put '`.`' at the start of the path, unix will find the local `cd` command before the system one.

Some people consider having the working directory in the path a security risk. If your directory is writable, someone could put a malicious script named `cd` (or any other system command) in your directory, and you would execute it unwittingly.

It is possible to define your own commands as aliases of existing commands.

Exercise. Do `alias chdir=cd` and convince yourself that now `chdir` works just like `cd`. Do `alias rm='rm -i'`; look up the meaning of this in the man pages. Some people find this alias a good idea; can you see why?

Expected outcome. The `-i` 'interactive' option for `rm` makes the command ask for confirmation before each delete. Since unix does not have a trashcan that needs to be emptied explicitly (as on Windows or the Mac OS), this can be a good idea.

1.3.2 Redirection

Purpose. In this section you will learn how to feed one command into another, and how to connect commands to input and output files.

So far, the unix commands you have used have taken their input from your keyboard, or from a file named on the command line; their output went to your screen. There are other possibilities for providing input from a file, or for storing the output in a file.

1.3.2.1 Input redirection

The `grep` command had two arguments, the second being a file name. You can also write `grep string < yourfile`, where the less-than sign means that the input will come from the named file, `yourfile`. This is known as *input redirection*.

1.3.2.2 Output redirection

Conversely, `grep string yourfile > outfile` will take what normally goes to the terminal, and *redirect* the output to `outfile`. The output file is created if it didn't already exist, otherwise it is overwritten. (To append, use `grep text yourfile >> outfile`.)

Exercise. Take one of the grep commands from the previous section, and send its output to a file. Check that the contents of the file are identical to what appeared on your screen before. Search for a string that does not appear in the file and send the output to a file. What does this mean for the output file?

> *Expected outcome.* Searching for a string that does not occur in a file gives no terminal output. If you redirect the output of this `grep` to a file, it gives a zero size file. Check this with `ls` and `wc`.

1.3.2.3 Standard files

Unix has three standard files that handle input and output:

`stdin` is the file that provides input for processes.
`stdout` is the file where the output of a process is written.
`stderr` is the file where error output is written.

In an interactive session, all three files are connected to the user terminal. Using input or output redirection then means that the input is taken or the output sent to a different file than the terminal.

1.3.3 Command sequencing

There are various ways of having multiple commands on a single commandline.

1.3.3.1 Simple sequencing

First of all, you can type

```
command1 ; command2
```

This is convenient if you repeat the same two commands a number of times: you only need to up-arrow once to repeat them both.

There is a problem: if you type

```
cc -o myprog myprog.c ; ./myprog
```

and the compilation fails, the program will still be executed, using an old version of the executable if that exists. This is very confusing.

A better way is:

```
cc -o myprog myprog.c && ./myprog
```

which only executes the second command if the first one was successful.

1.3.3.2 Pipelining

Instead of taking input from a file, or sending output to a file, it is possible to connect two commands together, so that the second takes the output of the first as input. The syntax for this is `cmdone | cmdtwo`; this is called a pipeline. For instance, `grep a yourfile | grep b` finds all lines that contains both an `a` and a `b`.

Exercise. Construct a pipeline that counts how many lines there are in your file that contain the string `th`. Use the `wc` command (see above) to do the counting.

1.3.3.3 Backquoting

There are a few more ways to combine commands. Suppose you want to present the result of `wc` a bit nicely. Type the following command

```
echo The line count is wc -l foo
```

where `foo` is the name of an existing file. The way to get the actual line count echoed is by the backquote:

```
echo The line count is `wc -l foo`
```

Anything in between backquotes is executed before the rest of the command line is evaluated.

Exercise 1.1. The way `wc` is used here, it prints the file name. Can you find a way to prevent that from happening?

1.3.3.4 Grouping in a subshell

Suppose you want to apply output redirection to a couple of commands in a row:

```
configure ; make ; make install > installation.log 2>&1
```

This only catches the last command. You could for instance group the three commands in a subshell and catch the output of that:

```
( configure ; make ; make install ) > installation.log 2>&1
```

1.3.4 Exit status

Commands can fail. If you type a single command on the command line, you see the error, and you act accordingly when you type the next command. When that failing command happens in a script, you have to tell the scrip how to act accordingly. For this, you use the *exit status* of the command: this is a value (zero for success, nonzero otherwise) that is stored in an internal variable, and that you can access with `$?`.

Example. Suppose we have a directory that is not writable

```
[testing] ls -ld nowrite/
dr-xr-xr-x  2 eijkhout  506  68 May 19 12:32 nowrite//
[testing] cd nowrite/
```

and write try to create a file there:

```
[nowrite] cat ../newfile
#!/bin/bash
touch $1
echo "Created file: $1"
[nowrite] newfile myfile
bash: newfile: command not found
[nowrite] ../newfile myfile
touch: myfile: Permission denied
Created file: myfile
[nowrite] ls
[nowrite]
```

The script reports that the file was created even though it wasn't.

Improved script:

```
[nowrite] cat ../betterfile
#!/bin/bash
touch $1
if [ $? -eq 0 ] ; then
    echo "Created file: $1"
else
    echo "Problem creating file: $1"
fi

[nowrite] ../betterfile myfile
```

```
touch: myfile: Permission denied
Problem creating file: myfile
```

1.3.5 Processes

The Unix operating system can run many programs at the same time, by rotating through the list and giving each only a fraction of a second to run each time. The command *ps* can tell you everything that is currently running.

Exercise. Type `ps`. How many programs are currently running? By default `ps` gives you only programs that you explicitly started. Do `ps guwax` for a detailed list of everything that is running. How many programs are running? How many belong to the root user, how many to you?

> *Expected outcome.* To count the programs belonging to a user, pipe the `ps` command through an appropriate `grep`, which can then be piped to `wc`.

In this long listing of `ps`, the second column contains the process numbers. Sometimes it is useful to have those. The `cut` command explained above can cut certain position from a line: type `ps guwax | cut -c 10-14`.

To get dynamic information about all running processes, use the `top` command. Read the man page to find out how to sort the output by CPU usage.

When you type a command and hit return, that command becomes, for the duration of its run, the *foreground process*. Everything else that is running at the same time is a *background process*.

Make an executable file `hello` with the following contents:

```
#!/bin/sh
while [ 1 ] ; do
  sleep 2
  date
done
```

and type `./hello`.

Exercise. Type `Control-z`. This suspends the foreground process. It will give you a number like `[1]` or `[2]` indicating that it is the first or second program that has been suspended or put in the background. Now type `bg` to put this process in the background. Confirm that there is no foreground process by hitting return, and doing an `ls`.

> *Expected outcome.* After you put a process in the background, the terminal is available again to accept foreground commands. If you hit return, you should see the command prompt. However, the background process still keeps generating output.

Exercise. Type `jobs` to see the processes in the current session. If the process you just put in the background was number 1, type `fg %1`. Confirm that it is a foreground process again.

Expected outcome. If a shell is executing a program in the foreground, it will not accept command input, so hitting return should only produce blank lines.

Exercise. When you have made the `hello` script a foreground process again, you can kill it with `Control-c`. Try this. Start the script up again, this time as `./hello &` which immediately puts it in the background. You should also get output along the lines of `[1] 12345` which tells you that it is the first job you put in the background, and that `12345` is its process ID. Kill the script with `kill %1`. Start it up again, and kill it by using the process number.

Expected outcome. The command `kill 12345` using the process number is usually enough to kill a running program. Sometimes it is necessary to use `kill -9 12345`.

1.3.6 Shell customization

Above it was mentioned that `ls -F` is an easy way to see which files are regular, executable, or directories; by typing `alias ls='ls -F'` the `ls` command will automatically expanded to `ls -F` every time it is invoked. If you would like this behaviour in every login session, you can add the `alias` command to your `.profile` file. Other shells than `sh`/`bash` have other files for such customizations.

1.4 Scripting

The unix shells are also programming environments. You will learn more about this aspect of unix in this section.

1.4.1 Shell environent variables

Above you encountered `PATH`, which is an example of an shell, or environment, variable. These are variables that are known to the shell and that can be used by all programs run by the shell. You can see the full list of all variables known to the shell by typing `env`.

You can get the value of a shell variable by prefixing it with a dollar sign. Type the following two commands and compare the output:

```
echo PATH
echo $PATH
```

Exercise. Check on the value of the `HOME` variable by typing `echo $HOME`. Also find the value of `HOME` by piping `env` through `grep`.

Environment variables can be set in a number of ways. The simplest is by an assignment as in other programming languages.

Exercise. Type `a=5` on the commandline. This defines a variable `a`; check on its value by using the `echo` command.

Expected outcome. The shell will respond by typing the value `5`.

Caveats. Beware not to have space around the equals sign; also be sure to use the dollar sign to print the value.

A variable set this way will be known to all subsequent commands you issue in this shell, but not to commands in new shells you start up. For that you need the *`export`* command. Reproduce the following session (the square brackets form the command prompt):

```
[] a=20
[] echo $a
20
[] /bin/bash
[] echo $a

[] exit
exit
[] export a=21
[] /bin/bash
[] echo $a
21
[] exit
```

You can also temporarily set a variable. Replay this scenario:

1. Find an environment variable that does not have a value:
   ```
   [] echo $b

   []
   ```
2. Write a short shell script to print this variable:
   ```
   [] cat > echob
   #!/bin/bash
   echo $b
   ```
   and of course make it execuable: `chmod +x echob`.
3. Now call the script, preceeding it with a setting of the variable `b`:
   ```
   [] b=5 ./echob
   5
   ```
   The syntax where you set the value, as a prefix without using a separate command, sets the value just for that one command.
4. Show that the variable is still undefined:
   ```
   [] echo $b

   []
   ```

That is, you defined the variable just for the execution of a single command.

In section 1.4.2 you will see that the `for` construct also defines a variable; section 1.4.3 shows some more built-in variables that apply in shell scripts.

1.4.2 Control structures

Like any good programming system, the shell has some control structures. Their syntax takes a bit of getting used to. (Different shells have different syntax; in this tutorial we only discuss the bash shell.

In the bash shell, control structures can be written over several lines:

```
if [ $PATH = "" ] ; then
  echo "Error: path is empty"
fi
```

or on a single line:

```
if [ `wc -l file` -gt 100 ] ; then echo "file too long" ; fi
```

There are a number of tests defined, for instance `-f somefile` tests for the existence of a file. Change your script so that it will report `-1` if the file does not exist.

There are also loops. A `for` loop looks like

```
for var in listofitems ; do
  something with $var
done
```

This does the following:

- for each item in `listofitems`, the variable `var` is set to the item, and
- the loop body is executed.

As a simple example:

```
[] for x in a b c ; do echo $x ; done
a
b
c
```

In a more meaningful example, here is how you would make backups of all your `.c` files:

```
for cfile in *.c ; do
  cp $cfile $cfile.bak
done
```

Shell variables can be manipulated in a number of ways. Execute the following commands to see that you can remove trailing characters from a variable:

```
[] a=b.c
[] echo ${a%.c}
b
```

With this as a hint, write a loop that renames all your `.c` files to `.x` files.

1.4.3 Scripting

It is possible to write programs of unix shell commands. First you need to know how to put a program in a file and have it be executed. Make a file `script1` containing the following two lines:

```
#!/bin/bash
echo "hello world"
```

and type `./script1` on the command line. Result? Make the file executable and try again.

You can give your script command line arguments. If you want to be able to call

```
./script1 foo bar
```

you can use variables `$1`,`$2` et cetera in the script:

```
#!/bin/bash

echo "The first argument is $1"
echo "There were $# arguments in all"
```

Write a script that takes as input a file name argument, and reports how many lines are in that file.

Edit your script to test whether the file has less than 10 lines (use the `foo -lt bar` test), and if it does, `cat` the file. Hint: you need to use backquotes inside the test.

The number of command line arguments is available as `$#`. Add a test to your script so that it will give a helpful message if you call it without any arguments.

1.5 Expansion

The shell performs various kinds of expansion on a command line, that is, replacing part of the command-line with different text.

Brace expansion:

```
[] echo a{b,cc,ddd}e
abe acce addde
```

This can for instance be used to delete all extension of some base file name:

```
[] rm tmp.{c,s,o}  # delete tmp.c tmp.s tmp.o
```

Tilde expansion gives your own, or someone else's home directory:

```
[] echo ~
/share/home/00434/eijkhout
[] echo ~eijkhout
/share/home/00434/eijkhout
```

Parameter expansion gives the value of shell variables:

```
[] x=5
[] echo $x
5
```

Undefined variables do not give an error message:

```
[] echo $y
```

There are many variations on parameter expansion. Above you already saw that you can strip trailing characters:

```
[] a=b.c
[] echo ${a%.c}
b
```

Here is how you can deal with undefined variables:

```
[] echo ${y:-0}
0
```

The backquote mechanism (section 1.3.2 above) is known as command substitution. It allows you to evalute part of a command and use it as input for another. For example, if you want to ask what type of file the command `ls` is, do

```
[] file `which ls`
```

This first evalutes `which ls`, giving `/bin/ls`, and then evaluates `file /bin/ls`. As another example, here we backquote a whole pipeline, and do a test on the result:

```
[] echo 123 > w
[] cat w
123
[] wc -c w
       4 w
[] if [ `cat w | wc -c` -eq 4 ] ; then echo four ; fi
four
```

Unix shell programming is very much oriented towards text manipulation, but it is possible to do arithmetic. Arithmetic substitution tells the shell to treat the expansion of a parameter as a number:

```
[] x=1
[] echo $((x*2))
2
```

Integer ranges can be used as follows:

```
[] for i in {1..10} ; do echo $i ; done
1
2
3
4
5
6
7
8
9
10
```

1.6 Startup files

In this tutorial you have seen several mechanisms for customizing the behaviour of your shell. For instance, by setting the `PATH` variable you can extend the locations where the shell looks for executables. Other environment variables (section 1.4.1) you can introduce for your own purposes. Many of these customizations will need to apply to every sessions, so you can have *shell startup files* that will be read at the start of any session.

Unfortunately, there are several startup files, and which one gets read is a complicated functions of circumstances. Here is a good common sense guideline[3]:

- Have a `.profile` that does nothing but read the `.bashrc`:

```
# ~/.profile
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

- Your `.bashrc` does the actual customizations:

```
# ~/.bashrc
# make sure your path is updated
if [ -z ``$MYPATH'' ]; then
  export MYPATH=1
```

3. Many thanks to Robert McLay for figuring this out.

```
  export PATH=$HOME/bin:$PATH
fi
```

1.7 Shell interaction

Interactive use of Unix, in contrast to script writing (section 1.4), is a complicated conversation between the user and the shell. You, the user, type a line, hit return, and the shell tries to interpret it. There are several cases.

- Your line contains one full command, such as `ls foo`: the shell will execute this command.
- You can put more than one command on a line, separated by semicolons: `mkdir foo; cd foo`. The shell will execute these commands in sequence.
- Your input line is not a full command, for instance `while [ 1]`. The shell will recognize that there is more to come, and use a different prompt to show you that it is waiting for the remainder of the command.
- Your input line would be a legitimate command, but you want to type more on a second line. In that case you can end your input line with a backslash character, and the shell will recognize that it needs to hold off on executing your command. In effect, the backslash will hide (*escape*) the return.

When the shell has collected a command line to execute, by using one or more of your input line or only part of one, as described just now, it will apply expansion to the command line (section 1.5). It will then interpret the commandline as a command and arguments, and proceed to invoke that command with the arguments as found.

There are some subtleties here. If you type `ls *.c`, then the shell will reognize the wildcard character and expand it to a command line, for instance `ls foo.c bar.c`. Then it will invoke the `ls` command with the argument list `foo.c bar.c`. Note that `ls` does not receive `*.c` as argument! In cases where you do want the unix command to receive an argument with a wildcard, you need to escape it so that the shell will not expand it. For instance, `find . -name \*.c` will make the shell invoke `find` with arguments `. -name *.c`.

1.8 The system and other users

Unix is a multi-user operating system. Thus, even if you use it on your own personal machine, you are a user with an *account* and you may occasionally have to type in your username and password.

If you are on your personal machine, you may be the only user logged in. On university machines or other servers, there will often be other users. Here are some commands relating to them.

`whoami` show your login name.

`who` show the other users currently logged in.

`finger` `otheruser` get information about another user; you can specify a user's login name here, or their real name, or other identifying information the system knows about.

top which processes are running on the system; use `top -u` to get this sorted the amount of cpu time they are currently taking. (On Linux, try also the `vmstat` command.)

uptime how long has it been since your last reboot?

1.9 The sed and awk tools

Apart from fairly small utilities such as `tr` and `cut`, Unix has some more powerful tools. In this section you will see two tools for line-by-line transformations on text files. Of course this tutorial merely touches on the depth of these tools; for more information see [1, 3].

1.9.1 sed

The streaming editor `sed` is like an editor by remote control, doing simple line edits with a commandline interface. Most of the time you will use `sed` as follows:

```
cat somefile | sed 's/abc/def/:g' > newfile
```

(The use of `cat` here is not strictly necessary.) The `s/abc/def/` part has the effect of replacing `abc` by `def` in every line; the `:g` modifier applies it to every instance in every line rather than just the first.

- If you have more than one edit, you can specify them with
  ```
  sed -e 's/one/two/' -e 's/three/four/'
  ```
- If an edit needs to be done only on certain lines, you can specify that by prefixing the edit with the match string. For instance
  ```
  sed '/^a/s/b/c/'
  ```
  only applies the edit on lines that start with an `a`. (See section 1.2 for regular expressions.)
- Traditionally, `sed` could only function in a stream, so the output file always had to be different from the input. The GNU version, which is standard on Linux systems, has a flag `-i` which edits 'in place':
  ```
  sed -e 's/ab/cd/' -e 's/ef/gh/' -i thefile
  ```

1.9.2 awk

The `awk` utility also operates on each line, but it can be described as having a memory. An awk program consists of a sequence of pairs, where each pair consists of a match string and an action. The simplest awk program is

```
cat somefile | awk '{ print }'
```

where the match string is omitted, meaning that all lines match, and the action is to print the line. Awk breaks each line into fields separated by whitespace. A common application of awk is to print a certain field:

```
awk '{print $2}' file
```

prints the second field of each line.

Suppose you want to print all subroutines in a Fortran program; this can be accomplished with

```
awk '/subroutine/ {print}' yourfile.f
```

Exercise 1.2. Build a commandpipeline that prints of each subroutine header only the subroutine name. For this you first use `sed` to replace the parentheses by spaces, then `awk` to print the subroutine name field.

Awk has variables with which it can remember things. For instance, instead of just printing the second field of every line, you can make a list of them and print that later:

```
cat myfile | awk 'BEGIN {v="Fields:"} {v=v " " $2} END {print v}'
```

As another example of the use of variables, here is how you would print all lines in between a `BEGIN` and `END` line:

```
cat myfile | awk '/END/ {p=0} p==1 {print} /BEGIN/ {p=1} '
```

Exercise 1.3. The placement of the match with `BEGIN` and `END` may seem strange. Rearrange the awk program, test it out, and explain the results you get.

1.10 Review questions

Exercise 1.4. Devise a pipeline that counts how many users are logged onto the system, whose name starts with a vowel and ends with a consonant.

Exercise 1.5. Write a shell script for making backups. When you call this script as `./backup somefile` it should test whether `somefile.bak` exists, and give a warning if it does. In either case, it should copy the original file to a backup.

Chapter 2

Text editing

A good text editor is an indispensable tool for any programmer[1]. In this tutorial you will learn the basics of the two most common Unix editors: *vi* and *emacs*.

2.1 Vi

Purpose. In this section you will learn the basics of text editing with 'vi'.

Good set of tips: https://www.cs.oberlin.edu/~kuperman/help/vim/home.html

The vi editor (pronounced 'vee-eye', not 'vai') has a 'modal' setup: you are either in input mode, where what you type becomes part of your file, or you are in edit mode, where your typing is interpreted as commands.

2.1.1 Indentation

Decent settings for new files:

```
set smartindent
set shiftwidth=2
syntax on
```

The command `[[=]]` applies the right indentation to the block surrounding the cursor position.

2.2 Emacs

Purpose. In this section you will learn the basics of text editing with 'emacs'.

The emacs is always in input mode: ordinary characters you type become part of the file you are currently editing. To execute commands you need the 'Control' and 'Escape' (for historical reasons often called 'Meta') keys.

---

1. Alternatively, you could use an Integrated Development Environment (IDE) such as *Visual Studio* or *Eclipse*, but they are usually harder to customize, not installed on every system, et cetera. Really: make sure that you learn at least one common text editor.

2.2.1 Indentation

Emacs can usually detect the proper indentation scheme from the name or extension of your file, and it will go into the appropriate mode automatically. You can also force the mode explicitly by `Esc x latex-mode` and similar commands.

However, a file does not remember its mode, so if emacs can not deduce the mode of your file, put a line

```
// -*- c++ -*-
```

at the top of your file. The first non-blank character(s) of this line are chosen to make it a comment in the language of a file. For example, to indicate that a TEX file is for LATEX, you would use

```
% -*- latex -*-
```

Chapter 3

Compilers and libraries

3.1 An introduction to binary files

Purpose. In this section you will be introduced to the different types of binary files that you encounter while programming.

One of the first things you become aware of when you start programming is the distinction between the readable source code, and the unreadable, but executable, program code. In this tutorial you will learn about a couple more file types:

- A source file can be compiled to an *object file*, which is a bit like a piece of an executable: by itself it does nothing, but it can be combined with other object files to form an executable.
- A *library* is a bundle of object files that can be used to form an executable. Often, libraries are written by an expert and contain code for specialized purposes such as linear algebra manipulations. Libraries are important enough that they can be commercial, to be bought if you need expert code for a certain purpose.

You will now learn how these types of files are created and used.

3.2 Simple compilation

Purpose. In this section you will learn about executables and object files.

Let's start with a simple program that has the whole source in one file.

One file: `hello.c`

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

int main() {
  printf("hello world\n");
  return 0;
}
```

Compile this program with your favourite compiler; we will use `gcc` in this tutorial, but substitute your own as desired. As a result of the compilation, a file `a.out` is created, which is the executable.

```
%% gcc hello.c
%% ./a.out
hello world
```

You can get a more sensible program name with the `-o` option:

```
%% gcc -o helloprog hello.c
%% ./helloprog
hello world
```

Now we move on to a program that is in more than one source file.

