Tutorials for High Performance Scientific Computing

The Art of HPC, volume 4

Victor Eijkhout

2022

Introduction to High-Performance Scientific Computing © Victor Eijkhout, distributed under a Creative Commons Attribution 3.0 Unported (CC BY 3.0) license and made possible by funding from The Saylor Foundation http://www.saylor.org.

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 `https://theartofhpc.com/` 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 scientific programming than that. A good part of being an effective practitioner of High Performance Scientific Computing is what can be called 'HPC Carpentry': a number of skills that are not scientific in nature, but that are still indispensable to getting your work done.

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 2.

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 3.
- 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 7 you will learn the use of *HDF5*.
- Visual output of program data is important, but too wide a topic to discuss here in great detail; chapter 8 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 9 considers how to code to prevent errors, and chapter 10 teaches you to debug code with *gdb*. Chapter 11 contains some information on how to write a program that uses more than one programming language.

				Exercises	
lesson	Topic	Book	Slides	in-class	homework
1	Unix	1	unix		1.39
2	Git	5			
3	Programming	2	programming	2.3	2.4
4	Libraries	2	programming		
5	Debugging	10			root code
6	Debugging	??			
7	LaTeX	14			14.13
8	Make	3			3.1, 3.2

Table 1: Timetable for the carpentry section of an HPC course.

Finally, chapter 14 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!

Table 1 gives a proposed lesson outline for the carpentry section of a course. The article by Wilson [21] is a good read on the thinking behind this 'HPC carpentry'.

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 this tutorial will work on any Unix-like platform, however, there is not just one Unix:

- Traditionally there are a few major flavors 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.
- The issue of command shells will be discussed below. This actually forms the most visible difference between different computers 'running Unix'.

1.1 Shells

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

- Most of this tutorial is focused on the *sh* or *bash* shell.
- For a variety of reasons, `bash`-like shells are to be preferred over the *csh* or *tcsh* shell. These will not be covered in this tutorial.
- Recent versions of the *Apple Mac OS* have the *zsh* as default. While this shell has many things in common with `bash`, we will point out differences explicitly.

1.2 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.

1.2.1 Looking at files

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

command	function
`ls`	list files or directories
`touch`	create new/empty file or update existing file
`cat > filename`	enter text into file
`cp`	copy files
`mv`	rename files
`rm`	remove files
`file`	report the type of file
`cat filename`	display file
`head,tail`	display part of a file
`less,more`	incrementally display a file

1.2.1.1 `ls`

Without any argument, the *`ls`* command gives you a listing of files that are in your present location.

Exercise 1.1. Type *`ls`*. Does anything show up?

Intended 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 behavior: no output does not mean that something went wrong, it only means that there is nothing to report.

Exercise 1.2. If the *`ls`* command shows that there are files, do `ls name` on one of those. By using an option, for instance `ls -s name` you can get more information about `name`.

Things to watch out for. If you mistype a name, or specify a name of a non-existing file, you'll get an error message.

The *`ls`* command can give you all sorts of information. In addition to the above `ls -s` for the size, there is `ls -l` for the 'long' listing. It shows (things we will get to later such as) ownership and permissions, as well as the size and creation date.

Remark 1 *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.*

1.2.1.2 `cat`

The *`cat`* command (short for 'concatenate') is often used to display files, but it can also be used to create some simple content.

Exercise 1.3. Type `cat > newfilename` (where you can pick any filename) and type some text. Conclude with `Control-d` on a line by itself: press the `Control` key and hold it while you press the `d` key. Now use *`cat`* to view the contents of that file: `cat newfilename`.

Intended outcome. In the first use of `cat`, text was appended 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.

Things to watch out for. 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 2 *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.*

1.2.1.3 man

The primary (though not always the most easily understood) source for unix commands is the *man* command, for 'manual'. The descriptions available this way are referred to as the *manual page*s.