Main program: `fooprog.c`

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

extern bar(char*);

int main() {
  bar("hello world\n");
  return 0;
}
```

Subprogram: `fooprog.c`

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

void bar(char *s) {
  printf("%s",s);
  return;
}
```

As before, you can make the program with one command.

```
%% gcc -o foo fooprog.c foosub.c
%% ./foo
hello world
```

However, you can also do it in steps, compiling each file separately and then linking them together.

```
%% gcc -c fooprog.c
%% gcc -c foosub.c
%% gcc -o foo fooprog.o foosub.o
%% ./foo
```

```
hello world
```

The `-c` option tells the compiler to compile the source file, giving an *object file*. The third command than acts as the *linker*, tieing together the object files into an executable. (With programs that are spread over several files there is always the danger of editing a subroutine definition and then forgetting to update all the places it is used. See the 'make' tutorial, section 4, for a way of dealing with this.)

3.3 Libraries

Purpose. In this section you will learn about libraries.

If you have written some subprograms, and you want to share them with other people (perhaps by selling them), then handing over individual object files is inconvenient. Instead, the solution is to combine them into a library. First we look at *static libraries*, for which the *archive utility* `ar` is used. A static library is linked into your executable, becoming part of it. This may lead to large executables; you will learn about shared libraries next, which do not suffer from this problem.

Create a directory to contain your library (depending on what your library is for this can be a system directory such as `/usr/bin`), and create the library file there.

```
%% mkdir ../lib
%% ar cr ../lib/libfoo.a foosub.o
```

The *nm* command tells you what's in the library:

```
%% nm ../lib/libfoo.a

../lib/libfoo.a(foosub.o):
00000000 T _bar
         U _printf
```

Line with `T` indicate functions defined in the library file; a `U` indicates a function that is used.

The library can be linked into your executable by explicitly giving its name, or by specifying a library path:

```
%% gcc -o foo fooprog.o ../lib/libfoo.a
# or
%% gcc -o foo fooprog.o -L../lib -lfoo
%% ./foo
hello world
```

A third possibility is to use the `LD_LIBRARY_PATH` shell variable. Read the man page of your compiler for its use, and give the commandlines that create the `foo` executable, linking the library through this path.

Although they are somewhat more complicated to use, *shared libraries* have several advantages. For instance, since they are not linked into the executable but only loaded at runtime, they lead to (much) smaller executables. They are not created with `ar`, but through the compiler. For instance:

```
%% gcc -dynamiclib -o ../lib/libfoo.so foosub.o
%% nm ../lib/libfoo.so

../lib/libfoo.so(single module):
00000fc4 t __dyld_func_lookup
00000000 t __mh_dylib_header
00000fd2 T _bar
         U _printf
00001000 d dyld__mach_header
00000fb0 t dyld_stub_binding_helper
```

Shared libraries are not actually linked into the executable; instead, the executable will contain the information where the library is to be found at execution time:

```
%% gcc -o foo fooprog.o -L../lib -Wl,-rpath,`pwd`/../lib -lfoo
%% ./foo
hello world
```

The link line now contains the library path twice:

1. once with the `-L` directive so that the linker can resolve all references, and
2. once with the linker directive `-Wl,-rpath,`pwd`/../lib` which stores the path into the executable so that it can be found at runtime.

Build the executable again, but without the `-Wl` directive. Where do things go wrong and why? You can also fix this problem by using `LD_LIBRARY_PATH`. Explore this.

Use the command `ldd` to get information about what shared libraries your executable uses. (On Mac OS X, use `otool -L` instead.)

Chapter 4

Managing projects with Make

The *Make* utility helps you manage the building of projects: its main task is to facilitate rebuilding only those parts of a multi-file project that need to be recompiled or rebuilt. This can save lots of time, since it can replace a minutes-long full installation by a single file compilation. *Make* can also help maintaining multiple installations of a program on a single machine, for instance compiling a library with more than one compiler, or compiling a program in debug and optimized mode.

Make is a Unix utility with a long history, and traditionally there are variants with slightly different behaviour, for instance on the various flavours of Unix such as HP-UX, AUX, IRIX. These days, it is advisable, no matter the platform, to use the GNU version of Make which has some very powerful extensions; it is available on all Unix platforms (on Linux it is the only available variant), and it is a *de facto* standard. The manual is available at http://www.gnu.org/software/make/manual/make.html, or you can read the book [7].

There are other build systems, most notably *Scons* and *Bjam*. We will not discuss those here. The examples in this tutorial will be for the C and Fortran languages, but *Make* can work with any language, and in fact with things like TEX that are not really a language at all; see section 4.6.

4.1 A simple example

Purpose. In this section you will see a simple example, just to give the flavour of *Make*.

The files for this section can be downloaded from http://tinyurl.com/ISTC-make-tutorial.

4.1.1 C

Make the following files:

foo.c

```
#include "bar.h"
int c=3;
int d=4;
int main()
```

```
{
  int a=2;
  return(bar(a*c*d));
}
```

bar.c

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return(b*a);
}
```

bar.h

```
extern int bar(int);
```

and a makefile:

Makefile

```
fooprog : foo.o bar.o
	cc -o fooprog foo.o bar.o
foo.o : foo.c
	cc -c foo.c
bar.o : bar.c
	cc -c bar.c
clean :
	rm -f *.o fooprog
```

The makefile has a number of rules like

```
foo.o : foo.c
<TAB>cc -c foo.c
```

which have the general form

```
target : prerequisite(s)
<TAB>rule(s)
```

where the rule lines are indented by a `TAB` character.

A rule, such as above, states that a 'target' file `foo.o` is made from a 'prerequisite' `foo.c`, namely by executing the command `cc -c foo.c`. The precise definition of the rule is:

- if the target `foo.o` does not exist or is older than the prerequisite `foo.c`,
- then the command part of the rule is executed: `cc -c foo.c`
- If the prerequisite is itself the target of another rule, than that rule is executed first.

Probably the best way to interpret a rule is:

- if any prerequisite has changed,
- then the target needs to be remade,
- and that is done by executing the commands of the rule;
- checking the prerequisite requires a recursive application of make:
  - if the prerequisite does not exist, find a rule to create it;
  - if the prerequisity already exists, check applicable rules to see if it needs to be remade.

If you call `make` without any arguments, the first rule in the makefile is evaluated. You can execute other rules by explicitly invoking them, for instance `make foo.o` to compile a single file.

Exercise. Call `make`.

> *Expected outcome.* The above rules are applied: `make` without arguments tries to build the first target, `fooprog`. In order to build this, it needs the prerequisites `foo.o` and `bar.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foo.o` and `bar.o`, and a link command for `fooprog`.
>
> *Caveats.* Typos in the makefile or in file names can cause various errors. In particular, make sure you use tabs and not spaces for the rule lines. Unfortunately, debugging a makefile is not simple. *Make*'s error message will usually give you the line number in the make file where the error was detected.

Exercise. Do `make clean`, followed by `mv foo.c boo.c` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foo.c`. This error was caused when `foo.c` was a prerequisite for making `foo.o`, and was found not to exist. *Make* then went looking for a rule to make it and no rule for creating `.c` files exists.

Now add a second argument to the function `bar`. This requires you to edit `bar.c` and `bar.h`: go ahead and make these edits. However, it also requires you to edit `foo.c`, but let us for now 'forget' to do that. We will see how *Make* can help you find the resulting error.

Exercise. Call `make` to recompile your program. Did it recompile `foo.c`?

> *Expected outcome.* Even through conceptually `foo.c` would need to be recompiled since it uses the `bar` function, *Make* did not do so because the makefile had no rule that forced it.

In the makefile, change the line

```
foo.o : foo.c
```

to

```
foo.o : foo.c bar.h
```

which adds `bar.h` as a prerequisite for `foo.o`. This means that, in this case where `foo.o` already exists, *Make* will check that `foo.o` is not older than any of its prerequisites. Since `bar.h` has been edited, it is younger than `foo.o`, so `foo.o` needs to be reconstructed.

Exercise. Confirm that the new makefile indeed causes `foo.o` to be recompiled if `bar.h` is changed. This compilation will now give an error, since you 'forgot' to edit the use of the `bar` function.

4.1.2 Fortran

Make the following files:

foomain.F

```
program test
use testmod

call func(1,2)

end program
```

foomod.F

```
module testmod

contains

subroutine func(a,b)
integer a,b
print *,a,b,c
end subroutine func

end module
```

and a makefile:

Makefile

```
fooprog : foomain.o foomod.o
	gfortran -o fooprog foo.o foomod.o
foomain.o : foomain.F
	gfortran -c foomain.F
foomod.o : foomod.F
	gfortran -c foomod.F
clean :
	rm -f *.o fooprog
```

If you call `make`, the first rule in the makefile is executed. Do this, and explain what happens.

Exercise. Call `make`.

Expected outcome. The above rules are applied: `make` without arguments tries to build the first target, `foomain`. In order to build this, it needs the prerequisites `foomain.o` and `foomod.o`, which do not exist. However, there are rules for making them, which `make` recursively invokes. Hence you see two compilations, for `foomain.o` and `foomod.o`, and a link command for `fooprog`.

Caveats. Typos in the makefile or in file names can cause various errors. Unfortunately, debugging a makefile is not simple. You will just have to understand the errors, and make the corrections.

Exercise. Do `make clean`, followed by `mv foomod.c boomod.c` and `make` again. Explain the error message. Restore the original file name.

> *Expected outcome. Make* will complain that there is no rule to make `foomod.c`. This error was caused when `foomod.c` was a prerequisite for `foomod.o`, and was found not to exist. *Make* then went looking for a rule to make it, and no rule for making `.F` files exists.

Now add an extra parameter to `func` in `foomod.F` and recompile.

Exercise. Call `make` to recompile your program. Did it recompile `foomain.F`?

> *Expected outcome.* Even through conceptually `foomain.F` would need to be recompiled, *Make* did not do so because the makefile had no rule that forced it.

Change the line

```
foomain.o : foomain.F
```

to

```
foomain.o : foomain.F foomod.o
```

which adds `foomod.o` as a prerequisite for `foomain.o`. This means that, in this case where `foomain.o` already exists, *Make* will check that `foomain.o` is not older than any of its prerequisites. Recursively, *Make* will then check if `foomode.o` needs to be updated, which is indeed the case. After recompiling `foomode.F`, `foomode.o` is younger than `foomain.o`, so `foomain.o` will be reconstructed.

Exercise. Confirm that the corrected makefile indeed causes `foomain.F` to be recompiled.

4.1.3 About the make file

The make file needs to be called `makefile` or `Makefile`; it is not a good idea to have files with both names in the same directory. If you want *Make* to use a different file as make file, use the syntax `make -f My_Makefile`.

4.2 Variables and template rules

> **Purpose.** In this section you will learn various work-saving mechanisms in *Make*, such as the use of variables and of template rules.

4.2.1 Makefile variables

It is convenient to introduce variables in your makefile. For instance, instead of spelling out the compiler explicitly every time, introduce a variable in the makefile:

```
CC = gcc
FC = gfortran
```

and use `${CC}` or `${FC}` on the compile lines:

```
foo.o : foo.c
        ${CC} -c foo.c
foomain.o : foomain.F
        ${FC} -c foomain.F
```

Exercise. Edit your makefile as indicated. First do `make clean`, then `make foo` (C) or `make fooprog` (Fortran).

Expected outcome. You should see the exact same compile and link lines as before.

Caveats. Unlike in the shell, where braces are optional, variable names in a makefile have to be in braces or parentheses. Experiment with what hapens if you forget the braces around a variable name.

One advantage of using variables is that you can now change the compiler from the commandline:

```
make CC="icc -O2"
make FC="gfortran -g"
```

Exercise. Invoke *Make* as suggested (after `make clean`). Do you see the difference in your screen output?

Expected outcome. The compile lines now show the added compiler option `-O2` or `-g`.

Make also has *automatic variables*:

- `$@` The target. Use this in the link line for the main program.
- `$^` The list of prerequisites. Use this also in the link line for the program.
- `$<` The first prerequisite. Use this in the compile commands for the individual object files.
- `$*` In *template rules* (section 4.2.2) this matches the template part, the part corresponding to the `%`.

Using these variables, the rule for `fooprog` becomes

```
fooprog : foo.o bar.o
        ${CC} -o $@ $^
```

and a typical compile line becomes

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

You can also declare a variable

```
THEPROGRAM = fooprog
```

and use this variable instead of the program name in your makefile. This makes it easier to change your mind about the name of the executable later.

Exercise. Edit your makefile to add this variable definition, and use it instead of the literal program name. Construct a commandline so that your makefile will build the executable `fooprog_v2`.

> *Expected outcome.* You need to specify the `THEPROGRAM` variable on the commandline using the syntax `make VAR=value`.
>
> *Caveats.* Make sure that there are no spaces around the equals sign in your commandline.

The full list of these automatic variables can be found at https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html.

4.2.2 Template rules

So far, you wrote a separate rule for each file that needed to be compiled. However, the rules for the various `.c` files are very similar:

- the rule header (`foo.o : foo.c`) states that a source file is a prerequisite for the object file with the same base name;
- and the instructions for compiling (`${CC} -c $<`) are even character-for-character the same, now that you are using *Make*'s built-in variables;
- the only rule with a difference is

  ```
  foo.o : foo.c bar.h
          ${CC} -c $<
  ```

  where the object file depends on the source file and another file.

We can take the commonalities and summarize them in one *template rule*[1]:

```
%.o : %.c
        ${CC} -c $<
%.o : %.F
        ${FC} -c $<
```

This states that any object file depends on the C or Fortran file with the same base name. To regenerate the object file, invoke the C or Fortran compiler with the `-c` flag. These template rules can function as a replacement for the multiple specific targets in the makefiles above, except for the rule for `foo.o`.

The dependence of `foo.o` on `bar.h`, or `foomain.o` on `foomod.o`, can be handled by adding a rule

1. This mechanism is the first instance you'll see that only exists in GNU make, though in this particular case there is a similar mechanism in standard make. That will not be the case for the wildcard mechanism in the next section.

```
# C
foo.o : bar.h
# Fortran
foomain.o : foomod.o
```

with no further instructions. This rule states, 'if file `bar.h` or `foomod.o` changed, file `foo.o` or `foomain.o` needs updating' too. *Make* will then search the makefile for a different rule that states how this updating is done, and it will find the template rule.

Exercise. Change your makefile to incorporate these ideas, and test.

4.2.3 Wildcards

Your makefile now uses one general rule for compiling any source file. Often, your source files will be all the `.c` or `.F` files in your directory, so is there a way to state 'compile everything in this directory'? Indeed there is.

Add the following lines to your makefile, and use the variable `COBJECTS` or `FOBJECTS` wherever appropriate. The command *`wildcard`* gives the result of `ls`, and you can manipulate file names with *`patsubst`*.

```
# wildcard: find all files that match a pattern
CSOURCES := ${wildcard *.c}
# pattern substitution: replace one pattern string by another
COBJECTS := ${patsubst %.c,%.o,${SRC}}

FSOURCES := ${wildcard *.F}
FOBJECTS := ${patsubst %.F,%.o,${SRC}}
```

4.2.4 Conditionals

There are various ways of making the behaviour of a makefile dynamic. You can for instance put a shell conditional in an action line. However, this can make for a cluttered makefile; an easier way is to use makefile conditionals. There are two types of conditionals: tests on string equality, and tests on environment variables.

The first type looks like

```
ifeq "${HOME}" "/home/thisisme"
  # case where the executing user is me
else
  # case where it's someone else
endif
```

and in the second case the test looks like

```
ifdef SOME_VARIABLE
```

The text in the true and false part can be most any part of a makefile. For instance, it is possible to let one of the action lines in a rule be conditionally included. However, most of the times you will use conditionals to make the definition of variables dependent on some condition.

Exercise. Let's say you want to use your makefile at home and at work. At work, your employer has a paid license to the Intel compiler `icc`, but at home you use the open source Gnu compiler `gcc`. Write a makefile that will work in both places, setting the appropriate value for `CC`.

4.3 Miscellania

4.3.1 What does this makefile do?

Above you learned that issuing the `make` command will automatically execute the first rule in the makefile. This is convenient in one sense[2], and inconvenient in another: the only way to find out what possible actions a makefile allows is to read the makefile itself, or the – usually insufficient – documentation.

A better idea is to start the makefile with a target

```
info :
        @echo "The following are possible:"
        @echo "  make"
        @echo "  make clean"
```

Now `make` without explicit targets informs you of the capabilities of the makefile.

If your makefile gets longer, you might want to document each section like this. This runs into a problem: you can not have two rules with the same target, `info` in this case. However, if you use a double colon it *is* possible. Your makefile would have the following structure:

```
info ::
        @echo "The following target are available:"
        @echo "  make install"
install :
        # ..... instructions for installing
info ::
       @echo "  make clean"
clean :
       # ..... instructions for cleaning
```

2. There is a convention among software developers that a package can be installed by the sequence `./configure ; make ; make install`, meaning: Configure the build process for this computer, Do the actual build, Copy files to some system directory such as `/usr/bin`.

4.3.2 Phony targets

The example makefile contained a target `clean`. This uses the *Make* mechanisms to accomplish some actions that are not related to file creation: calling `make clean` causes *Make* to reason 'there is no file called `clean`, so the following instructions need to be performed'. However, this does not actually cause a file `clean` to spring into being, so calling `make clean` again will make the same instructions being executed.

To indicate that this rule does not actually make the target, declare

```
.PHONY : clean
```

One benefit of declaring a target to be phony, is that the *Make* rule will still work, even if you have a file named `clean`.

4.3.3 Predefined variables and rules

Calling `make -p yourtarget` causes make to print out all its actions, as well as the values of all variables and rules, both in your makefile and ones that are predefined. If you do this in a directory where there is no makefile, you'll see that make actually already knows how to compile `.c` or `.F` files. Find this rule and find the definition of the variables in it.

You see that you can customize make by setting such variables as `CFLAGS` or `FFLAGS`. Confirm this with some experimentation. If you want to make a second makefile for the same sources, you can call `make -f othermakefile` to use this instead of the default `Makefile`.

Note, by the way, that both `makefile` and `Makefile` are legitimate names for the default makefile. It is not a good idea to have both `makefile` and `Makefile` in your directory.

4.3.4 Using the target as prerequisite

Suppose you have two different targets that are treated largely the same. You would want to write:

```
PROGS = myfoo other
${PROGS} : $@.o # this is wrong!!
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

and saying `make myfoo` would cause

```
cc -c myfoo.c
cc -o myfoo myfoo.o ${list of libraries}
```

and likewise for `make other`. What goes wrong here is the use of `$@.o` as prerequisite. In Gnu Make, you can repair this as follows[3]:

3. Technical explanation: Make will now look at lines twice: the first time `$$` gets converted to a single `$`, and in the second pass `$@` becomes the name of the target.

```
.SECONDEXPANSION:
${PROGS} : $$@.o
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

Exercise. Write a second main program `foosecond.c` or `foosecond.F`, and change your makefile so that the calls `make foo` and `make foosecond` both use the same rule.

4.4 Shell scripting in a Makefile

Purpose. In this section you will see an example of a longer shell script appearing in a makefile rule.

In the makefiles you have seen so far, the command part was a single line. You can actually have as many lines there as you want. For example, let us make a rule for making backups of the program you are building.

Add a `backup` rule to your makefile. The first thing it needs to do is make a backup directory:

```
.PHONY : backup
backup :
        if [ ! -d backup ] ; then
          mkdir backup
        fi
```

Did you type this? Unfortunately it does not work: every line in the command part of a makefile rule gets executed as a single program. Therefore, you need to write the whole command on one line:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
```

or if the line gets too long:

```
backup :
        if [ ! -d backup ] ; then \
          mkdir backup ; \
        fi
```

Next we do the actual copy:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

But this backup scheme only saves one version. Let us make a version that has the date in the name of the saved program.