Exercise 1.4. Read the man page of the `ls` command: `man ls`. Find out the size and the time / date of the last change to some files, for instance the file you just created.

Intended 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.

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.

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.*

1.2.1.4 touch

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

1.2.1.5 cp, mv, rm

The *cp* can be used for copying a file (or directories, see below): `cp file1 file2` makes a copy of `file1` and names it `file2`.

Exercise 1.5. Use `cp file1 file2` to copy a file. Confirm that the two files have the same contents. If you change the original, does anything happen to the copy?

Intended outcome. You should see that the copy does not change if the original changes or is deleted.

Things to watch out for. If `file2` already exists, you will get an error message.

A file can be renamed with *mv*, for 'move'.

Exercise 1.6. Rename a file. What happens if the target name already exists?

Files are deleted with `rm`. This command is dangerous: there is no undo. For this reason you can do `rm -i` (for 'interactive') which asks your confirmation for every file.

See section 1.2.4 for more aggressive removing.

1.2.1.6 `head, tail`

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

Exercise 1.7. 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.

Intended 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 *`file`*: it tells you what type of file you are dealing with.

Exercise 1.8. Do `file foo` for various 'foo': a text file, a directory, or the `/bin/ls` command.

Intended outcome. Some of the information may not be intelligible to you, but the words to look out for are 'text', 'directory', or 'executable'.

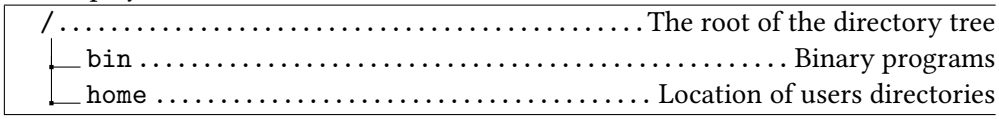
At this point it is advisable to learn to use a text *editor*, such as *emacs* or *vi*.

1.2.2 Directories

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

command	function
`ls`	list the contents of directories
`mkdir`	make new directory
`cd`	change directory
`pwd`	display present working directory

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 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 1.9. 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.

Intended 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 1.10. Make a new directory with *mkdir* `newdir` and view the current directory with `ls`.

Intended 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 1.11. 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?

Intended 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 1.12. 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.

Intended 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 1.13. Predict where you will be after `cd ./otherdir/..` and check to see if you were right.

Intended 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 1.14. What does `ls ..` do?

Intended 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 1.15. 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`.

Intended 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?

1.2.3 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 1.16. 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.

Intended outcome. 1. A file is only accessible by others if the surrounding folder is readable. Can you figure out how to do this? 2. 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 get the script executed?

In the three permission categories it is clear who 'you' and 'others' refer to. How about 'group'? We'll go into that in section 1.13.

Remark 4 *There are more obscure permissions. For instance the* setuid *bit declares that the program should run with the permissions of the creator, rather than the user executing it. This is useful for system utilities such* `passwd` *or* `mkdir`*, which alter the password file and the directory structure, for which* root privileges *are needed. Thanks to the setuid bit, a user can run these programs, which are then so designed that a user can only make changes to their own password entry, and their own directories, respectively. The setuid bit is set with chmod:* `chmod 4ugo file`.

1.2.4 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.

You can use `rm` with wildcards, but this can be dangerous.

```
rm -f foo    ## remove foo if it exists
rm -r foo    ## remove directory foo with everything in it
rm -rf foo/* ## delete all contents of foo
```

Zsh note. No match Removing with a wildcard `rm foo*` is an error of there are no such files. Set `setopt +o nomatch` to allow no matches to occur.

1.3 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 1.17. 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.

Intended 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`.

Things to watch out for. If the string you are looking for does not occur, `grep` will simply not output anything. Remember that this is standard behavior 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 1.2.4) 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 1.18. Construct the search strings for finding
- lines that start with an uppercase character, and
- lines that contain exactly one character.

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

Exercise 1.19. 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 1.20. 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.`

Intended 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.

1.3.1 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[1], 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.

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

1.4 Other useful commands: `tar`

The *`tar`* command stands for 'tape archive', that is, it was originally meant to package files on a tape. (The 'archive' part derives from the *`ar`* command.) These days, it's used to package files together for distribution on web sites and such: if you want to publish a library of hundreds of files this bundles them into a single file.

The two most common options are for

1. creating a tar file:
   ```
   tar fc package.tar directory_with_stuff
   ```
   pronounced 'tar file create', and
2. unpacking a tar file:
   ```
   tar fx package.tar
   # this creates the directory that was packaged
   ```
   pronounced 'tar file extract'.

Text files can often be compressed to a large extent, so adding the `z` compressiong for *`gzip`* is a good idea:

```
tar fcz package.tar.gz directory_with_stuff
tar fx package.tar.gz
```

Naming the 'gzipped' file `package.tgz` is also common.

1.5 Command execution

1.5.1 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 1.21. 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.

Intended 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 1.22. 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`.

Intended 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 1.23. 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?

Intended 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.

1.5.2 Command sequencing

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

1.5.2.1 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.

1.5.2.2 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 1.24. 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.

1.5.2.3 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 1.25. The way `wc` is used here, it prints the file name. Can you find a way to prevent that from happening?

There is another mechanism for out-of-order evaluation:

```
echo "There are $( cat Makefile | wc -l ) lines"
```

This mechanism makes it possible to nest commands, but for compatibility and legacy purposes backquotes may still be preferable when nesting is not neeeded.

1.5.2.4 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
```

1.5.3 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 script 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
```

1.5.4 Processes and jobs

`ps`	list (all) processes
`kill`	kill a process
`CTRL-c`	kill the foreground job
`CTRL-z`	suspect the foreground job
`jobs`	give the status of all jobs
`fg`	bring the last suspended job to the foreground
`fg %3`	bring a specific job to the foreground
`bg`	run the last suspended job in the background

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 1.26. 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?

Intended 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: if a program misbehaves you can *`kill`* it with

```
kill 123456
```

where `12345` is the process number.

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.

Processes that are started in a shell are known as *jobsjob (unix)*. In addition to the process number, they have a job number. We will now explore manipulating jobs.

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 1.27. 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`.

Intended 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 1.28. 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.

Intended 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 1.29. 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.

Intended 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`.

1.5.5 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 behavior 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.

1.6 Input/output 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.

1.6.1 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*.

1.6.2 Standard files

Unix has three standard files that handle input and output:

Standard file	Purpose
`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.

1.6.3 Output redirection

Just as with the input, you can redirect the output of your program. In the simplest case, `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 1.30. 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?

Intended 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`.

Sometimes you want to run a program, but ignore the output. For that, you can redirect your output to the system *null device*: */dev/null*.

```
yourprogram >/dev/null
```

Here are some useful idioms:

Idiom	Meaning
`program 2>/dev/null`	send only errors to the null device
`program >/dev/null 2>&1`	send output to dev-null, and errors to output Note the counterintuitive sequence of specifications!
`program 2>&1 | less`	send output and errors to `less`

1.7 Shell environment 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. While `PATH` is a built-in variable, you can also define your own variables, and use those in shell scripting.

Shell variables are roughly divided in the following categories:

- Variables that are specific to the shell, such as `HOME` or `PATH`.
- Variables that are specific to some program, such as `TEXINPUTS` for TEX/LATEX.
- Variables that you define yourself; see next.
- Variables that are defined by control structures such as *`for`*; see below.

You can see the full list of all variables known to the shell by typing *`env`*.

Remark 5 *This does not include variables you define yourself, unless you `export` them; see below.*

Exercise 1.31. Check on the value of the `PATH` variable by typing `echo $PATH`. Also find the value of `PATH` by piping `env` through `grep`.