The Unix `date` command can customize its output by accepting a format string. Type the following: `date` This can be used in the makefile.

Exercise. Edit the `cp` command line so that the name of the backup file includes the current date.

> *Expected outcome.* Hint: you need the backquote. Consult the Unix tutorial, section 1.3.3, if you do not remember what backquotes do.

If you are defining shell variables in the command section of a makefile rule, you need to be aware of the following. Extend your `backup` rule with a loop to copy the object files:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $f backup ; \
        done
```

(This is not the best way to copy, but we use it for the purpose of demonstration.) This leads to an error message, caused by the fact that *Make* interprets `$f` as an environment variable of the outer process. What works is:

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
          cp $$f backup ; \
        done
```

(In this case *Make* replaces the double dollar by a single one when it scans the commandline. During the execution of the commandline, `$f` then expands to the proper filename.)

4.5 Practical tips for using Make

Here are a couple of practical tips.

- *Debugging* a makefile is often frustratingly hard. Just about the only tool is the `-p` option, which prints out all the rules that Make is using, based on the current makefile.
- You will often find yourself first typing a make command, and then invoking the program. Most Unix shells allow you to use commands from the *shell command history* by using the up arrow key. Still, this may get tiresome, so you may be tempted to write
  ```
  make myprogram ; ./myprogram -options
  ```

and keep repeating this. There is a danger in this: if the make fails, for instance because of compilation problems, your program will still be executed. Instead, write

```
make myprogram && ./myprogram -options
```

which executes the program conditional upon make concluding successfully.

4.6 A Makefile for LaTeX

The *Make* utility is typically used for compiling programs, but other uses are possible too. In this section, we will discuss a makefile for LaTeX documents.

We start with a very basic makefile:

```
info :
        @echo "Usage: make foo"
        @echo "where foo.tex is a LaTeX input file"

%.pdf : %.tex
        pdflatex $<
```

The command `make myfile.pdf` will invoke `pdflatex myfile.tex`, if needed, once. Next we repeat invoking `pdflatex` until the log file no longer reports that further runs are needed:

```
%.pdf : %.tex
        pdflatex $<
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
            | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

We use the `${basename fn}` macro to extract the base name without extension from the target name.

In case the document has a bibliography or index, we run `bibtex` and `makeindex`.

```
%.pdf : %.tex
        pdflatex ${basename $@}
        -bibtex ${basename $@}
        -makeindex ${basename $@}
        while [ `cat ${basename $@}.log | grep "Rerun to get" \
                  | wc -l` -gt 0 ] ; do \
          pdflatex ${basename $@} ; \
        done
```

The minus sign at the start of the line means that *Make* should not abort if these commands fail.

Finally, we would like to use *Make*'s facility for taking dependencies into account. We could write a makefile that has the usual rules

```
mainfile.pdf : mainfile.tex includefile.tex
```

but we can also discover the include files explicitly. The following makefile is invoked with

```
make pdf TEXTFILE=mainfile
```

The `pdf` rule then uses some shell scripting to discover the include files (but not recursively), and it calls *Make* again, invoking another rule, and passing the dependencies explicitly.

```
pdf :
        export includes=`grep "^.input " ${TEXFILE}.tex \
            | awk '{v=v FS $$2".tex"} END {print v}'` ; \
        ${MAKE} ${TEXFILE}.pdf INCLUDES="$$includes"

%.pdf : %.tex ${INCLUDES}
        pdflatex $< ; \
        while [ `cat ${basename $@}.log \
                | grep "Rerun to get" | wc -l` -gt 0 ] ; do \
          pdflatex $< ; \
        done
```

This shell scripting can also be done outside the makefile, generating the makefile dynamically.

Chapter 5

Source code control

Source code control systems, also called *revision control* or *version control* systems, are a way of storing software, where not only the current version is stored, but also all previous versions. This is done by maintaining a *repository* for all versions, while one or more users work on a 'checked out' copy of the latest version. Those of the users that are developers can then commit their changes to the repository. Other users then update their local copy. The repository typically resides on a remote machine that is reliably backup up.

There are various reasons for keeping your source in a repository.

- If you work in a team, it is the best way to synchronize your work with your colleagues. It is also a way to document what changes were made, by whom, and why.
- It will allow you to roll back a defective code to a version that worked.
- It allows you to have branches, for instance for customizations that need to be kept out of the main development line. If you are working in a team, a branch is a way to develop a major feature, stay up to date with changes your colleagues make, and only add your feature to the main development when it is sufficiently tested.
- If you work alone, it is a way to synchronize between more than one machine. (You could even imagine traveling without all your files, and installing them from the repository onto a borrowed machine as the need arises.)
- Having a source code repository is one way to backup your work.

There are various source code control systems; in this tutorial you can learn the basics of *Subversion* (also called *svn*), which is probably the most popular of the traditional source code control systems, and Mercurial (or `hg`), which is an example of the new generation of *distributed source code control* systems.

5.1 Workflow in source code control systems

Source code control systems are built around the notion of *repository*: a central store of the files of a project, together with their whole history. Thus, a repository allows you to share files with multiple people, but also to roll back changes, apply patches to old version, et cetera.

The basic actions on a repository are:

- Creating the repository; this requires you to have space and write permissions on some server. Maybe your sysadmin has to do it for you.
- Checking out the repository, that is, making a local copy of its contents in your own space.
- Adding your changes to the repository, and
- Updating your local copy with someone else's changes.

Adding your own changes is not always possible: there are many projects where the developer allows you to check out the repository, but not to incorporate changes. Such a repository is said to be read-only.

Figure 5.1 illustrates these actions for the Subversion system. Users who have checked out the repository

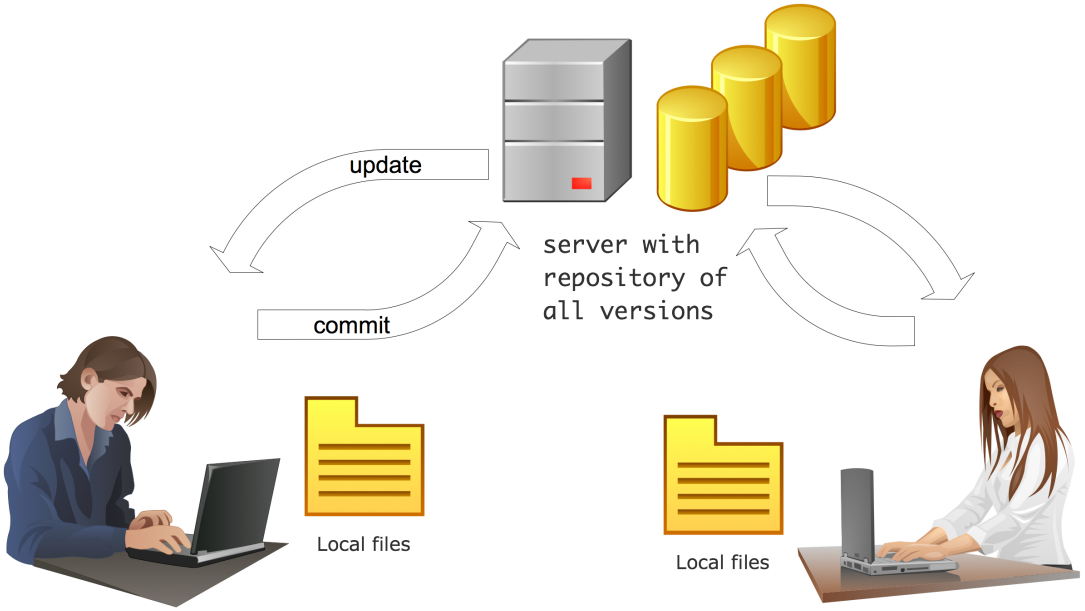

Figure 5.1: Workflow in traditional source code control systems such as Subversion

can edit files, and check in the new versions with the `commit` command; to get the changes committed by other users you use `update`.

One of the uses of committing is that you can roll your code back to an earlier version if you realize you made a mistake or introduced a bug. It also allows you to easily see the difference between different code version. However, committing many small changes may be confusing to other developers, for instance if they come to rely on something you introduce which you later remove again. For this reason, *distributed source code control* systems use two levels of repositories.

There is still a top level that is authoritative, but now there is a lower level, typically of local copies, where you can commit your changes and accumulate them until you finally add them to the central repository. This also makes it easier to contribute to a read-only repository: you make your local changes, and when you are finished you tell the developer to inspect your changes and pull them into the top level repository. This structure is illustrated in figure 5.2.

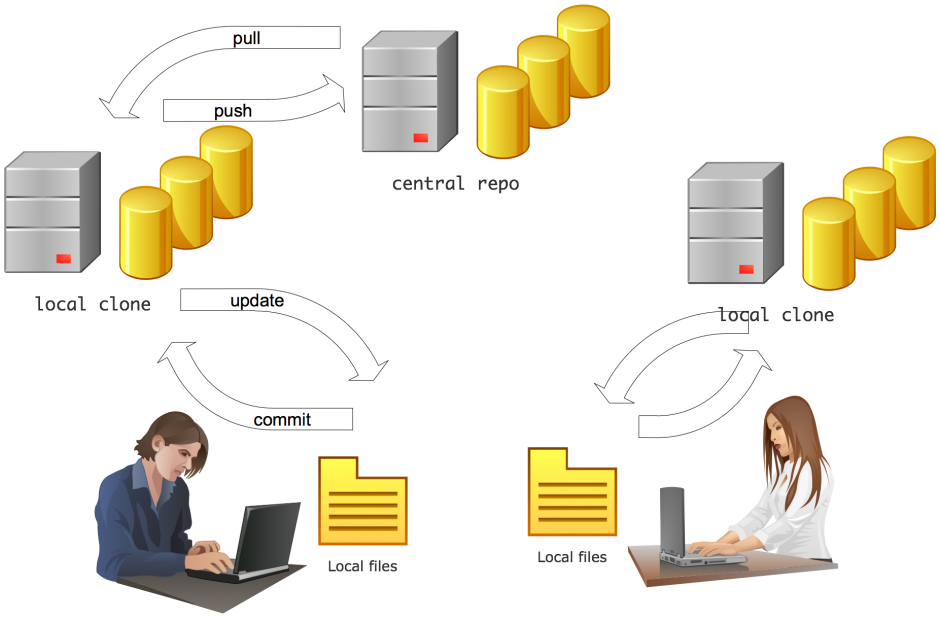

Figure 5.2: Workflow in distributed source code control systems such as Mercurial

5.2 Subversion or SVN

This lab should be done two people, to simulate a group of programmers working on a joint project. You can also do this on your own by using two copies of the repository.

5.2.1 Create and populate a repository

Purpose. In this section you will create a repository and make a local copy to work on.

First we need to have a repository. In practice, you will often use one that has been set up by a sysadmin, but there are several ways to set up a repository yourself.

- There are commercial and free hosting services such as *Google code* `http://code.google.com/projecthosting` (open source only) or *BitBucket* `https://bitbucket.org/` (this has favourable academic licenses). Once you create a repository there, you can make a local copy on your computer:

```
%% svn co http://yourservice.com//yourproject/ project
Checked out revision 0.
```

  where `co` is short for 'check out'[1].

You now have an empty directory `project`.

Exercise. Go into the project directory and see if it is really empty.

1. Alternatively, you can create a repository in the local file system with `svnadmin create ./repository --fs-type=fsfs` and check it out with `svn co file://path.../`; we will not go into this.

Expected outcome. There is a hidden directory `.svn`

The project directory knows where the master copy of the repository is stored.

Exercise. Do `svn info`

Expected outcome. You will see the URL of the repository and the revision number

Caveats. Make sure you are in the right directory. In the outer directories, you will get `svn: warning: '.' is not a working copy`

5.2.2 New files

Purpose. In this section you will make some simple changes: editing an existing file and creating a new file.

One student now makes a file to add to the repository[2].

```
%%  cat > firstfile
a
b
c
d
e
f
^D
```

This file is unknown to `svn`:

```
%% svn status
?       firstfile
```

We need to declare the file as belonging to the repository; a subsequent `svn commit` command then copies it into the master repository.

```
%% svn add firstfile
A         firstfile
%% svn status
A       firstfile
%% svn commit -m "made a first file"
Adding         firstfile
Transmitting file data .
Committed revision 1.
```

Exercise. The second student can now do `svn update` to update their copy of the repository

2. It is also possible to use `svn import` to create a whole repository from an existing directory tree.

Expected outcome. `svn` should report `A firstfile` and `Updated to revision 1.`. Check that the contents of the file are correct.

Caveats. In order to do the update command, you have to be in a checked-out copy of the repository. Do `svn info` to make sure that you are in the right place.

Exercise. Let both students create a new directory with a few files. Declare the directory and commit it. Do `svn update` to obtain the changes the other mde.

Expected outcome. You can do `svn add` on the directory, this will also add the files contained in it.

Caveats. Do not forget the `commit`.

In order for svn to keep track of your files, you should never do `cp` or `mv` on files that are in the repository. Instead, do `svn cp` or `svn mv`. Likewise, there are commands `svn rm` and `svn mkdir`.

5.2.3 Conflicts

Purpose. In this section you will learn about how do deal with conflicting edits by two users of the same repository.

Now let's see what happens when two people edit the same file. Let both students make an edit to `firstfile`, but one to the top, the other to the bottom. After one student commits the edit, the other will see

```
%% emacs firstfile # make some change
%% svn commit -m "another edit to the first file"
Sending        firstfile
svn: Commit failed (details follow):
svn: Out of date: 'firstfile' in transaction '5-1'
```

The solution is to get the other edit, and commit again. After the update, svn reports that it has resolved a conflict successfully.

```
%% svn update
G  firstfile
Updated to revision 5.
%% svn commit -m "another edit to the first file"
Sending        firstfile
Transmitting file data .
Committed revision 6.
```

The `G` at the start of the line indicates that svn has resolved a conflicting edit.

If both students make edits on the same part of the file, svn can no longer resolve the conflicts. For instance, let one student insert a line between the first and the second, and let the second student edit the second line. Whoever tries to commit second, will get messages like this:

```
%% svn commit -m "another edit to the first file"
svn: Commit failed (details follow):
svn: Aborting commit: '/share/home/12345/yourname/myproject/firstfile'
remains in conflict
%% svn update
C  firstfile
Updated to revision 7.
```

Subversion will give you several options. For instance, you can type `e` to open the file in an editor. You can also type `p` for 'postpone' and edit it later. Opening the file in an editor, it will look like

```
aa
<<<<<<< .mine
bb
=======
123
b
>>>>>>> .r7
cc
```

indicating the difference between the local version ('mine') and the remote. You need to edit the file to resolve the conflict.

After this, you tell svn that the conflict was resolved, and you can commit:

```
%% svn resolved firstfile
Resolved conflicted state of 'firstfile'
%% svn commit -m "another edit to the first file"
Sending        firstfile
Transmitting file data .
Committed revision 8.
```

The other student then needs to do another update to get the correction.

5.2.4 Inspecting the history

Purpose. In this section, you will learn how to get information about the repository.

You've already seen `svn info` as a way of getting information about the repository. To get the history, do `svn log` to get all log messages, or `svn log 2:5` to get a range.

To see differences in various revisions of individual files, use `svn diff`. First do `svn commit` and `svn update` to make sure you are up to date. Now do `svn diff firstfile`. No output, right? Now make an edit in `firstfile` and do `svn diff firstfile` again. This gives you the difference between the last commited version and the working copy.

You can also ask for differences between committed versions with `svn diff -r 4:6 firstfile`.

The output of this diff command is a bit cryptic, but you can understand it without too much trouble. There are also fancy GUI implementations of svn for every platform that show you differences in a much nicer way.

If you simply want to see what a file used to look like, do `svn cat -r 2 firstfile`. To get a copy of a certain revision of the repository, do `svn export -r 3 . ../rev3`, which exports the repository at the current directory ('dot') to the directory `../rev3`.

If you save the output of `svn diff`, it is possible to apply it with the Unix `patch` command. This is a quick way to send patches to someone without them needing to check out the repository.

5.2.5 Shuffling files around

We now realize that we really wanted all these files in a subdirectory in the repository. First we create the directory, putting it under svn control:

```
%% svn mkdir trunk
A         trunk
```

Then we move all files there, again prefixing all comands with `svn`:

```
%% for f in firstfile otherfile myfile mysecondfile ; do \
        svn mv $f trunk/ ; done
A         trunk/firstfile
D         firstfile
A         trunk/otherfile
D         otherfile
A         trunk/myfile
D         myfile
A         trunk/mysecondfile
D         mysecondfile
```

Finally, we commit these changes:

```
%% svn commit -m "trunk created"
Deleting       firstfile
Adding         trunk/firstfile
Deleting       myfile
Deleting       mysecondfile
Deleting       otherfile
Adding         trunk
Adding         trunk/myfile
Adding         trunk/mysecondfile
Adding         trunk/otherfile
```

You probably now have picked up on the message that you always use `svn` to do file manipulations. Let's pretend this has slipped your mind.

Exercise. Create a file `somefile` and commit it to the repository. Then do `rm somefile`, thereby deleting a file without `svn` knowing about it. What is the output of `svn status`?

Expected outcome. svn indicates with an exclamation point that the file has disappeared.

You can fix this situation in a number of ways:

- `svn revert` restores the file to the state in which it was last restored. For a deleted file, this means that it is brought back into existence from the repository. This command is also useful to undo any local edits, if you change your mind about something.
- `svn rm firstfile` is the official way to delete a file. You can do this even if you have already deleted the file outside `svn`.
- Sometimes `svn` will get confused about your attempts to delete a file. You can then do `svn --force rm yourfile`.

5.2.6 Branching and merging

Suppose you want to tinker with the repository, while still staying up to date with changes that other people make.

```
%% svn copy \
  <the URL of your repository>/trunk \
  <the URL of your repository>/onebranch \
  -m "create a branch"

Committed revision 11.
```

You can check this out as before:

```
%% svn co <the URL of your repository>/onebranch \
          projectbranch
A  projectbranch/mysecondfile
A  projectbranch/otherfile
A  projectbranch/myfile
A  projectbranch/firstfile
Checked out revision 11.
```

Now, if you make edits in this branch, they will not be visible in the trunk:

```
%% emacs firstfile # do some edits here
%% svn commit -m "a change in the branch"
Sending        firstfile
Transmitting file data .
Committed revision 13.
```

```
%% (cd ../myproject/trunk/ ; svn update )
At revision 13.
```

On the other hand, edits in the main trunk can be pulled into this branch:

```
%% svn merge ^/trunk
--- Merging r13 through r15 into '.':
U    secondfile
```

When you are done editing, the branch edits can be added back to the trunk. For this, it is best to have a clean checkout of the branch:

```
%% svn co  file://`pwd`/repository/trunk mycleanproject
A    # all the current files
```

and then do a special merge:

```
%% cd mycleanproject
%% svn merge --reintegrate ^/branch
--- Merging differences between repository URLs into '.':
U    firstfile
 U   .
%% svn info
Path: .
URL: <the URL of your repository>/trunk
Repository Root: <the URL of your repository>
Repository UUID: dc38b821-b9c6-4a1a-a194-a894fba1d7e7
Revision: 16
Node Kind: directory
Schedule: normal
Last Changed Author: build
Last Changed Rev: 14
Last Changed Date: 2009-05-18 13:34:55 -0500 (Mon, 18 May 2009)
%% svn commit -m "trunk updates from the branch"
Sending        .
Sending        firstfile
Transmitting file data .
Committed revision 17.
```

5.2.7 Repository browsers

The `svn` command can give you some amount of information about a repository, but there are graphical tools that are easier to use and that give a better overview of a repository. For the common platforms, several such tools exist, free or commercial. Here is a browser based tool: http://wwww.websvn.info.

5.3 Mercurial (hg) and Git

Mercurial and *git* are the best known of a new generation of *distributed source code control* systems. Many commands are the same as for subversion, but there are some new ones, corresponding to the new level of sophistication. Mercurial and git share some commands, but there are also differences. Git is ultimately more powerful, but mercurial is easier to use at first.

Here is a translation between the two systems: https://github.com/sympy/sympy/wiki/Git-hg-rosetta-s

This lab should be done two people, to simulate a group of programmers working on a joint project. You can also do this on your own by using two clones of the repository, preferably opening two windows on your computer.

5.3.1 Create and populate a repository

Purpose. In this section you will create a repository and make a local copy to work on.

First we need to have a repository. In practice, you will often use one that has been previousy set up, but there are several ways to set up a repository yourself. There are commercial and free hosting services such as http://bitbucket.org. (Academic users can have more private repositories.)

Let's assume that one student has created a repository `your-project` on Bitbucket. Both students can then clone it:

```
%% hg clone https://YourName@bitbucket.org/YourName/your-project
updating to branch default
0 files updated, 0 files merged, 0 files removed,
    0 files unresolved
```

or

```
%% git clone git@bitbucket.org:YourName/yourproject.git
Cloning into 'yourproject'...
warning: You appear to have cloned an empty repository.
```

You now have an empty directory `your-project`.

Exercise. Go into the project directory and see if it is really empty.

Expected outcome. There is a hidden directory `.hg` or `.git`

5.3.2 New files

Creating an untracked file

Purpose. In this section you will make some simple changes: creating a new file and editing an existing file

One student now makes a file to add to the repository:

```
%%  cat > firstfile
a
b
c
d
e
f
^D
```

(where `^D` stands for control-D, which terminates the input.) This file is unknown to `hg`:

```
%% hg status
? firstfile
```

Git is a little more verbose:

```
git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

firstfile

nothing added to commit but untracked files present
  (use "git add" to track)
```

Adding the file to the repository We need to declare the file as belonging to the repository; a subsequent `hg commit` command then copies it into the repository.

```
%% hg add firstfile
%% hg status
A firstfile
%% hg commit -m "made a first file"
```

or

```
%% git add firstfile
%% git status
On branch master
```

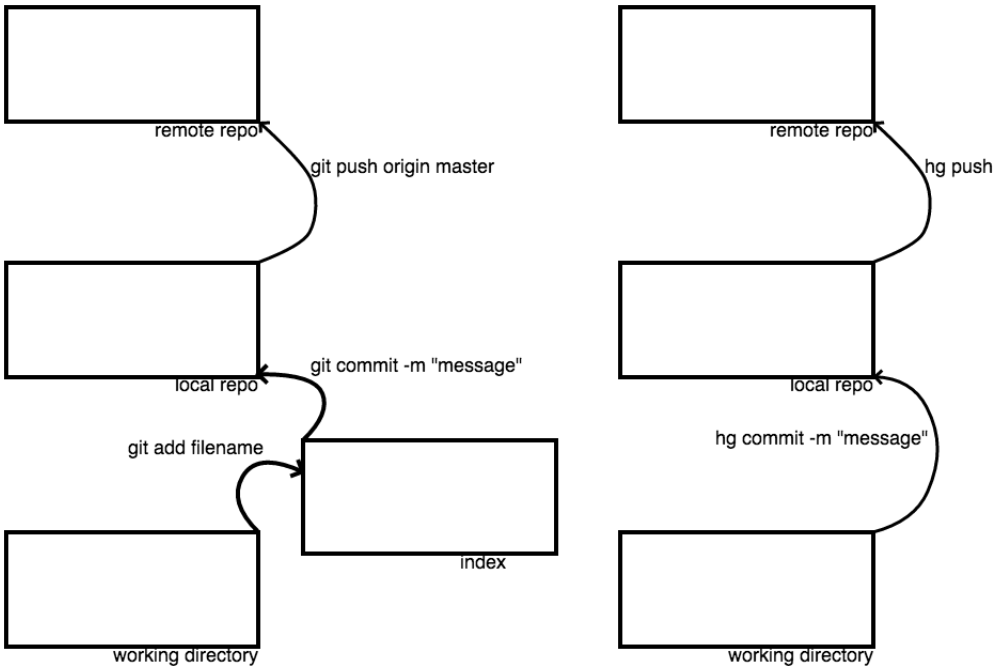

Figure 5.3: Add local changes to the remote repository

```
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   firstfile
%% git commit -a -m "adding a first file"
  [master (root-commit) f4b738c] adding a first file
  1 file changed, 5 insertions(+)
  create mode 100644 firstfile
```

Unlike with Subversion, the file has now only been copied into the local repository, so that you can, for instance, roll back your changes. If you want this file added to the master repository, you need the `hg push` command:

```
%% hg push https://YourName@bitbucket.org/YourName/your-project
pushing to https://YourName@bitbucket.org/YourName/your-project
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files
```

```
remote: bb/acl: YourName is allowed. accepted payload.
```

In the `push` step you were probably asked for your password. You can prevent that by having some lines in your `$HOME/.hgrc` file:

```
[paths]
projectrepo = https://YourName:yourpassword@bitbucket.org/YourName/my-project
[ui]
username=Your Name <you@somewhere.youruniversity.edu>
```

Now the command `hg push projectrepo` will push the local changes to the global repository without asking for your password. Of course, now you have a file with a cleartext password, so you should set the permissions of this file correctly.

With git you need to be more explicit, since the ties between your local copy and the 'upstream' repository can be more fluid.

```
git remote add origin git@bitbucket.org:YourName/yourrepo.git
git push origin master
```

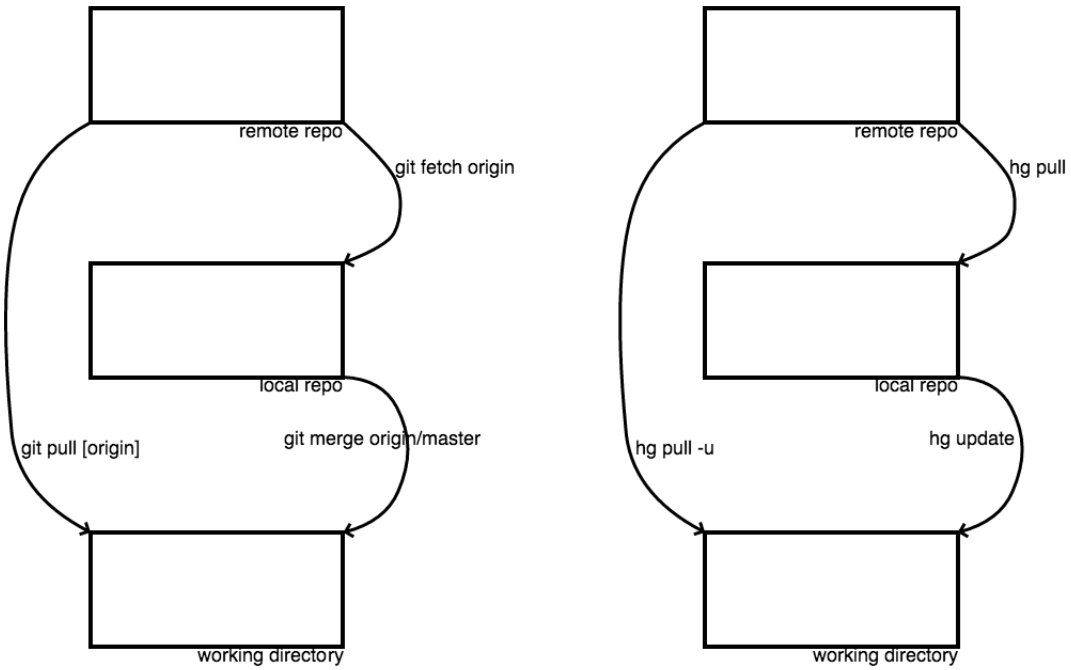

Figure 5.4: Get changes that were made to the remote repository

The second student now needs to update their repository. Just like the upload took two commands, this pass also takes two. First you do `hg pull` to update your local repository. This does not update the local files you have: for that you need to do `hg update`.

Exercise. Do this and check that the contents of the file are correct.

Expected outcome. In order to do the update command, you have to be in a checked-out copy of the repository.

Caveats.

Exercise. Let both students create a new directory with a few files. Declare the directory and commit it. Pull and update to obtain the changes the other mde.

Expected outcome. You can do `hg add` on the directory, this will also add the files contained in it.

In order for Mercurial to keep track of your files, you should never do the shell commands `cp` or `mv` on files that are in the repository. Instead, do `hg cp` or `hg mv`. Likewise, there is a command `hg rm`.

5.3.3 Conflicts

Purpose. In this section you will learn about how do deal with conflicting edits by two users of the same repository.

Now let's see what happens when two people edit the same file. Let both students make an edit to `firstfile`, but one to the top, the other to the bottom. After one student commits the edit, the other can commit changes, after all, these only affect the local repository. However, trying to push that change gives an error:

```
%% emacs firstfile # make some change
%% hg commit -m ''first again''
%% hg push test
abort: push creates new remote head b0d31ea209b3!
(you should pull and merge or use push -f to force)
```

The solution is to get the other edit, and commit again. This takes a couple of commands:

```
%% hg pull myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)

%% hg merge
merging firstfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)

%% hg status
M firstfile
%% hg commit -m ''my edit again''
%% hg push test
pushing to https://VictorEijkhout:***@bitbucket.org/
```

```
                        VictorEijkhout/my-project
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 2 changesets with 2 changes to 1 files
remote: bb/acl: VictorEijkhout is allowed. accepted payload.
```

This may seem complicated, but you see that Mercurial prompts you for what commands to execute, and the workflow is clear, if you refer to figure 5.2.

Exercise. Do a `cat` on the file that both of you have been editing. You should find that both edits are incorporated. That is the 'merge' that Mercurial referred to.

If both students make edits on the same part of the file, Mercurial can no longer resolve the conflicts. For instance, let one student insert a line between the first and the second, and let the second student edit the second line. Whoever tries to push second, will get messages like this:

```
%% hg pull test
added 3 changesets with 3 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
%% hg merge
merging firstfile
warning: conflicts during merge.
merging firstfile incomplete!
    (edit conflicts, then use 'hg resolve --mark')
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
use 'hg resolve' to retry unresolved file merges
    or 'hg update -C .' to abandon
```

Mercurial will give you several options. It is easiest to resolve the conflict with a text editor. If you open the file that has the conflict you'll see something like:

```
<<<<<<< local
aa
bbbb
=======
aaa
a2
b
>>>>>>> other
c
```

indicating the difference between the local version ('mine') and the other, that is the version that you pulled and tried to merge. You need to edit the file to resolve the conflict.

After this, you tell hg that the conflict was resolved:

```
 hg resolve --mark
%% hg status
M firstfile
? firstfile.orig
```

After this you can commit and push again. The other student then needs to do another update to get the correction.

Not all files can be merged: for binary files Mercurial will ask you:

```
%% hg merge
merging proposal.tex
merging summary.tex
merking references.tex
 no tool found to merge proposal.pdf
keep (l)ocal or take (o)ther? o
```

This means that the only choices are to keep your local version (type `l` and hit return) or take the other version (type `o` and hit return). In the case of a binary file that was obvious generated automatically, some people argue that they should not be in the repository to begin with.

5.3.4 Inspecting the history

Purpose. In this section, you will learn how to get information about the repository.

If you want to know where you cloned a repository from, look in the file `.hg/hgrc`.

The main sources of information about the repository are `hg log` and `hg id`. The latter gives you global information, depending on what option you use. For instance, `hg id -n` gives the local revision number.

`hg log` gives you a list of all changesets so far, with the comments you entered.
`hg log -v` tells you what files were affected in each changeset.
`hg log -r 5` or `hg log -r 6:8` gives information on one or more changesets.

To see differences in various revisions of individual files, use `hg diff`. First make sure you are up to date. Now do `hg diff firstfile`. No output, right? Now make an edit in `firstfile` and do `hg diff firstfile` again. This gives you the difference between the last commited version and the working copy.

Check for yourself what happens when you do a commit but no push, and you issue the above diff command.

You can also ask for differences between committed versions with `hg diff -r 4:6 firstfile`.

The output of this diff command is a bit cryptic, but you can understand it without too much trouble. There are also fancy GUI implementations of hg for every platform that show you differences in a much nicer way.

If you simply want to see what a file used to look like, do `hg cat -r 2 firstfile`. To get a copy of a certain revision of the repository, do `hg export -r 3 . ../rev3`, which exports the repository at the current directory ('dot') to the directory `../rev3`.

If you save the output of `hg diff`, it is possible to apply it with the Unix `patch` command. This is a quick way to send patches to someone without them needing to check out the repository.

Chapter 6

Scientific Data Storage

There are many ways of storing data, in particular data that comes in arrays. A surprising number of people stores data in spreadsheets, then exports them to ascii files with comma or tab delimiters, and expects other people (or other programs written by themselves) to read that in again. Such a process is wasteful in several respects:

- The ascii representation of a number takes up much more space than the internal binary representation. Ideally, you would want a file to be as compact as the representation in memory.
- Conversion to and from ascii is slow; it may also lead to loss of precision.

For such reasons, it is desirable to have a file format that is based on binary storage. There are a few more requirements on a useful file format:

- Since binary storage can differ between platforms, a good file format is platform-independent. This will, for instance, prevent the confusion between *big-endian* and *little-endian* storage, as well as conventions of 32 versus 64 bit floating point numbers.
- Application data can be heterogeneous, comprising integer, character, and floating point data. Ideally, all this data should be stored together.
- Application data is also structured. This structure should be reflected in the stored form.
- It is desirable for a file format to be *self-documenting*. If you store a matrix and a right-hand side vector in a file, wouldn't it be nice if the file itself told you which of the stored numbers are the matrix, which the vector, and what the sizes of the objects are?

This tutorial will introduce the HDF5 library, which fulfills these requirements. HDF5 is a large and complicated library, so this tutorial will only touch on the basics. For further information, consult `http://www.hdfgroup.org/HDF5/`. While you do this tutorial, keep your browser open on `http://www.hdfgroup.org/HDF5/doc/` or `http://www.hdfgroup.org/HDF5/RM/RM_H5Front.html` for the exact syntax of the routines.

6.1 Introduction to HDF5

As described above, HDF5 is a file format that is machine-independent and self-documenting. Each HDF5 file is set up like a directory tree, with subdirectories, and leaf nodes which contain the actual data. This means that data can be found in a file by referring to its name, rather than its location in the file. In this

section you will learn to write programs that write to and read from HDF5 files. In order to check that the files are as you intend, you can use the `h5dump` utility on the command line.[1]

Just a word about compatibility. The HDF5 format is not compatible with the older version HDF4, which is no longer under development. You can still come across people using hdf4 for historic reasons. This tutorial is based on HDF5 version 1.6. Some interfaces changed in the current version 1.8; in order to use 1.6 APIs with 1.8 software, add a flag `-DH5_USE_16_API` to your compile line.

Many HDF5 routines are about creating objects: file handles, members in a dataset, et cetera. The general syntax for that is

```
hid_t h_id;
h_id = H5Xsomething(...);
```

Failure to create the object is indicated by a negative return parameter, so it would be a good idea to create a file `myh5defs.h` containing:

```
#include "hdf5.h"
#define H5REPORT(e) \
  {if (e<0) {printf("\nHDF5 error on line %d\n\n",__LINE__); \
   return e;}}
```

and use this as:

```
#include "myh5defs.h"

hid_t h_id;
h_id = H5Xsomething(...); H5REPORT(h_id);
```

6.2 Creating a file

First of all, we need to create an HDF5 file.

```
hid_t file_id;
herr_t status;

file_id = H5Fcreate( filename, ... );
    ...
status = H5Fclose(file_id);
```

This file will be the container for a number of data items, organized like a directory tree.

Exercise. Create an HDF5 file by compiling and running the `create.c` example below.

1. In order to do the examples, the `h5dump` utility needs to be in your path, and you need to know the location of the `hdf5.h` and `libhdf5.a` and related library files.

Expected outcome. A file `file.h5` should be created.

Caveats. Be sure to add HDF5 include and library directories:
`cc -c create.c -I. -I/opt/local/include`
and
`cc -o create create.o -L/opt/local/lib -lhdf5`. The include and lib directories will be system dependent.

On the TACC clusters, do `module load hdf5`, which will give you environment variables `TACC_HDF5_INC` and `TACC_HDF5_LIB` for the include and library directories, respectively.

```
/*
 * File: create.c
 * Author: Victor Eijkhout
 */
#include "myh5defs.h"
#define FILE "file.h5"

main() {

   hid_t       file_id;   /* file identifier */
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
   H5REPORT(file_id);

   /* Terminate access to the file. */
   status = H5Fclose(file_id);
}
```

You can display the created file on the commandline:

```
%% h5dump file.h5
HDF5 "file.h5" {
GROUP "/" {
}
}
```

Note that an empty file corresponds to just the root of the directory tree that will hold the data.

6.3 Datasets

Next we create a dataset, in this example a 2D grid. To describe this, we first need to construct a dataspace:

```
dims[0] = 4; dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);
dataset_id = H5Dcreate(file_id, "/dset", dataspace_id, .... );
```

```
....
status = H5Dclose(dataset_id);
status = H5Sclose(dataspace_id);
```

Note that datasets and dataspaces need to be closed, just like files.

Exercise. Create a dataset by compiling and running the `dataset.c` code below

Expected outcome. This creates a file `dset.h` that can be displayed with `h5dump`.

```
/*
 * File: dataset.c
 * Author: Victor Eijkhout
 */
#include "myh5defs.h"
#define FILE "dset.h5"

main() {

   hid_t       file_id, dataset_id, dataspace_id;  /* identifiers */
   hsize_t     dims[2];
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the data space for the dataset. */
   dims[0] = 4;
   dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the dataset. */
   dataset_id = H5Dcreate(file_id, "/dset", H5T_NATIVE_INT,
                          dataspace_id, H5P_DEFAULT);
   /*H5T_STD_I32BE*/

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset_id);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace_id);

   /* Close the file. */
   status = H5Fclose(file_id);
}
```

We again view the created file online:

```
%% h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_STD_I32BE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 0, 0, 0, 0, 0, 0,
      (1,0): 0, 0, 0, 0, 0, 0,
      (2,0): 0, 0, 0, 0, 0, 0,
      (3,0): 0, 0, 0, 0, 0, 0
      }
   }
}
}
```

The datafile contains such information as the size of the arrays you store. Still, you may want to add related scalar information. For instance, if the array is output of a program, you could record with what input parameter was it generated.