We start by exploring the use of this dollar sign in relation to shell variables.

1.7.1 Use of shell variables

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

```
echo x
echo $x
x=5
echo x
echo $x
```

You see that the shell treats everything as a string, unless you explicitly tell it to take the value of a variable, by putting a dollar in front of the name. A variable that has not been previously defined will print as a blank string.

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

When you do the next exercise, it is good to bear in mind that the shell is a text based language.

Exercise 1.32. Type `a=5` on the commandline. Check on its value with the `echo` command.
Define the variable `b` to another integer. Check on its value.
Now explore the values of `a+b` and `$a+$b`, both by `echo`'ing them, or by first assigning them.

Intended outcome. The shell does not perform integer addition here: instead you get a string with a plus-sign in it. (You will see how to do arithmetic on variables in section 1.10.1.)

Things to watch out for. Beware not to have space around the equals sign; also be sure to use the dollar sign to print the value.

1.7.2 Exporting variables

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 executable: `chmod +x echob`.
3. Now call the script, preceding 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 1.8 you will see that the `for` construct also defines a variable; section 1.9.1 shows some more built-in variables that apply in shell scripts.

If you want to un-set an environment variable, there is the *`unset`* command.

1.8 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.

1.8.1 Conditionals

The *conditional* of the bash shell is predictably called *`if`*, and it 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
```

(The backquote is explained in section 1.5.2.3.) 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.

The syntax of this is finicky:

- *if* and *elif* are followed by a conditional, followed by a semicolon.
- The brackets of the conditional need to have spaces surrounding them.
- There is no semicolon after *then* of *else*: they are immediately followed by some command.

Exercise 1.33. Bash conditionals have an *elif* keyword. Can you predict the error you get from this:

```
if [ something ] ; then
  foo
else if [ something_else ] ; then
  bar
fi
```

Code it out and see if you were right.

1.8.2 Looping

In addition to conditionals, the shell has 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
```

(See the section 1.10 on expansion.) With this as a hint, write a loop that renames all your `.c` files to `.x` files.

The above construct loops over words, such as the output of `ls`. To do a numeric loop, use the command *seq*:

```
[shell:474] seq 1 5
1
2
3
4
5
```

Looping over a sequence of numbers then typically looks like

```
for i in `seq 1 ${HOWMANY}` ; do echo $i ; done
```

Note the *backtick*, which is necessary to have the `seq` command executed before evaluating the loop.

1.9 Scripting

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

1.9.1 How to execute scripts

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.

Zsh note. Bash scripts If you use the `zsh`, but you have `bash` scripts that you wrote in the past, they will keep working. The 'hash-bang' line determines which shell executes the script, and it is perfectly possible to have `bash` in your script, while using `zsh` for interactive use.

In order write scripts that you want to invoke from anywhere, people typically put them in a directory `bin` in their home directory. You would then add this directory to your *search path*, contained in *PATH*; see section 1.5.1.

1.9.2 Script arguments

You can invoke a shell script with options and arguments:

```
./my_script -a file1 -t -x file2 file3
```

You will now learn how to incorporate this functionality in your scripts.

First of all, all commandline arguments and options are available as variables `$1`,`$2` et cetera in the script, and the number of command line arguments is available as `$#`:

```
#!/bin/bash

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

Formally:

variable	meaning
`$#`	number of arguments
`$0`	the name of the script
`$1,$2,...`	the arguments
`$*,$@`	the list of all arguments

Exercise 1.34. 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.
Add a test to your script so that it will give a helpful message if you call it without any arguments.

The standard way to parse argument is using the *`shift`* command, which pops the first argument off the list of arguments. Parsing the arguments in sequence then involves looking at `$1`, shifting, and looking at the new `$1`.

Code:

```
// arguments.sh
while [ $# -gt 0 ] ; do
    echo "argument: $1"
    shift
done
```

Output
[code/shell] arguments:

```
./arguments.sh the quick "
    brown fox" jumps
argument: the
argument: quick
argument: brown fox
argument: jumps
```

Exercise 1.35. Write a script `say.sh` that prints its text argument. However, if you invoke it with

```
./say.sh -n 7 "Hello world"
```

it should be print it as many times as you indicated. Using the option `-u`:

```
./say.sh -u -n 7 "Goodbye cruel world"
```

should print the message in uppercase. Make sure that the order of the arguments does not matter, and give an error message for any unrecognized option.

The variables `$@` and `$*` have a different behavior with respect to double quotes. Let's say we evaluate `myscript "1 2" 3`, then

- Using `$*` is the list of arguments after removing quotes: `myscript 1 2 3`.
- Using `"$*"` is the list of arguments, with quotes removed, in quotes: `myscript "1 2 3"`.
- Using `"$@"` preserved quotes: `myscript "1 2" 3`.

1.10 Expansion

The shell performs various kinds of expansion on a command line, that is, replacing part of the commandline 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 1.5.2.3 above) is known as command substitution. It allows you to evaluate 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 evaluates `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
```

1.10.1 Arithmetic expansion

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
```

1.11 Startup files

In this tutorial you have seen several mechanisms for customizing the behavior 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 1.7) 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.

Popular things to do in a startup file are defining *aliases*:

```
alias grep='grep -i'
alias ls='ls -F'
```

and setting a custom commandline *prompt*.

Unfortunately, there are several startup files, and which one gets read is a complicated functions of circumstances. Here is a good common sense guideline[2]:

- 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 "$MYPATH" ]; then
    export MYPATH=1
    export PATH=$HOME/bin:$PATH
  fi
  ```

1.12 Shell interaction

Interactive use of Unix, in contrast to script writing (section 1.9), 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.

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

- 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 1.10). 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 recognize 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`.

1.13 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?

1.13.1 Groups

In section 1.2.3 you saw that there is a permissions category for 'group'. This allows you to open up files to your close collaborators, while leaving them protected from the wide world.

When your account is created, your system administrator will have assigned you to one or more groups. (If you admin your own machine, you'll be in some default group; read on for adding yourself to more groups.)

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.

Creating a new group, or adding a user to a group needs system privileges. To create a group:

```
sudo groupadd new_group_name
```

To add a user to a group:

```
sudo usermod -a -G thegroup theuser
```

1.13.2 The super user

Even if you own your machine, there are good reasons to work as much as possible from a regular user account, and use *root privileges* only when strictly needed. (The root account is also known as the *super user*.) If you have root privileges, you can also use that to 'become another user' and do things with their privileges, with the *sudo* ('superuser do') command.

- To execute a command as another user:
  ```
  sudo -u otheruser command arguments
  ```
- To execute a command as the root user:
  ```
  sudo command arguments
  ```
- Become another user:
  ```
  sudo su - otheruser
  ```
- Become the *super user*:
  ```
  sudo su -
  ```

1.14 Other systems: `ssh` and `scp`

No man is an island, and no computer is either. Sometimes you want to use one computer, for instance your laptop, to connect to another, for instance a supercomputer.

If you are already on a Unix computer, you can log into another with the 'secure shell' command *ssh*, a more secure variant of the old 'remote shell' command *rsh*:

```
ssh yourname@othermachine.otheruniversity.edu
```

where the yourname can be omitted if you have the same name on both machines.

To only copy a file from one machine to another you can use the 'secure copy' *scp*, a secure variant of 'remote copy' *rcp*. The scp command is much like cp in syntax, except that the source or destination can have a machine prefix.

To copy a file from the current machine to another, type:

```
scp localfile yourname@othercomputer:otherdirectory
```

where yourname can again be omitted, and otherdirectory can be an absolute path, or a path relative to your home directory:

```
# absolute path:
scp localfile yourname@othercomputer:/share/
# path relative to your home directory:
scp localfile yourname@othercomputer:mysubdirectory
```

Leaving the destination path empty puts the file in the remote home directory:

```
scp localfile yourname@othercomputer:
```

Note the colon at the end of this command: if you leave it out you get a local file with an 'at' in the name.