```
parmspace = H5Screate(H5S_SCALAR);
parm_id = H5Dcreate
  (file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);
```

Exercise. Add a scalar dataspace to the HDF5 file, by compiling and running the `parmwrite.c` code below.

Expected outcome. A new file `wdset.h5` is created.

```
/*
 * File: parmdataset.c
 * Author: Victor Eijkhout
 */
#include "myh5defs.h"
#define FILE "pdset.h5"

main() {

   hid_t       file_id, dataset_id, dataspace_id;  /* identifiers */
   hid_t       parm_id,parmspace;
   hsize_t     dims[2];
   herr_t      status;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the data space for the dataset. */
```

```
   dims[0] = 4;
   dims[1] = 6;
   dataspace_id = H5Screate_simple(2, dims, NULL);

   /* Create the dataset. */
   dataset_id = H5Dcreate
     (file_id, "/dset", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

   /* Add a descriptive parameter */
   parmspace = H5Screate(H5S_SCALAR);
   parm_id = H5Dcreate
     (file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset_id);
   status = H5Dclose(parm_id);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace_id);
   status = H5Sclose(parmspace);

   /* Close the file. */
   status = H5Fclose(file_id);
}
```

```
%% h5dump wdset.h5
HDF5 "wdset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5,
      (1,0): 6.5, 7.5, 8.5, 9.5, 10.5, 11.5,
      (2,0): 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
      (3,0): 18.5, 19.5, 20.5, 21.5, 22.5, 23.5
      }
   }
   DATASET "parm" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SCALAR
      DATA {
      (0): 37
      }
   }
```

```
}
}
```

6.4 Writing the data

The datasets you created allocate the space in the hdf5 file. Now you need to put actual data in it. This is done with the `H5Dwrite` call.

```
/* Write floating point data */
for (i=0; i<24; i++) data[i] = i+.5;
status = H5Dwrite
  (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
   data);
/* write parameter value */
parm = 37;
status = H5Dwrite
  (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
   &parm);
```

```
/*
 * File: parmwrite.c
 * Author: Victor Eijkhout
 */
#include "myh5defs.h"
#define FILE "wdset.h5"

main() {

   hid_t       file_id, dataset, dataspace;  /* identifiers */
   hid_t       parmset,parmspace;
   hsize_t     dims[2];
   herr_t      status;
   double data[24]; int i,parm;

   /* Create a new file using default properties. */
   file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

   /* Create the dataset. */
   dims[0] = 4; dims[1] = 6;
   dataspace = H5Screate_simple(2, dims, NULL);
   dataset = H5Dcreate
     (file_id, "/dset", H5T_NATIVE_DOUBLE, dataspace, H5P_DEFAULT);

   /* Add a descriptive parameter */
   parmspace = H5Screate(H5S_SCALAR);
   parmset = H5Dcreate
     (file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);
```

```
   /* Write data to file */
   for (i=0; i<24; i++) data[i] = i+.5;
   status = H5Dwrite
     (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      data); H5REPORT(status);

   /* write parameter value */
   parm = 37;
   status = H5Dwrite
     (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
      &parm); H5REPORT(status);

   /* End access to the dataset and release resources used by it. */
   status = H5Dclose(dataset);
   status = H5Dclose(parmset);

   /* Terminate access to the data space. */
   status = H5Sclose(dataspace);
   status = H5Sclose(parmspace);

   /* Close the file. */
   status = H5Fclose(file_id);
}



%% h5dump wdset.h5
HDF5 "wdset.h5" {
GROUP "/" {
   DATASET "dset" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
      DATA {
      (0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5,
      (1,0): 6.5, 7.5, 8.5, 9.5, 10.5, 11.5,
      (2,0): 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
      (3,0): 18.5, 19.5, 20.5, 21.5, 22.5, 23.5
      }
   }
   DATASET "parm" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SCALAR
      DATA {
      (0): 37
      }
   }
}
}
```

If you look closely at the source and the dump, you see that the data types are declared as 'native', but rendered as `LE`. The 'native' declaration makes the datatypes behave like the built-in C or Fortran data types. Alternatively, you can explicitly indicate whether data is *little-endian* or *big-endian*. These terms describe how the bytes of a data item are ordered in memory. Most architectures use little endian, as you can see in the dump output, but, notably, IBM uses big endian.

6.5 Reading

Now that we have a file with some data, we can do the mirror part of the story: reading from that file. The essential commands are

```
h5file = H5Fopen( .... )
....
H5Dread( dataset, .... data .... )
```

where the `H5Dread` command has the same arguments as the corresponding `H5Dwrite`.

Exercise. Read data from the `wdset.h5` file that you create in the previous exercise, by compiling and running the `allread.c` example below.

Expected outcome. Running the `allread` executable will print the value `37` of the parameter, and the value `8.5` of the `(1,2)` data point of the array.

Caveats. Make sure that you run `parmwrite` to create the input file.

```
/*
 * File: allread.c
 * Author: Victor Eijkhout
 */
#include "myh5defs.h"
#define FILE "wdset.h5"

main() {

   hid_t       file_id, dataset, parmset;
   herr_t      status;
   double data[24]; int parm;

   /* Open an existing file */
   file_id = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
   H5REPORT(file_id);

   /* Locate the datasets. */
   dataset = H5Dopen(file_id, "/dset"); H5REPORT(dataset);
   parmset = H5Dopen(file_id,"/parm");  H5REPORT(parmset);

   /* Read data back */
   status = H5Dread
```

```
      (parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,
       &parm); H5REPORT(status);
    printf("parameter value: %d\n",parm);

    status = H5Dread
      (dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,
       data); H5REPORT(status);
    printf("arbitrary data point [1,2]: %e\n",data[1*6+2]);

    /* Terminate access to the datasets */
    status = H5Dclose(dataset); H5REPORT(status);
    status = H5Dclose(parmset); H5REPORT(status);

    /* Close the file. */
    status = H5Fclose(file_id);
}
```

```
 %% ./allread
parameter value: 37
arbitrary data point [1,2]: 8.500000e+00
```

Chapter 7

Plotting with GNUplot

The *gnuplot* utility is a simple program for plotting sets of points or curves. This very short tutorial will show you some of the basics. For more commands and options, see the manual http://www.gnuplot.info/docs/gnuplot.html.

7.1 Usage modes

The two modes for running `gnuplot` are *interactive* and *from file*. In interactive mode, you call `gnuplot` from the command line, type commands, and watch output appear; you terminate an interactive session with `quit`. If you want to save the results of an interactive session, do `save "name.plt"`. This file can be edited, and loaded with `load "name.plt"`.

Plotting non-interactively, you call `gnuplot <your file>`.

The output of `gnuplot` can be a picture on your screen, or drawing instructions in a file. Where the output goes depends on the setting of the *terminal*. By default, `gnuplot` will try to draw a picture. This is equivalent to declaring

```
set terminal x11
```

or `aqua`, `windows`, or any choice of graphics hardware.

For output to file, declare

```
set terminal pdf
```

or `fig`, `latex`, `pbm`, et cetera. Note that this will only cause the pdf commands to be written to your screen: you need to direct them to file with

```
set output "myplot.pdf"
```

or capture them with

```
gnuplot my.plt > myplot.pdf
```

7.2 Plotting

The basic plot commands are `plot` for 2D, and `splot` ('surface plot') for 3D plotting.

7.2.1 Plotting curves

By specifying

```
plot x**2
```

you get a plot of $f(x) = x^2$; `gnuplot` will decide on the range for x. With

```
set xrange [0:1]
plot 1-x title "down", x**2 title "up"
```

you get two graphs in one plot, with the x range limited to $[0, 1]$, and the appropriate legends for the graphs. The variable `x` is the default for plotting functions.

Plotting one function against another – or equivalently, plotting a parametric curve – goes like this:

```
set parametric
plot [t=0:1.57] cos(t),sin(t)
```

which gives a quarter circle.

To get more than one graph in a plot, use the command `set multiplot`.

7.2.2 Plotting data points

It is also possible to plot curves based on data points. The basic syntax is `plot 'datafile'`, which takes two columns from the data file and interprets them as (x, y) coordinates. Since data files can often have multiple columns of data, the common syntax is `plot 'datafile' using 3:6` for columns 3 and 6. Further qualifiers like `with lines` indicate how points are to be connected.

Similarly, `splot "datafile3d.dat" 2:5:7` will interpret three columns as specifying (x, y, z) coordinates for a 3D plot.

If a data file is to be interpreted as level or height values on a rectangular grid, do `splot "matrix.dat" matrix` for data points; connect them with

```
split "matrix.dat" matrix with lines
```

7.2.3 Customization

Plots can be customized in many ways. Some of these customizations use the `set` command. For instance,

```
set xlabel "time"
set ylabel "output"
set title "Power curve"
```

You can also change the default drawing style with

```
set style function dots
```

(`dots`, `lines`, `dots`, `points`, et cetera), or change on a single plot with

```
plot f(x) with points
```

7.3 Workflow

Imagine that your code produces a dataset that you want to plot, and you run your code for a number of inputs. It would be nice if the plotting can be automated. Gnuplot itself does not have the facilities for this, but with a little help from shell programming this is not hard to do.

Suppose you have data files

```
data1.dat data2.dat data3.dat
```

and you want to plot them with the same gnuplot commands. You could make a file `plot.template`:

```
set term pdf
set output "FILENAME.pdf"
plot "FILENAME.dat"
```

The string `FILENAME` can be replaced by the actual file names using, for instance `sed`:

```
for d in data1 data2 data3 ; do
  cat plot.template | sed s/FILENAME/$d/ > plot.cmd
  gnuplot plot.cmd
done
```

Variations on this basic idea are many.

Chapter 8

Good coding practices

Sooner or later, and probably sooner than later, every programmer is confronted with code not behaving as intended. In this section you will learn some techniques of dealing with this problem. At first we will see a number of techniques for *preventing* errors; in the next chapter we will discuss debugging, the process of finding the inevitable errors in a program, once they have occurred.

8.1 Defensive programming

In this section we will discuss a number of techniques that are aimed at preventing the likelihood of programming errors, or increasing the likelikhood of them being found at runtime. We call this *defensive programming*.

Scientific codes are often large and involved, so it is a good practice to code knowing that you are going to make mistakes and prepare for them. Another good coding practice is the use of tools: there is no point in reinventing the wheel if someone has already done it for you. Some of these tools are be described in other sections:

- Build systems, such as Make, Scons, Bjam; see section 4.
- Source code management with SVN, Git; see section 5.
- Regression testing and designing with testing in mind (unit testing)

First we will have a look at runtime sanity checks, where you test for things that can not or should not happen.

8.1.1 Assertions

In the things that can go wrong with a program we can distinguish between errors and bugs. Errors are things that legitimately happen but that should not. File systems are common sources of errors: a program wants to open a file but the file doesn't exist because the user mistyped the name, or the program writes to a file but the disk is full. Other errors can come from arithmetic, such as *overflow* errors.

On the other hand, a *bug* in a program is an occurence that cannot legitimately occur. Of course, 'legitimately' here means 'according to the programmer's intentions'. Bugs can often be described as 'the computer always does what you ask, not necessarily what you want'.

Assertions serve to detect bugs in your program: an *assertion* is a predicate that should be true at a certain point in your program. Thus, an assertion failing means that you didn't code what you intended to code. An assertion is typically a statement in your programming language, or a preprocessor macro; upon failure of the assertion, your program will take some abortive action.

Some examples of assertions:

- If a subprogram has an array argument, it is a good idea to test whether the actual argument is a null pointer before indexing into the array.
- Similarly, you could test a dynamically allocated data structure for not having a null pointer.
- If you calculate a numerical result for which certain mathematical properties hold, for instance you are writing a sine function, for which the result has to be in $[-1, 1]$, you should test whether this property indeed holds for the result.

Assertions are often disabled in a program once it's sufficiently tested. The reason for this is that assertions can be expensive to execute. For instance, if you have a complicated data structure, you could write a complicated integrity test, and perform that test in an assertion, which you put after every access to the data structure.

Because assertions are often disabled in the 'production' version of a code, they should not affect any stored data . If they do, your code may behave differently when you're testing it with assertions, versus how you use it in practice without them. This is also formulated as 'assertions should not have *side-effects*'.

8.1.1.1 The C `assert` macro

The C standard library has a file `assert.h` which provides an `assert()` macro. Inserting `assert(foo)` has the following effect: if `foo` is zero (false), a diagnostic message is printed on standard error:

```
Assertion failed: foo, file filename, line line-number
```

which includes the literal text of the expression, the file name, and line number; and the program is subsequently aborted. Here is an example:

```
#include<assert.h>

void open_record(char *record_name)
{
  assert(record_name!=NULL);
  /* Rest of code */
}

int main(void)
{
  open_record(NULL);
}
```

The `assert` macro can be disabled by defining the `NDEBUG` macro.

8.1.1.2 *An assert macro for Fortran*

(Thanks to Robert Mclay for this code.)

```
#if (defined( GFORTRAN ) || defined( G95 ) || defined ( PGI) )
# define MKSTR(x) "x"
#else
# define MKSTR(x) #x
#endif
#ifndef NDEBUG
# define ASSERT(x, msg) if (.not. (x) ) \
                call assert( FILE , LINE ,MKSTR(x),msg)
#else
# define ASSERT(x, msg)
#endif
subroutine assert(file, ln, testStr, msgIn)
implicit none
character(*) :: file, testStr, msgIn
integer :: ln
print *, "Assert: ",trim(testStr)," Failed at ",trim(file),":",ln
print *, "Msg:", trim(msgIn)
stop
end subroutine assert
```

which is used as

```
ASSERT(nItemsSet.gt.arraySize,"Too many elements set")
```

8.1.2 Try-catch in C++

8.1.3 Use of error codes

In some software libraries (for instance MPI or PETSc) every subprogram returns a result, either the function value or a parameter, to indicate success or failure of the routine. It is good programming practice to check these error parameters, even if you think that nothing can possibly go wrong.

It is also a good idea to write your own subprograms in such a way that they always have an error parameter. Let us consider the case of a function that performs some numerical computation.

```
float compute(float val)
{
  float result;
  result = ... /* some computation */
  return result;
}
```

```
float value,result;
result = compute(value);
```

Looks good? What if the computation can fail, for instance:

```
  result = ... sqrt(val) ... /* some computation */
```

How do we handle the case where the user passes a negative number?

```
float compute(float val)
{
  float result;
  if (val<0) { /* then what? */
  } else
    result = ... sqrt(val) ... /* some computation */
  return result;
}
```

We could print an error message and deliver some result, but the message may go unnoticed, and the calling environment does not really receive any notification that something has gone wrong.

The following approach is more flexible:

```
int compute(float val,float *result)
{
  float result;
  if (val<0) {
    return -1;
  } else {
    *result = ... sqrt(val) ... /* some computation */
  }
  return 0;
}

float value,result; int ierr;
ierr = compute(value,&result);
if (ierr!=0) { /* take appropriate action */
}
```

You can save yourself a lot of typing by writing

```
#define CHECK_FOR_ERROR(ierr) \
  if (ierr!=0) { \
    printf("Error %d detected\n",ierr); \
    return -1 ; }
....
```

```
ierr = compute(value,&result); CHECK_FOR_ERROR(ierr);
```

Using some cpp macros you can even define

```
#define CHECK_FOR_ERROR(ierr) \
  if (ierr!=0) { \
    printf("Error %d detected in line %d of file %s\n",\
           ierr,__LINE__,__FILE__); \
    return -1 ; }
```

Note that this macro not only prints an error message, but also does a further return. This means that, if you adopt this use of error codes systematically, you will get a full backtrace of the calling tree if an error occurs. (In the Python language this is precisely the wrong approach since the backtrace is built-in.)

8.2 Guarding against memory errors

In scientific computing it goes pretty much without saying that you will be working with large amounts of data. Some programming languages make managing data easy, others, one might say, make making errors with data easy.

The following are some examples of *memory violations*.

- Writing outside array bounds. If the address is outside the user memory, your code may abort with an error such as *segmentation violation*, and the error is reasonably easy to find. If the address is just outside an array, it will corrupt data but not crash the program; such an error may go undetected for a long time, as it can have no effect, or only introduce subtly wrong values in your computation.
- Reading outside array bounds can be harder to find than errors in writing, as it will often not abort your code, but only introduce wrong values.
- The use of uninitialized memory is similar to reading outside array bounds, and can go undetected for a long time. One variant of this is through attaching memory to an unallocated pointer.
  This particular kind of error can manifest itself in interesting behaviour. Let's say you notice that your program misbehaves, you recompile it with debug mode to find the error, and now the error no longer occurs. This is probably due to the effect that, with low optimization levels, all allocated arrays are filled with zeros. Therefore, your code was originally reading a random value, but is now getting a zero.

This section contains some techniques to prevent errors in dealing with memory that you have reserved for your data.

8.2.1 Array bound checking and other memory techniques

In parallel codes, memory errors will often show up by a crash in an MPI routine. This is hardly ever an MPI problem or a problem with your cluster.

Compilers for Fortran often have support for array bound checking. Since this makes your code much slower, you would only enable it during the development phase of your code.

8.2.2 Memory leaks

We say that a program has a *memory leak*, if it allocates memory, and subsequently loses track of that memory. The operating system then thinks the memory is in use, while it is not, and as a result the computer memory can get filled up with allocated memory that serves no useful purpose.

In this example data is allocated inside a lexical scope:

```
for (i=.... ) {
  real *block = malloc( /* large number of bytes */ )
  /* do something with that block of memory */
  /* and forget to call "free" on that block */
}
```

The block of memory is allocated in each iteration, but the allocation of one iteration is no longer available in the next. A similar example can be made with allocating inside a conditional.

It should be noted that this problem is far less serious in Fortran, where memory is deallocated automatically as a variable goes out of scope.

There are various tools for detecting memory errors: Valgrind, DMALLOC, Electric Fence. For valgrind, see section 9.3.

8.2.3 Roll-your-own malloc

Many programming errors arise from improper use of dynamically allocated memory: the program writes beyond the bounds, or writes to memory that has not been allocated yet, or has already been freed. While some compilers can do bound checking at runtime, this slows down your program. A better strategy is to write your own memory management. Some libraries such as PETSc already supply an enhanced malloc; if this is available you should certainly make use of it. (The *gcc* compiler has a function `mcheck`, defined in `mcheck.h`, that has a similar function.)

If you write in C, you will probably know the `malloc` and `free` calls:

```
int *ip;
ip = (int*) malloc(500*sizeof(int));
if (ip==0) {/* could not allocate memory */}
..... do stuff with ip .....
free(ip);
```

You can save yourself some typing by

```
#define MYMALLOC(a,b,c) \
  a = (c*)malloc(b*sizeof(c)); \
  if (a==0) {/* error message and appropriate action */}

int *ip;
MYMALLOC(ip,500,int);
```

Runtime checks on memory usage (either by compiler-generated bounds checking, or through tools like valgrind or Rational Purify) are expensive, but you can catch many problems by adding some functionality to your malloc. What we will do here is to detect memory corruption after the fact.

We allocate a few integers to the left and right of the allocated object (line `1` in the code below), and put a recognizable value in them (line `2` and `3`), as well as the size of the object (line `2`). We then return the pointer to the actually requested memory area (line `4`).

```
#define MEMCOOKIE 137
#define MYMALLOC(a,b,c) { \
  char *aa; int *ii; \
  aa = malloc(b*sizeof(c)+3*sizeof(int)); /* 1 */ \
  ii = (int*)aa; ii[0] = b*sizeof(c); \
          ii[1] = MEMCOOKIE;              /* 2 */ \
  aa = (char*)(ii+2); a = (c*)aa ;        /* 4 */ \
  aa = aa+b*sizesof(c); ii = (int*)aa; \
          ii[0] = MEMCOOKIE;              /* 3 */ \
  }
```

Now you can write your own `free`, which tests whether the bounds of the object have not been written over.

```
#define MYFREE(a) { \
  char *aa; int *ii,; ii = (int*)a; \
  if (*(--ii)!=MEMCOOKIE) printf("object corrupted\n"); \
  n = *(--ii); aa = a+n; ii = (int*)aa; \
  if (*ii!=MEMCOOKIE)  printf("object corrupted\n"); \
  }
```

You can extend this idea: in every allocated object, also store two pointers, so that the allocated memory areas become a doubly linked list. You can then write a macro `CHECKMEMORY` which tests all your allocated objects for corruption.

Such solutions to the memory corruption problem are fairly easy to write, and they carry little overhead. There is a memory overhead of at most 5 integers per object, and there is practically no performance penalty.

(Instead of writing a wrapper for `malloc`, on some systems you can influence the behaviour of the system routine. On linux, `malloc` calls hooks that can be replaced with your own routines; see http://www.gnu.org/s/libc/manual/html_node/Hooks-for-Malloc.html.)

8.2.4 Specific techniques: Fortran

Use `Implicit none`.

Put all subprograms in modules so that the compiler can check for missing arguments and type mismatches. It also allows for automatic dependency building with `fdepend`.

Use the C preprocessor for conditional compilation and such.

8.3 Testing

There are various philosophies for testing the correctness of a code.

- Correctness proving: the programmer draws up predicates that describe the intended behaviour of code fragments and proves by mathematical techniques that these predicates hold [4, 2].
- Unit testing: each routine is tested separately for correctness. This approach is often hard to do for numerical codes, since with floating point numbers there is essentially an infinity of possible inputs, and it is not easy to decide what would constitute a sufficient set of inputs.
- Integration testing: test subsystems
- System testing: test the whole code. This is often appropriate for numerical codes, since we often have model problems with known solutions, or there are properties such as bounds that need to hold on the global solution.
- Test-driven design: the program development process is driven by the requirement that testing is possible at all times.

With parallel codes we run into a new category of difficulties with testing. Many algorithms, when executed in parallel, will execute operations in a slightly different order, leading to different roundoff behaviour. For instance, the parallel computation of a vector sum will use partial sums. Some algorithms have an inherent damping of numerical errors, for instance stationary iterative methods (section **??**), but others have no such built-in error correction (nonstationary methods; section **??**). As a result, the same iterative process can take different numbers of iterations depending on how many processors are used.

8.3.1 Test-driven design and development

In test-driven design there is a strong emphasis on the code always being testable. The basic ideas are as follows.

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

Chapter 9

Debugging

When a program misbehaves, *debugging* is the process of finding out *why*. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all proccessors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive *debugger*, a program that allows you to monitor and control the behaviour of a running program. In this section you will familiarize yourself with *gdb*, which is the open source debugger of the *GNU* project. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that gdb is a commandline debugger; there are graphical debuggers such as *ddd* (a frontend to gdb) or *DDT* and *TotalView* (debuggers for parallel codes). We limit ourselves to gdb, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with gdb and valgrind. The files can be downloaded from `http://tinyurl.com/ISTC-debug-tutorial`.

9.1 Invoking `gdb`

There are three ways of using gdb: using it to start a program, attaching it to an already running program, or using it to inspect a *core dump*. We will only consider the first possibility.

Here is an exaple of how to start gdb with program that has no arguments (Fortran users, use `hello.F`):

tutorials/gdb/c/hello.c

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  printf("hello world\n");
  return 0;
}

%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
Copyright 2004 Free Software Foundation, Inc. .... copyright info ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
(gdb) quit
%%
```

Important note: the program was compiled with the *debug flag* `-g`. This causes the *symbol table* (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations[1].

To illustrate the presence of the symbol table do

```
%% cc -g -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

and compare it with leaving out the `-g` flag:

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

1. Compiler optimizations are not supposed to change the semantics of a program, but sometimes do. This can lead to the nightmare scenario where a program crashes or gives incorrect results, but magically works correctly with compiled with debug and run in a debugger.

For a program with commandline input we give the arguments to the `run` command (Fortran users use `say.F`):

tutorials/gdb/c/say.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int i;
  for (i=0; i<atoi(argv[1]); i++)
    printf("hello world\n");
  return 0;
}

%% cc -o say -g say.c
%% ./say 2
hello world
hello world
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb/c/say 2
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.
```

9.2 Finding errors

Let us now consider some programs with errors.

9.2.1 C programs

```
// square.c
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",nmax);
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i); sum += squares[i];
  }
  printf("Sum: %e\n",sum);
```

```
%% cc -g -o square square.c
 %% ./square
5000
Segmentation fault
```

The *segmentation fault* (other messages are possible too) indicates that we are accessing memory that we are not allowed to, making the program abort. A debugger will quickly tell us where this happens:

```
%% gdb square
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()
```

Apparently the error occurred in a function `__svfscanf_l`, which is not one of ours, but a system function. Using the `backtrace` (or `bt`, also `where` or `w`) command we display the *call stack*. This usually allows us to find out where the error lies:

```
(gdb) backtrace
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7
```

We take a close look at line 7, and see that we need to change `nmax` to `&nmax`.

There is still an error in our program:

```
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x0000000100000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9           squares[i] = 1./(i*i); sum += squares[i];
```

We investigate further:

```
(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
(gdb) print squares
$2 = (float *) 0x0
```

and we quickly see that we forgot to allocate `squares`.

Memory errors can also occur if we have a legitimate array, but we access it outside its bounds.

```
// up.c
  int nlocal = 100,i;
  double s, *array = (double*) malloc(nlocal*sizeof(double));
  for (i=0; i<nlocal; i++) {
    double di = (double)i;
    array[i] = 1/(di*di);
  }
  s = 0.;
  for (i=nlocal-1; i>=0; i++) {
    double di = (double)i;
    s += array[i];
  }
```

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x0000000100200000
0x0000000100000f43 in main (argc=1, argv=0x7fff5fbfe2c0) at up.c:15
15          s += array[i];
(gdb) print array
$1 = (double *) 0x100104d00
(gdb) print i
$2 = 128608
```

9.2.2 Fortran programs

Compile and run the following program:

tutorials/gdb/f/square.F

```
      Program square
      real squares(1)
      integer i

      do i=1,100
         squares(i) = sqrt(1.*i)
         sum = sum + squares(i)
      end do
      print *,"Sum:",sum

      End
```

It should abort with a message such as 'Illegal instruction'. Running the program in gdb quickly tells you where the problem lies:

```
(gdb) run
Starting program: tutorials/gdb//fsquare
```

```
Reading symbols for shared libraries ++++. done

Program received signal EXC_BAD_INSTRUCTION,
Illegal instruction/operand.
0x0000000100000da3 in square () at square.F:7
7                sum = sum + squares(i)
```

We take a close look at the code and see that we did not allocate `squares` properly.

9.3 Memory debugging with Valgrind

Insert the following allocation of `squares` in your program:

```
squares = (float *) malloc( nmax*sizeof(float) );
```

Compile and run your program. The output will likely be correct, although the program is not. Can you see the problem?

To find such subtle memory errors you need a different tool: a memory debugging tool. A popular (because open source) one is *valgrind*; a common commercial tool is *purify*.

tutorials/gdb/c/square1.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",&nmax);
  squares = (float*) malloc(nmax*sizeof(float));
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i);
    sum += squares[i];
  }
  printf("Sum: %e\n",sum);

  return 0;
}
```

Compile this program with `cc -o square1 square1.c` and run it with `valgrind square1` (you need to type the input value). You will lots of output, starting with:

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==53695== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==53695== Command: a.out
```

```
==53695==
10
==53695== Invalid write of size 4
==53695==    at 0x100000EB0: main (square1.c:10)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
==53695==
==53695== Invalid read of size 4
==53695==    at 0x100000EC1: main (square1.c:11)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
```

Valgrind is informative but cryptic, since it works on the bare memory, not on variables. Thus, these error messages take some exegesis. They state that a line 10 writes a 4-byte object immediately after a block of 40 bytes that was allocated. In other words: the code is writing outside the bounds of an allocated array. Do you see what the problem in the code is?

Note that valgrind also reports at the end of the program run how much memory is still in use, meaning not properly `free`d.

If you fix the array bounds and recompile and rerun the program, valgrind still complains:

```
==53785== Conditional jump or move depends on uninitialised value(s)
==53785==    at 0x10006FC68: __dtoa (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x10003199F: __vfprintf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000738AA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000A1006: printf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x100000EF3: main (in ./square2)
```

Although no line number is given, the mention of `printf` gives an indication where the problem lies. The reference to an 'uninitialized value' is again cryptic: the only value being output is `sum`, and that is not uninitialized: it has been added to several times. Do you see why valgrind calls it uninitialized all the same?

9.4 Stepping through a program

Often the error in a program is sufficiently obscure that you need to investigate the program run in detail. Compile the following program

tutorials/gdb/c/roots.c

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

float root(int n)
{
```

```
  float r;
  r = sqrt(n);
  return r;
}

int main() {
  int i;
  float x=0;
  for (i=100; i>-100; i--)
    x += root(i+5);
  printf("sum: %e\n",x);
  return 0;
}
```

and run it:

```
%% ./roots
sum: nan
```

Start it in gdb as before:

```
%% gdb roots
GNU gdb 6.3.50-20050815
Copyright 2004 Free Software Foundation, Inc.
....
```

but before you run the program, you set a *breakpoint* at `main`. This tells the execution to stop, or 'break', in the main program.

```
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
```

Now the program will stop at the first executable statement in `main`:

```
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14        float x=0;
```

If execution is stopped at a breakpoint, you can do various things, such as issuing the `step` command:

```
Breakpoint 1, main () at roots.c:14
14        float x=0;
(gdb) step
15        for (i=100; i>-100; i--)
(gdb)
```

```
16          x += root(i);
(gdb)
```

(if you just hit return, the previously issued command is repeated). Do a number of `steps` in a row by hitting return. What do you notice about the function and the loop?

Switch from doing `step` to doing `next`. Now what do you notice about the loop and the function?

Set another breakpoint: `break 17` and do `cont`. What happens?

Rerun the program after you set a breakpoint on the line with the `sqrt` call. When the execution stops there do `where` and `list`.

9.5 Inspecting values

Run the previous program again in gdb: set a breakpoint at the line that does the `sqrt` call before you actually call `run`. When the program gets to line 8 you can do `print n`. Do `cont`. Where does the program stop?

If you want to repair a variable, you can do `set var=value`. Change the variable `n` and confirm that the square root of the new value is computed. Which commands do you do?

9.6 Breakpoints

If a problem occurs in a loop, it can be tedious keep typing `cont` and inspecting the variable with `print`. Instead you can add a condition to an existing breakpoint. First of all, you can make the breakpoint subject to a condition: with

```
condition 1 if (n<0)
```

breakpoint 1 will only obeyed if `n<0` is true.

You can also have a breakpoint that is only activated by some condition. The statement

```
break 8 if (n<0)
```

means that breakpoint 8 becomes (unconditionally) active after the condition `n<0` is encountered.

Another possibility is to use `ignore 1 50`, which will not stop at breakpoint 1 the next 50 times.

Remove the existing breakpoint, redefine it with the condition `n<0` and rerun your program. When the program breaks, find for what value of the loop variable it happened. What is the sequence of commands you use?

You can set a breakpoint in various ways:

- `break foo.c` to stop when code in a certain file is reached;

- `break 123` to stop at a certain line in the current file;
- `break foo` to stop at subprogram `foo`
- or various combinations, such as `break foo.c:123`.
- Finally,
- If you set many breakpoints, you can find out what they are with `info breakpoints`.
- You can remove breakpoints with `delete n` where `n` is the number of the breakpoint.
- If you restart your program with `run` without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: `save breakpoints <file>`. Use `source <file>` to read them in on the next gdb run.

Finally, you can execute commands at a breakpoint:

```
break 45
command
print x
cont
end
```

This states that at line 45 variable `x` is to be printed, and execution should immediately continue.

If you want to run repeated gdb sessions on the same program, you may want to save an reload breakpoints. This can be done with

```
save-breakpoint filename
source filename
```

9.7 Parallel debugging

Debugging in parallel is harder than sequentially, because you will run errors that are only due to interaction of processes such as *deadlock*; see section **??**.

As an example, consider this segment of MPI code:

```
MPI_Init(0,0);
// set comm, ntids, mytid
for (int it=0; ; it++) {
  double randomnumber = ntids * ( rand() / (double)RAND_MAX );
  printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
}
MPI_Finalize();
```

Each process computes random numbers until a certain condition is satisfied, then exits. However, consider introducing a barrier (or something that acts like it, such as a reduction):

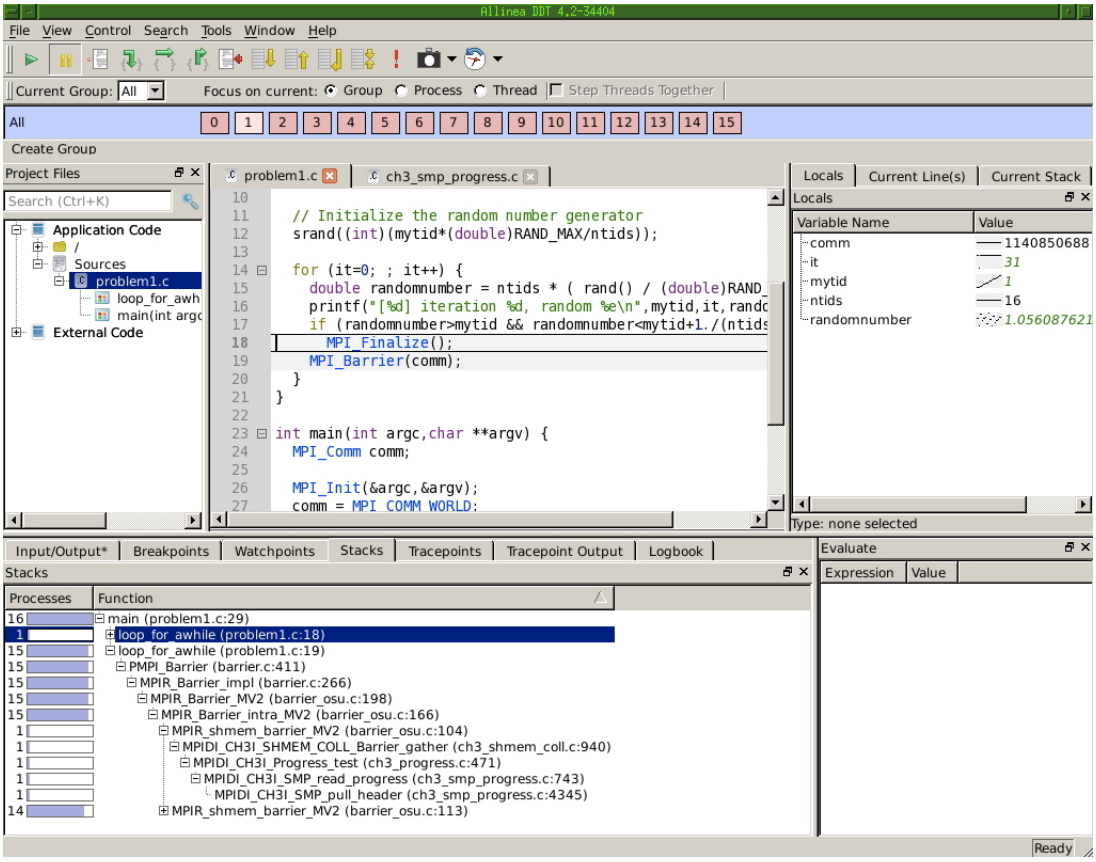

Figure 9.1: Display of 16 processes in the DDT debugger

```
for (int it=0; ; it++) {
  double randomnumber = ntids * ( rand() / (double)RAND_MAX );
  printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
  MPI_Barrier(comm);
}
MPI_Finalize();
```

Now the execution will hang, and this is not due to any particular process: each process has a code path from init to finalize that does not develop any memory errors or other runtime errors. However as soon as one process reaches the finalize call in the conditional it will stop, and all other processes will be waiting at the barrier.

Figure 9.1 shows the main display of the Allinea *DDT* debugger (http://www.allinea.com/products/ddt) at the point where this code stops. Above the source panel you see that there are 16 processes, and that the status is given for process 1. In the bottom display you see that out of 16 processes 15 are call-

ing `MPI_Barrier` on line 19, while one is at line 18. In the right display you see a listing of the local variables: the value specific to process 1. A rudimentary graph displays the values over the processors: the value of `ntids` is constant, that of `mytid` is linearly increasing, and `it` is constant except for one process.

Exercise 9.1. Make and run `ring_1a`. The program does not terminate and does not crash. In the debugger you can interrupt the execution, and see that all processes are executing a receive statement. This is probably a case of deadlock. Diagnose and fix the error.

Exercise 9.2. The author of `ring_1c` was very confused about how MPI works. Run the program. While it terminates without a problem, the output is wrong. Set a breakpoint at the send and receive statements to figure out what is happening.

9.8 Further reading

A good tutorial: http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.

Chapter 10

C for high performance

In this tutorial we will discuss some of the techniques that lift you from being a C beginner to a 'power user'.

10.1 C standards

The commonly accepted standard for the C language is *C99*. However, that doesn't mean compilers automatically accept source using this standard.

Use compiler option:

```
cc -std=c99 yourprogram.c
```

There are some *gcc extensions* that require the *Intel compiler* to specify

```
icc -std=gnu99 yourprogram.c
```

instead.

10.2 Allocation

The easiest way to define an array is to use *static allocation*:

```
double array[50];
```

This has some advantages, such as giving you *cacheline boundary alignment*.

However, since statically allocated arrays are created on the *stack*, they are subject to stacksize limits. Making the array too large will give a runtime error.

Use the Unix call

```
ulimit -s 123456
```

to increase the stacksize if needed.

The alternative is *dynamic allocation*, or allocation on the *heap*. This can use all available memory. The simplest way is by using `malloc`:

```
double *array = (double*) malloc( 50*sizeof(double) );
```

The `sizeof` function is necessary since `malloc` takes an argument in bytes.

The problem with this call is that you lose cacheline alignment, which can negatively influence performance. A better option is `posix_memalign`.

10.3 Compiler defines

There are various ways of altering program parameters without recompiling the code. One is to use *compiler macros*:

```
cc -c -DVALUE=50 myprogram.c
```

With the `-D` option your source acts as if there was a statement

```
#define VALUE 50
```

in it.

To use a default value for the case where you don't specify the compiler macros, have

```
#ifndef VALUE
#define VALUE 23
#endif
```

in your source.

10.4 Commandline arguments

10.5 Timers

See section 11.1.

Chapter 11

Performance measurement

Much of the teaching in this book is geared towards enabling you to write fast code, whether this is through the choice of the right method, or through optimal coding of a method. Consequently, you sometimes want to measure just *how fast* your code is. If you have a simulation that runs for many hours, you'd think just looking on the clock would be enough measurement. However, as you wonder whether your code could be faster than it is, you need more detailed measurements. This tutorial will teach you some ways to measure the behaviour of your code in more or less detail.

Here we will discuss

- timers: ways of measuring the execution time (and sometimes other measurements) of a particular piece of code, and
- profiling tools: ways of measuring how much time each piece of code, typically a subroutine, takes during a specific run.