You can also copy a file from the remote machine. For instance, to copy a file, preserving the name:

```
scp yourname@othercomputer:otherdirectory/otherfile .
```

1.15 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, 8].

1.15.1 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 1.6 for more on output *redirection*.

- 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 1.3 for regular expressions.)
  You can also apply it on a numbered line:
  ```
  sed '25/s/foo/bar'
  ```

- The `a` and `i` commands are for 'append' and 'insert' respectively. They are somewhat strange in how they take their argument text: the command letter is followed by a backslash, with the insert/append text on the next line(s), delimited by the closing quote of the command.
  ```
  sed -e '/here/a\
  appended text
  ' -e '/there/i\
  inserted text
  ' -i file
  ```

- 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
  ```

1.15.2 `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 1.36. Build a command pipeline 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 1.37. 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.

1.16 Review questions

Exercise 1.38. 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 1.39. Pretend that you're a professor writing a script for homework submission: if a student invokes this script it copies the student file to some standard location.

```
submit_homework myfile.txt
```

For simplicity, we simulate this by making a directory `submissions` and two different files `student1.txt` and `student2.txt`. After

```
submit_homework student1.txt
submit_homework student2.txt
```

there should be copies of both files in the `submissions` directory. Start by writing a simple script; it should give a helpful message if you use it the wrong way.
Try to detect if a student is cheating. Explore the *`diff`* command to see if the submitted file is identical to something already submitted: loop over all submitted files and

1. First print out all differences.
2. Count the differences.
3. Test if this count is zero.

Now refine your test by catching if the cheating student randomly inserted some spaces.
For a harder test: try to detect whether the cheating student inserted newlines. This can not be done with *`diff`*, but you could try *`tr`* to remove the newlines.

Chapter 2

Compilers and libraries

2.1 File types in programming

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

2.1.1 Introduction to file types

Your file system has many files, and for purposes of programming we can roughly divide them into 'text file', which are readable to you, and 'binary files', which are not meaningfully readable to you, but which make sense to the computer.

The unix command *file* can tell you what type of file you are dealing with.

```
$$ file README.txt
README.txt: ASCII text
$$ mkdir mydir
$$ file mydir
mydir: directory
$$ which ls
```

This command can also tell you about binary files. Here the output differs by operating system.

```
$$ which ls
/bin/ls

# on a Mac laptop:
$$ file /bin/ls
/bin/ls: Mach-O 64-bit x86_64 executable

# on a Linux box
$$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64
```

Exercise 2.1. Apply the `file` command to sources for different programming language. Can you find out how `file` figures things out?

In figure 2.1 you find a brief summary of file types. We will now discuss them in more detail.

Text files	
Source	Program text that you write
Header	also written by you, but not really program text.
Binary files	
Object file	The compiled result of a single source file
Library	Multiple object files bundled together
Executable	Binary file that can be invoked as a command
Data files	Written and read by a program

Figure 2.1: Different types of files.

2.1.2 About 'text' files

Readable files are sometimes called *text files*; but this is not a concept with a hard definition. One not-perfect definition is that text files are *ascii* files, meaning files where every byte uses '7-bit ascii': the first bit of every byte is zero.

This definition is incomplete, since modern programming languages can often use *unicode*, at least in character strings. (For a tutorial on ascii and unicode, see chapter 6 of [9].)

2.1.3 Source versus program

There are two types of programming languages:

1. In an *interpreted language* you write human-readable source code and you execute it directly: the computer translates your source line by line as it encounters it.
2. In a *compiled language* your code whole source is first compiled to a program, which you then execute.

Examples of interpreted languages are *Python*, *Matlab*, *Basic*, *Lisp*. Interpreted languages have some advantages: often you can write them in an interactive environment that allows you to test code very quickly. They also allow you to construct code dynamically, during runtime. However, all this flexibility comes at a price: if a source line is executed twice, it is translated twice. In the context of this book, then, we will focus on compiled languages, using *C* and *Fortran* as prototypical examples.

So now you have a distinction between the readable source code, and the unreadable, but executable, program code. In this tutorial you will learn about the translation process from the one to the other. The program doing this translation is known as a *compiler*. This tutorial will be a 'user manual' for compilers, as it were; what goes on inside a compiler is a different branch of computer science.

2.1.4 Binary files

Binary files fall in two categories:

- executable code,
- data

Data files can be really anything: they are typically output from a program, and their format is often specific to that program, although there are some standards, such as *hdf5*. You get a binary data file if you write out the exact bytes of certain integers or floating point numbers, rather than a readable representation of that number.

Exercise 2.2. Why don't programs write their results to file in readable form?

Enrichment. How do you write/read a binary file in C and Fortran? Use the function `hexdump` to make sense of the binary file. Can you generate the file from Fortran, and read it from C? (Answer: yes, but it's not quite straightforward.) What does this tell you about binary data?