11.1 Timers

There are various ways of timing your code, but mostly they come down to calling a routine twice that tells you the clock values:

```
tstart = clockticks()
....
tend = clockticks()
runtime = (tend-tstart)/ticks_per_sec
```

For instance, in Fortran there is the `system_clock` routine:

```
implicit none
INTEGER :: rate, tstart, tstop
REAL    :: time
real    :: a
integer :: i

CALL SYSTEM_CLOCK(COUNT_RATE = rate)
```

```
if (rate==0) then
   print *,"No clock available"
   stop
else
   print *,"Clock frequency:",rate
end if
CALL SYSTEM_CLOCK(COUNT = tstart)
a = 5
do i=1,1000000000
   a = sqrt(a)
end do
CALL SYSTEM_CLOCK(COUNT = tstop)
time = REAL( ( tstop - tstart )/ rate )
print *,a,tstart,tstop,time
end
```

with output

```
 Clock frequency:       10000
   1.000000       813802544   813826097   2.000000
```

In C there is the `clock` function:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main() {
  double start, stop, time;
  int i; double a=5;
  i = CLOCKS_PER_SEC; printf("clock resolution: %d\n",i);
  start = (double)clock()/CLOCKS_PER_SEC;
  for (i=0; i<1000000000; i++)
    a = sqrt(a);
  stop = (double)clock()/CLOCKS_PER_SEC;
  time = stop - start;
  printf("res: %e\nstart/stop: %e,%e\nTime: %e\n",a,start,stop,time);
  return 0;
}
```

with output

```
clock resolution: 1000000
res: 1.000000e+00
start/stop: 0.000000e+00,2.310000e+00
```

```
Time: 2.310000e+00
```

Do you see a difference between the Fortran and C approaches? Hint: what happens in both cases when the execution time becomes long? At what point do you run into trouble?

11.1.1 System utilities

There are unix system calls that can be used for timing: `getrusage` and `gettimeofday`. These timers have the advantage that they can distinguish between user time and system time, that is, exclusively timing program execution or giving *wallclock time* including all system activities.

11.2 Accurate counters

The timers in the previous section had a resolution of at best a millisecond, which corresponds to several thousand cycles on a modern CPU. For more accurate counting it is typically necessary to use assembly language, such as the Intel `RDTSC` (ReaD Time Stamp Counter) instruction http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM. However, this approach of using processor-specific timers is not portable. For this reason, the *PAPI* package (http://icl.cs.utk.edu/papi/) provides a uniform interface to *hardware counters*. You can see this package in action in the codes in appendix **??**.

In addition to timing, hardware counters can give you information about such things as cache misses and instruction counters. A processor typically has only a limited number of counters, but they can be assigned to various tasks. Additionally, PAPI has the concept of *derived metrics*.

11.3 Profiling tools

Profiling tools will give you the time spent in various events in the program, typically functions and subroutines, or parts of the code that you have declared as such. The tool will then report how many time the event occurred, total and average time spent, et cetera.

The only tool we mention here is *gprof*, the profiler of the *GNU* compiler. The TAU tool, discussed in section 11.4.1 for the purposes of tracing, also has profiling capabilities, presented in a nice graphic way. Finally, we mention that the *PETSc* library allows you to define your own timers and events.

```
% gcc  -g -pg ./srcFile.
% gprof    ./exeFile gmon.out > profile.txt
% gprof -l ./exeFile gmon.out > profile_line.txt
% gprof -A ./exeFile gmon.out > profile_anotated.tx
```

11.3.1 MPI profiling

The MPI library has been designed to make it easy to profile. Each function such as `MPI_Send` does no more than calling `PMPI_Send`. This means that a profiling library can redefine `MPI_Send` to

- initialize a profiling stage,
- call `PMPI_Send`,
- finish the profiling stage.

11.4 Tracing

In profiling we are only concerned with aggregate information: how many times a routine was called, and with what total/average/min/max runtime. However sometimes we want to know about the exact timing of events. This is especially relevant in a parallel context when we care about *load unbalance* and *idle time*.

Tools such as Vampyr can collect trace information about events and in particular messages, and render them in displays such as figure 11.1.

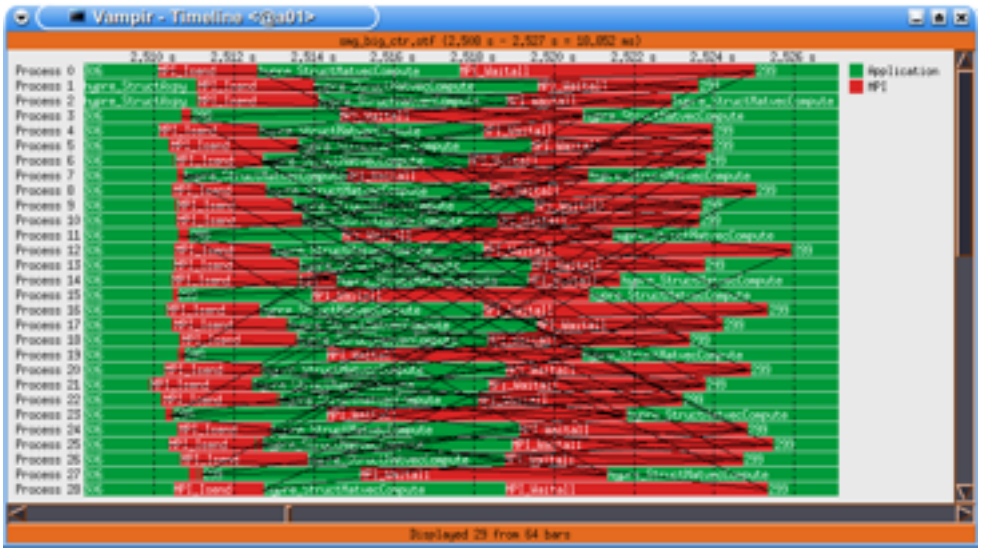

Figure 11.1: A Vampyr timeline diagram of a parallel process.

11.4.1 TAU

The TAU tool [10] (see `http://www.cs.uoregon.edu/research/tau/home.php` for the official documentation) uses *instrumentation* to profile and trace your code. That is, it adds profiling and trace calls to your code. You can then inspect the output after the run.

There are two ways to instrument your code:

- You can use *dynamic instrumentation*, where TAU adds the measurement facility at runtime:

```
# original commandline:
% mpicxx wave2d.cpp -o wave2d
# with TAU dynamic instrumentation:
% mpirun -np 12 tau_exec ./wave2d 500 500 3 4 5
```

- You can have the instrumentation added at compile time. For this, you need to let TAU take over the compilation in some sense.
  1. TAU has its own makefiles. The names and locations depend on your installation, but typically it will be something like
     ```
     export TAU_MAKEFILE=$TAU_HOME/lib/Makefile.tau-mpi-pdt
     ```
  2. Now you can invoke the TAU compilers `tau_cc,sh`, `tau_cxx.sh`, `tau_f90.sh`.

When you run your program you need to tell TAU what to do:

```
export TAU_TRACE=1
export TAU_PROFILE=1
export TRACEDIR=/some/dir
export PROFILEDIR=/some/dir
```

In order to generate trace plots you need to convert TAU output:

```
cd /some/dir # where the trace and profile output went
tau_treemerge.pl
tau2slog2 tau.trc tau.edf -o yourrun.slog2
```

The `slog2` file can be displayed with *jumpshot*.

Chapter 12

C/Fortran interoperability

Most of the time, a program is written is written in a single language, but in some circumstances it is necessary or desirable to mix sources in more than one language for a single executable. One such case is when a library is written in one language, but used by a program in another. In such a case, the library writer will probably have made it easy for you to use the library; this section is for the case that you find yourself in the place of the library writer. We will focus on the common case of *interoperability* between C/C++ and Fortran.

This issue is complicated by the fact that both languages have been around for a long time, and various recent language standards have introduced mechanisms to facilitate interoperability. However, there is still a lot of old code around, and not all compilers support the latest standards. Therefore, we discuss both the old and the new solutions.

12.1 Linker conventions

As explained above, a compiler turns a source file into a binary, which no longer has any trace of the source language: it contains in effect functions in machine language. The linker will then match up calls and definitions, which can be in different files. The problem with using multiple languages is then that compilers have different notions of how to translate function names from the source file to the binary file.

Let's look at codes (you can find example files in `tutorials/linking`):

```
// C:
      Subroutine foo()
      Return
      End Subroutine
! Fortran
void foo() {
  return;
}
```

After compilation you can use *nm* to investigate the binary *object file*:

```
%% nm fprog.o
0000000000000000 T _foo_
....
%% nm cprog.o
0000000000000000 T _foo
....
```

You see that internally the `foo` routine has different names: the Fortran name has an underscore appended. This makes it hard to call a Fortran routine from C, or vice versa. The possible name mismatches are:

- The Fortran compiler appends an underscore. This is the most common case.
- Sometimes it can append two underscores.
- Typically the routine name is lowercase in the object file, but uppercase is a possibility too.

Since C is a popular language to write libraries in, this means that the problem is often solved in the C library by:

- Appending an underscore to all C function names; or
- Including a simple wrapper call:

```
int SomeCFunction(int i,float f)
{
  .....
}
int SomeCFunction_(int i,float f)
{
  return SomeCFunction(i,f);
}
```

12.1.1 C bindings in Fortran 2003

With the latest Fortran standard there are explicit *C bindings*, making it possible to declare the external name of variables and routines:

```
module operator
  real, bind(C) :: x
contains
  subroutine s() bind(C,name='s')
  return
  end subroutine
end module

%% ifort -c fbind.F90
%% nm fbind.o
.... T _s
.... C _x
```

It is also possible to declare data types to be C-compatible:

```
Program fdata

  use iso_c_binding

  type, bind(C) :: c_comp
    real (c_float)  :: data
    integer (c_int) :: i
    type (c_ptr)    :: ptr
  end type

end Program fdata
```

12.1.2 C++ linking

Libraries written in C++ offer further problems. The C++ compiler makes external symbols by combining the names a class and its methods, in a process known as *name mangling*. You can force the compiler to generate names that are intelligible to other languages by

```
#ifdef __cplusplus
  extern"C" {
#endif
  .
  .
  place declarations here
  .
  .
#ifdef __cplusplus
  }
#endif
```

Example: compiling

```
#include <stdlib.h>

int foo(int x) {
  return x;
}
```

and inspecting the output with `nm` gives:

```
0000000000000010 s EH_frame1
0000000000000000 T _foo
```

On the other hand, the identical program compiled as C++ gives

```
0000000000000010 s EH_frame1
0000000000000000 T __Z3fooi
```

You see that the name for `foo` is something mangled, so you can not call this routine from a program in a different language. On the other hand, if you add the `extern` declaration:

```
#include <stdlib.h>

#ifdef __cplusplus
  extern"C" {
#endif
int foo(int x) {
  return x;
}
#ifdef __cplusplus
  }
#endif
```

you again get the same linker symbols as for C, so that the routine can be called from both C and Fortran.

If your main program is in C, you can use the C++ compiler as linker. If the main program is in Fortran, you need to use the Fortran compiler as linker. It is then necessary to link in extra libraries for the C++ system routines. For instance, with the Intel compiler `-lstdc++ -lc` needs to be added to the link line.

The use of `extern` is also needed if you link other languages to a C++ main program. For instance, a Fortran subprogram `foo` should be declared as

```
extern "C" {
void foo_();
}
```

In that case, you again use the C++ compiler as linker.

12.1.3 Complex numbers

The *complex data types in C/C++ and Fortran* are compatible with each other. Here is an example of a C++ program linking to Lapack's complex vector scaling routine *zscal*.

```
// zscale.cxx
extern "C" {
void zscal_(int*,double complex*,double complex*,int*);
}
  complex double *xarray,*yarray, scale=2.;
  xarray = new double complex[n]; yarray = new double complex[n];
  zscal_(&n,&scale,xarray,&ione);
```

12.2 Arrays

C and Fortran have different conventions for storing multi-dimensional arrays. You need to be aware of this when you pass an array between routines written in different languages.

Fortran stores multi-dimensional arrays in *column-major* order; see figure 12.1. For two dimensional arrays `A(i,j)` this means that the elements in each column are stored contiguously: a 2×2 array is stored as `A(1,1), A(2,1), A(1,2), A(2,2)`. Three and higher dimensional arrays are an obvious exten-

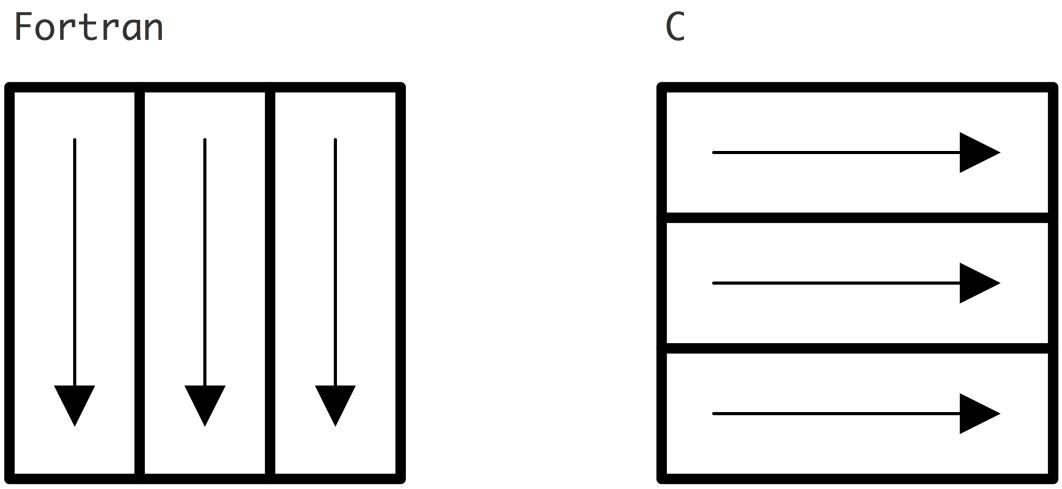

Figure 12.1: Fortran and C array storage by columns and rows respectively

sion: it is sometimes said that 'the left index varies quickest'.

C arrays are stored in *row-major* order: elements in each row are stored contiguous, and columns are then placed sequentially in memory. A 2×2 array `A[2][2]` is stored as `A[1][1], A[1][2], A[2][1], A[2][2]`.

A number of remarks about arrays in C.

- C (before the C99 standard) has multi-dimensional arrays only in a limited sense. You can declare them, but if you pass them to another C function, they no longer look multi-dimensional: they have become plain `float*` (or whatever type) arrays. That brings us to the next point.
- Multi-dimensional arrays in C look as if they have type `float**`, that is, an array of pointers that point to (separately allocated) arrays for the rows. While you could certainly implement this:

```
float **A;
A = (float**)malloc(m*sizeof(float*));
for (i=0; i<n; i++)
  A[i] = (float*)malloc(n*sizeof(float));
```

  careful reading of the standard reveals that a multi-dimensional array is in fact a single block of memory, no further pointers involved.

Given the above limitation on passing multi-dimensional arrays, and the fact that a C routine can not tell

whether it's called from Fortran or C, it is best not to bother with multi-dimensional arrays in C, and to emulate them:

```
float *A;
A = (float*)malloc(m*n*sizeof(float));
#define SUB(i,j,m,n) i+j*m
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    .... A[SUB(i,j,m,n)] ....
```

where for interoperability we store the elements in column-major fashion.

12.2.1 Array alignment

For reasons such as Single Instruction Multiple Data (SIMD) *vector instructions*, it can be advantageous to use *aligned allocation*. For instance, '16-byte alignment' means that the starting address of your array, expressed in bytes, is a multiple of 16.

In C, you can force such alignment with `posix_memalign`. In Fortran there is no general mechanism for this. The Intel compiler allows you to write:

```
double precision, allocatable :: A(:), B(:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A, B
```

12.3 Strings

Programming languages differ widely in how they handle strings.

- In C, a string is an array of characters; the end of the string is indicated by a null character, that is the ascii character zero, which has an all zero bit pattern. This is called *null termination*.
- In Fortran, a string is an array of characters. The length is maintained in a internal variable, which is passed as a hidden parameter to subroutines.
- In Pascal, a string is an array with an integer denoting the length in the first position. Since only one byte is used for this, strings can not be longer than 255 characters in Pascal.

As you can see, passing strings between different languages is fraught with peril. This situation is made even worse by the fact that passing strings as subroutine arguments is not standard.

Example: the main program in Fortran passes a string

```
Program Fstring
  character(len=5) :: word = "Word"
  call cstring(word)
end Program Fstring
```

and the C routine accepts a character string and its length:

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

void cstring_(char *txt,int txtlen) {
  printf("length = %d\n",txtlen);
  printf("<<");
  for (int i=0; i<txtlen; i++)
    printf("%c",txt[i]);
  printf(">>\n");
}
```

which produces:

```
length = 5
<<Word >>
```

To pass a Fortran string to a C program you need to append a null character:

```
call cfunction ('A string'//CHAR(0))
```

Some compilers support extensions to facilitate this, for instance writing

```
DATA forstring /'This is a null-terminated string.'C/
```

Recently, the 'C/Fortran interoperability standard' has provided a systematic solution to this.

12.4 Subprogram arguments

In C, you pass a `float` argument to a function if the function needs its value, and `float*` if the function has to modify the value of the variable in the calling environment. Fortran has no such distinction: every variable is passed *by reference*. This has some strange consequences: if you pass a literal value `37` to a subroutine, the compiler will allocate a nameless variable with that value, and pass the address of it, rather than the value[1].

For interfacing Fortran and C routines, this means that a Fortran routine looks to a C program like all its argument are 'star' arguments. Conversely, if you want a C subprogram to be callable from Fortran, all its arguments have to be star-this or that. This means on the one hand that you will sometimes pass a variable by reference that you would like to pass by value.

Worse, it means that C subprograms like

```
void mysub(int **iarray) {
 *iarray = (int*)malloc(8*sizeof(int));
 return;
```

1. With a bit of cleverness and the right compiler, you can have a program that says `print *,7` and prints `8` because of this.

```
}
```

can not be called from Fortran. There is a hack to get around this (check out the Fortran77 interface to the Petsc routine `VecGetValues`) and with more cleverness you can use `POINTER` variables for this.

12.5 Input/output

Both languages have their own system for handling input/output, and it is not really possible to meet in the middle. Basically, if Fortran routines do I/O, the main program has to be in Fortran. Consequently, it is best to isolate I/O as much as possible, and use C for I/O in mixed language programming.

12.6 Fortran/C interoperability in Fortran2003

The latest version of Fortran, unsupported by many compilers at this time, has mechanisms for interfacing to C.

- There is a module that contains named kinds, so that one can declare
  ```
  INTEGER,KIND(C_SHORT) :: i
  ```
- Fortran pointers are more complicated objects, so passing them to C is hard; Fortran2003 has a mechanism to deal with C pointers, which are just addresses.
- Fortran derived types can be made compatible with C structures.

Chapter 13

LaTeX for scientific documentation

13.1 The idea behind LaTeX, some history of TeX

TeX is a typesetting system that dates back to the late 1970s. In those days, graphics terminals where you could design a document layout and immediately view it, the way you can with for instance Microsoft Word, were rare. Instead, TeX uses a two-step workflow, where you first type in your document with formatting instructions in an ascii document, using your favourite text editor. Next, you would invoke the `latex` program, as a sort of compiler, to translate this document to a form that can be printed or viewed.

```
%% edit mydocument.tex
%% latex mydocument
%% # print or view the resulting output
```

The process is comparable to making web pages by typing HTML commands.

This way of working may seem clumsy, but it has some advantages. For instance, the TeX input files are plain ascii, so they can easily be generated automatically, for instance from a database. Also, you can edit them with whatever your favourite editor happens to be.

Another point in favour of TeX is the fact that the layout is specified by commands that are written in a sort of programming language. This has some important consequences:

- Separation of concerns: when you are writing your document, you do not have to think about layout. You give the 'chapter' command, and the implementation of that command will be decided independently, for instance by you choosing a document style.
- Changing the layout of a finished document is easily done by choosing a different realization of the layout commands in the input file: the same 'chapter' command is used, but by choosing a different style the resulting layout is different. This sort of change can be as simple as a one-line change to the document style declaration.
- If you have unusual typesetting needs, it is possible to write new TeX commands for this. For many needs such extensions have in fact already been written; see section 13.4.

The commands in TeX are fairly low level. For this reason, a number of people have written systems on top of TeX that offer powerful features, such as automatic cross-referencing, or generation of a table of contents. The most popular of these systems is LaTeX. Since TeX is an interpreted system, all of its mechanisms are still available to the user, even though LaTeX is loaded on top of it.

13.1.1 Installing LaTeX

The easiest way to install LaTeX on your system is by downloading the TeXlive distribution from `http://tug.org/texlive`. Apple users can also use `fink` or `macports`. Various front-ends to TeX exist, such as TeXshop on the Mac.

13.1.2 Running LaTeX

Purpose. In this section you will run the LaTeX compiler

Originally, the `latex` compiler would output a device independent file format, named `dvi`, which could then be translated to PostScript or PDF, or directly printed. These days, many people use the `pdflatex` program which directly translates `.tex` files to `.pdf` files. This has the big advantage that the generated PDF files have automatic cross linking and a side panel with table of contents. An illustration is found below.

Let us do a simple example.

```
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Figure 13.1: A minimal LaTeX document

Exercise. Create a text file `minimal.tex` with the content as in figure 13.1. Try the command `pdflatex minimal` or `latex minimal`. Did you get a file `minimal.pdf` in the first case or `minimal.dvi` in the second case? Use a pdf viewer, such as Adobe Reader, or `dvips` respectively to view the output.

Caveats. If you make a typo, TeX can be somewhat unfriendly. If you get an error message and TeX is asking for input, typing `x` usually gets you out, or `Ctrl-C`. Some systems allow you to type `e` to go directly into the editor to correct the typo.

13.2 A gentle introduction to LaTeX

Here you will get a very brief run-through of LaTeX features. There are various more in-depth tutorials available, such as the one by Oetiker [9].

13.2.1 Document structure

Each LaTeX document needs the following lines:

```
\documentclass{ .... } % the dots will be replaced

\begin{document}

\end{document}
```

The 'documentclass' line needs a class name in between the braces; typical values are 'article' or 'book'. Some organizations have their own styles, for instance 'ieeeproc' is for proceedings of the IEEE.

All document text goes between the `\begin{document}` and `\end{document}` lines. (Matched 'begin' and 'end' lines are said to denote an 'environment', in this case the document environment.)

The part before `\begin{document}` is called the 'preamble'. It contains customizations for this particular document. For instance, a command to make the whole document double spaced would go in the preamble. If you are using `pdflatex` to format your document, you want a line

```
\usepackage{hyperref}
```

here.

Have you noticed the following?

- The backslash character is special: it starts a LaTeX command.
- The braces are also special: they have various functions, such as indicating the argument of a command.
- The percent character indicates that everything to the end of the line is a comment.

13.2.2 Some simple text

Purpose. In this section you will learn some basics of text formatting.

Exercise. Create a file `first.tex` with the content of figure 13.1 in it. Type some text in the preamble, that is, before the `\begin{document}` line and run `pdflatex` on your file.

Expected outcome. You should get an error message because you are not allowed to have text in the preamble. Only commands are allowed there; all text has to go after `\begin{document}`.

Exercise. Edit your document: put some text in between the `\begin{document}` and `\end{document}` lines. Let your text have both some long lines that go on for a while, and some short ones. Put superfluous spaces between words, and at the beginning or end of lines. Run `pdflatex` on your document and view the output.

Expected outcome. You notice that the white space in your input has been collapsed in the output. TeX has its own notions about what space should look like, and you do not have to concern yourself with this matter.

Exercise. Edit your document again, cutting and pasting the paragraph, but leaving a blank line between the two copies. Paste it a third time, leaving several blank lines. Format, and view the output.

Expected outcome. TEX interprets one or more blank lines as the separation between paragraphs.

Exercise. Add `\usepackage{pslatex}` to the preamble and rerun `pdflatex` on your document. What changed in the output?

Expected outcome. This should have the effect of changing the typeface from the default to Times Roman.

Caveats. Typefaces are notoriously unstandardized. Attempts to use different typefaces may or may not work. Little can be said about this in general.

Add the following line before the first paragraph:

```
\section{This is a section}
```

and a similar line before the second. Format. You see that LATEX automatically numbers the sections, and that it handles indentation different for the first paragraph after a heading.

Exercise. Replace `article` by `artikel3` in the documentclass declaration line and reformat your document. What changed?

Expected outcome. There are many documentclasses that implement the same commands as `article` (or another standard style), but that have their own layout. Your document should format without any problem, but get a better looking layout.

Caveats. The `artikel3` class is part of most distributions these days, but you can get an error message about an unknown documentclass if it is missing or if your environment is not set up correctly. This depends on your installation. If the file seems missing, download the files from http://tug.org/texmf-dist/tex/latex/ntgclass/ and put them in your current directory; see also section 13.2.9.

13.2.3 Math

Purpose. In this section you will learn the basics of math typesetting

One of the goals of the original TEX system was to facilitate the setting of mathematics. There are two ways to have math in your document:

- Inline math is part of a paragraph, and is delimited by dollar signs.
- Display math is, as the name implies, displayed by itself.

Exercise. Put `$x+y$` somewhere in a paragraph and format your document. Put `\[x+y\]` somewhere in a paragraph and format.

Expected outcome. Formulas between single dollars are included in the paragraph where you declare them. Formulas between `\[...\]` are typeset in a display.

For display equations with a number, use an `equation` environment. Try this.

Here are some common things to do in math. Make sure to try them out.

- Subscripts and superscripts: `$x_i^2$`. If the sub or superscript is more than a single symbol, it needs to be grouped: `$x_{i+1}^{2n}$`. If you need a brace in a formula, use `$\{ \}$`.
- Greek letters and other symbols: `$\alpha\otimes\beta_i$`.
- Combinations of all these `$\int_{t=0}^\infty tdt$`.

Exercise. Take the last example and typeset it as display math. Do you see a difference with inline math?

> *Expected outcome.* TEX tries not to include the distance between text lines, even if there is math in a paragraph. For this reason it typesets the bounds on an integral sign differently from display math.

13.2.4 Referencing

> **Purpose.** In this section you will see TEX's cross referencing mechanism in action.

So far you have not seen LATEX do much that would save you any work. The cross referencing mechanism of LATEX will definitely save you work: any counter that LATEX inserts (such as section numbers) can be referenced by a label. As a result, the reference will always be correct.

Start with an example document that has at least two section headings. After your first section heading, put the command `\label{sec:first}`, and put `\label{sec:other}` after the second section heading. These label commands can go on the same line as the `section` command, or on the next. Now put

```
As we will see in section~\ref{sec:other}.
```

in the paragraph before the second section. (The tilde character denotes a non-breaking space.)

Exercise. Make these edits and format the document. Do you see the warning about an undefined reference? Take a look at the output file. Format the document again, and check the output again. Do you have any new files in your directory?

> *Expected outcome.* On a first pass through a document, the TEX compiler will gather all labels with their values in a `.aux` file. The document will display a double question mark for any references that are unknown. In the second pass the correct values will be filled in.
>
> *Caveats.* If after the second pass there are still undefined references, you probably made a typo. If you use the `bibtex` utility for literature references, you will regularly need three passes to get all references resolved correctly.

Above you saw that the `equation` environment gives displayed math with an equation number. You can add a label to this environment to refer to the equation number.

Exercise. Write a formula in an `equation` environment, and add a label. Refer to this label anywhere in the text. Format (twice) and check the output.

> *Expected outcome.* The `\label` and `\ref` command are used in the same way for formulas as for section numbers. Note that you must use `\begin/end{equation}` rather than `\[...\]` for the formula.

13.2.5 Lists

Purpose. In this section you will see the basics of lists.

Bulleted and numbered lists are provided through an environment.

```
\begin{itemize}
\item This is an item;
\item this is one too.
\end{itemize}
\begin{enumerate}
\item This item is numbered;
\item this one is two.
\end{enumerate}
```

Exercise. Add some lists to your document, including nested lists. Inspect the output.

Expected outcome. Nested lists will be indented further and the labeling and numbering style changes with the list depth.

Exercise. Add a label to an item in an `enumerate` list and refer to it.

Expected outcome. Again, the `\label` and `\ref` commands work as before.

13.2.6 Source code and algorithms

As a computer scientist, you will often want to include algorithms in your writings; sometimes even source code.

In this tutorial so far you have seen that some characters have special meaning to LaTeX, and just can not just type them and expect them to show up in the output. Since funny characters appear quite regularly in programming languages, we need a tool for this: the *verbatim mode*.

To display bits of code inside a paragraph, you use the `\verb` command. This command delimits its argument with two identical characters that can not appear in the verbatim text. For instance, the output `if (x%5>0) { ... }` is produced by `\verb+if (x%5>0) { ... }+`. (Exercise: how did the author of this book get that verbatim command in the text?)

For longer stretches of verbatim text, that need to be displayed by themselves you use

```
\begin{verbatim}
stuff
\end{verbatim}
```

Finally, in order to include a whole file as verbatim listing, use .

Verbatim text is one way of displaying algorithms, but there are more elegant solutions. For instance, in this book the following is used:

```
\usepackage[algo2e,noline,noend]{algorithm2e}
```

The result can be seen, for instance, on page **??**.

13.2.7 Graphics

Since you can not immediately see the output of what you are typing, sometimes the output may come as a surprise. That is especially so with graphics. LaTeX has no standard way of dealing with graphics, but the following is a common set of commands:

```
\usepackage{graphicx} % this line in the preamble

\includegraphics{myfigure} % in the body of the document
```

The figure can be in any of a number of formats, except that PostScript figures (with extension `.ps` or `.eps`) can not be used if you use pdflatex.

Since your figure is often not the right size, the include line will usually have something like:

```
\includegraphics[scale=.5]{myfigure}
```

A bigger problem is that figures can be too big to fit on the page if they are placed where you declare them. For this reason, they are usually treated as 'floating material'. Here is a typical declaration of a figure:

```
\begin{figure}[ht]
  \includegraphics{myfigure}
  \caption{This is a figure}
  \label{fig:first}
\end{figure}
```

It contains the following elements:

- The `figure` environment is for 'floating' figures; they can be placed right at the location where they are declared, at the top or bottom of the next page, at the end of the chapter, et cetera.
- The `[ht]` argument of the `\begin{figure}` line states that your figure should be attempted to be placed `h`ere; it that does not work, it should go `t`op of the next page. The remaining possible specifications are `b` for placement at the bottom of a page, or `p` for placement on a page by itself. For example

  ```
  \begin{figure}[hbp]
  ```

  declares that the figure has to be placed here if possible, at the bottom of the page if that's not possible, and on a page of its own if it is too big to fit on a page with text.
- A caption to be put under the figure, including a figure number;
- A label so that you can refer to the figure number by its label: `figure~\ref{fig:first}`.
- And of course the figure material. There are various ways to fine-tune the figure placement. For instance

  ```
  \begin{center}
    \includegraphics{myfigure}
  \end{center}
  ```

  gives a centered figure.

13.2.8 Bibliography references

The mechanism for citing papers and books in your document is a bit like that for cross referencing. There are labels involved, and there is a `\cite{thatbook}` command that inserts a reference, usually numeric. However, since you are likely to refer to a paper or book in more than one document your write, LaTeX allows you to have a database of literature references in a file by itself, rather than somewhere in your document.

Make a file `mybibliography.bib` with the following content:

```
@article{JoeDoe1985,
author = {Joe Doe},
title = {A framework for bibliography references},
journal = {American Library Assoc. Mag.},
year = {1985}
}
```

In your document `mydocument.tex`, put

```
For details, refer to Doe~\cite{JoeDoe1985} % somewhere in the text

\bibliography{mybibliography} % at the end of the document
\bibliographystyle{plain}
```

Format your document, then type on the commandline

```
bibtex mydocument
```

and format your document two more times. There should now be a bibliography in it, and a correct citation. You will also see that files `mydocument.bbl` and `mydocument.blg` have been created.

13.2.9 Environment variables

On Unix systems, TeX investigates the `TEXINPUTS` *environment variable* when it tries to find an include file. Consequently, you can create a directory for your styles and other downloaded include files, and set this variable to the location of that directory. Similarly, the `BIBINPUTS` variable indicates the location of bibliography files for bibtex (section 13.2.8).

13.3 A worked out example

The following example `demo.tex` contains many of the elements discussed above.

```
\documentclass{artikel3}

\usepackage{pslatex,graphicx,amsmath,amssymb}
\usepackage{pdflatex}

\newtheorem{theorem}{Theorem}
```

```
\newcounter{excounter}
\newenvironment{exercise}
  {\refstepcounter{excounter}
   \begin{quotation}\textbf{Exercise \arabic{excounter}.} }
  {\end{quotation}}

\begin{document}
\title{SSC 335: demo}
\author{Victor Eijkhout}
\date{today}
\maketitle

\section{This is a section}
\label{sec:intro}

This is a test document, used in~\cite{latexdemo}. It contains a
discussion in section~\ref{sec:discussion}.

\begin{exercise}\label{easy-ex}
  Left to the reader.
\end{exercise}
\begin{exercise}
  Also left to the reader, just like in exercise~\ref{easy-ex}
\end{exercise}

\begin{theorem}
  This is cool.
\end{theorem}
This is a formula: $a\Leftarrow b$.
\begin{equation}
  \label{eq:one}
    x_i\leftarrow y_{ij}\cdot x^{(k)}_j
\end{equation}
Text: $\int_0^1 \sqrt x\,dx$
\[
  \int_0^1 \sqrt x\,dx
\]
\section{This is another section}
\label{sec:discussion}

\begin{table}[ht]
  \centering
  \begin{tabular}{|rl|}
    \hline one&value \\ \hline another&values \\ \hline
  \end{tabular}
  \caption{This is the only table in my demo}
  \label{tab:thetable}
\end{table}
```

```
\begin{figure}[ht]
  \centering
  \includegraphics{graphics/caches}
  \caption{this is the only figure}
  \label{fig:thefigure}
\end{figure}
As I showed in the introductory section~\ref{sec:intro}, in the
paper~\cite{AdJo:colorblind}, it was shown that
equation~\eqref{eq:one}
\begin{itemize}
\item There is an item.
\item There is another item
  \begin{itemize}
  \item sub one
  \item sub two
  \end{itemize}
\end{itemize}
\begin{enumerate}
\item item one
\item item two
  \begin{enumerate}
  \item sub one
  \item sub two
  \end{enumerate}
\end{enumerate}

\tableofcontents
\listoffigures

\bibliography{math}
\bibliographystyle{plain}

\end{document}
```

You also need the file `math.bib`:

```
@article{AdJo:colorblind,
author = {Loyce M. Adams and Harry F. Jordan},
title = {Is {SOR} color-blind?},
journal = {SIAM J. Sci. Stat. Comput.},
year = {1986},
volume = {7},
pages = {490--506},
abstract = {For what stencils do ordinary and multi-colour SOR have
the same eigenvalues.},
keywords = {SOR, colouring}
}

@misc{latexdemo,
```

```
author = {Victor Eijkhout},
title = {Short {\LaTeX}\ demo},
note = {SSC 335, oct 1, 2008}
}
```

The following sequence of commands

```
pdflatex demo
bibtex demo
pdflatex demo
pdflatex demo
```

gives

SSC 335: demo

Victor Eijkhout

today

1 This is a section

This is a test document, used in [2]. It contains a discussion in section 2.

Exercise 1. Left to the reader.

Exercise 2. Also left to the reader, just like in exercise 1

Theorem 1 *This is cool.*

This is a formula: $a \Leftarrow b$.

$$x_i \leftarrow y_{ij} \cdot x_j^{(k)} \tag{1}$$

Text: $\int_0^1 \sqrt{x}dx$

$$\int_0^1 \sqrt{x}dx$$

2 This is another section

one	value
another	values

Table 1: This is the only table in my demo

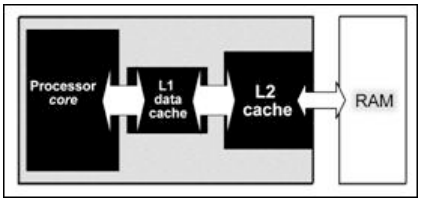

Figure 1: this is the only figure

As I showed in the introductory section 1, in the paper [1], it was shown that equation (1)

- There is an item.

1

- There is another item
  - sub one
  - sub two

1. item one
2. item two
   (a) sub one
   (b) sub two

Contents

List of Figures

References

[1] Loyce M. Adams and Harry F. Jordan. Is SOR color-blind? *SIAM J. Sci. Stat. Comput.*, 7:490–506, 1986.
[2] Victor Eijkhout. Short LaTeX demo. SSC 335, oct 1, 2008.

13.4 Where to take it from here

This tutorial touched only briefly on some essentials of TEX and LATEX. You can find longer intros online [9], or read a book [6, 5, 8]. Macro packages and other software can be found on the Comprehensive TEX Archive `http://www.ctan.org`. For questions you can go to the newsgroup `comp.text.tex`, but the most common ones can often already be found on web sites [11].

13.5 Review questions

Exercise 13.1. Write a one or two page document about your field of study. Show that you have mastered the following constructs:

- formulas, including labels and referencing;
- including a figure;
- using bibliography references;
- construction of nested lists.

Chapter 14

Bibliography

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The Awk Programming Language*. Addison-Wesley Series in Computer Science. Addison-Wesley Publ., 1988. ISBN 020107981X, 9780201079814.

[2] Edsger W. Dijkstra. Programming as a discipline of mathematical nature. *Am. Math. Monthly*, 81:608–612, 1974.

[3] Dale Dougherty and Arnold Robbins. *sed & awk*. O'Reilly Media, 2nd edition edition. Print ISBN: 978-1-56592-225-9 , ISBN 10:1-56592-225-5; Ebook ISBN: 978-1-4493-8700-6, ISBN 10:1-4493-8700-4.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969.

[5] Helmut Kopka and Patrick W. Daly. *A Guide to LaTeX*. Addison-Wesley, first published 1992.

[6] L. Lamport. *LaTeX, a Document Preparation System*. Addison-Wesley, 1986.

[7] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly Media, 3rd edition edition, 2004. Print ISBN:978-0-596-00610-5 ISBN 10:0-596-00610-1 Ebook ISBN:978-0-596-10445-0 ISBN 10:0-596-10445-6.

[8] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LaTeX Companion, 2nd edition*. Addison-Wesley, 2004.

[9] Tobi Oetiker. The not so short introductino to LaTeX. http://tobi.oetiker.ch/lshort/.

[10] S. Shende and A. D. Malony. *International Journal of High Performance Computing Applications*, 20:287–331, 2006.

[11] TeX frequently asked questions.

Chapter 15

List of acronyms

AMR Adaptive Mesh Refinement
AOS Array-Of-Structures
API Application Programmer Interface
AVX Advanced Vector Extensions
BEM Boundary Element Method
BFS Breadth-First Search
BLAS Basic Linear Algebra Subprograms
BSP Bulk Synchronous Parallel
BVP Boundary Value Problem
CAF Co-array Fortran
CCS Compressed Column Storage
CG Conjugate Gradients
CGS Classical Gram-Schmidt
COO Coordinate Storage
CRS Compressed Row Storage
DAG Directed Acyclic Graph
DRAM Dynamic Random-Access Memory
DSP Digital Signal Processing
FD Finite Difference
FMA Fused Multiply-Add
FDM Finite Difference Method
FEM Finite Element Method
FMM Fast Multipole Method
FOM Full Orthogonalization Method
FPU Floating Point Unit
FFT Fast Fourier Transform
FSA Finite State Automaton
FSB Front-Side Bus
FPGA Field-Programmable Gate Array
GMRES Generalized Minimum Residual
GPU Graphics Processing Unit
GPGPU General Purpose Graphics Processing Unit
GS Gram-Schmidt
HDFS Hadoop File System
HPC High-Performance Computing
HPF High Performance Fortran
IBVP Initial Boundary Value Problem
IDE Integrated Development Environment
ILP Instruction Level Parallelism
ILU Incomplete LU
IVP Initial Value Problem
LAN Local Area Network
LBM Lattice Boltzmann Method
LRU Least Recently Used
MIC Many Integrated Cores
MIMD Multiple Instruction Multiple Data
MGS Modified Gram-Schmidt
MPI Message Passing Interface
MSI Modified-Shared-Invalid
MTA Multi-Threaded Architecture
NUMA Non-Uniform Memory Access
ODE Ordinary Diffential Equation
OS Operating System
PGAS Partitioned Global Address Space
PDE Partial Diffential Equation
PRAM Parallel Random Access Machine
RDMA Remote Direct Memory Access
SAN Storage Area Network
SAS Software As a Service
SFC Space-Filling Curve
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Thread

SM Streaming Multiprocessor
SMP Symmetric Multi Processing
SMT Symmetric Multi Threading
SOA Structure-Of-Arrays
SOR Successive Over-Relaxation
SP Streaming Processor
SPMD Single Program Multiple Data
SPD symmetric positive definite
SRAM Static Random-Access Memory
SSE SIMD Streaming Extensions
TLB Translation Look-aside Buffer
UMA Uniform Memory Access
UPC Unified Parallel C
WAN Wide Area Network

Chapter 16

Index