```
[linking:31] make binary
clang -o binary_write_c binary_write.c
./binary_write_c
clang -o binary_read_c binary_read.c
./binary_read_c
0 1 2 3 4 5 6 7 8 9
[linking:32] hexdump binary.out
0000000 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
0000010 04 00 00 00 05 00 00 00 06 00 00 00 07 00 00 00
0000020 08 00 00 00 09 00 00 00
0000028
```

```
// binary_write.c
  FILE *binfile;
  binfile = fopen("binarydata.out","wb");
  for (int i=0; i<10; i++)
    fwrite(&i,sizeof(int),1,binfile);
  fclose(binfile);
```

MISSING SNIPPET cbinwread

Fortran works differently: each *record*, that is, the output of each `Write` statement, has the record length (in bytes) before and after it.

```
[linking:68] make xbinary
gfortran -o binary_write_f binary_write.F90
./binary_write_f
hexdump binary.out
0000000 04 00 00 00 00 00 00 00 04 00 00 00 04 00 00 00
0000010 01 00 00 00 04 00 00 00 04 00 00 00 02 00 00 00
0000020 04 00 00 00 04 00 00 00 03 00 00 00 04 00 00 00
0000030 04 00 00 00 04 00 00 00 04 00 00 00 04 00 00 00
0000040 05 00 00 00 04 00 00 00 04 00 00 00 06 00 00 00
0000050 04 00 00 00 04 00 00 00 07 00 00 00 04 00 00 00
0000060 04 00 00 00 08 00 00 00 04 00 00 00 04 00 00 00
0000070 09 00 00 00 04 00 00 00
```

```
// binary_write.F90
Open(Unit=13,File="binarydata.out",Form="
    unformatted")
  do i=0,9
    write(13) i
  end do
  Close(Unit=13)
```

In this tutorial you will mostly be concerned with executable binary files. We then distinguish between:

- program files, which are executable by themselves;

- object files, which are like bit of programs; and
- library files, which combine object files, but are not executable.

Object files come from the fact that your source is often spread over multiple source files, and these can be compiled separately. In this way, an *object file*, is a piece of an executable: by itself it does nothing, but it can be combined with other object files to form an executable.

If you have a collection of object files that you need for more than one program, it is usually a good idea to make a *library*: 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.

2.2 Simple compilation

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

2.2.1 Compilers

Your main tool for turning source into a program is the *compiler*. Compilers are specific to a language: you use a different compiler for C than for Fortran. You can also have two compilers for the same language, but from different 'vendors'. For instance, while many people use the open source *gcc* or *clang* compiler families, companies like *Intel* and *IBM* offer compilers that may give more efficient code on their processors.

2.2.2 Compile a single file

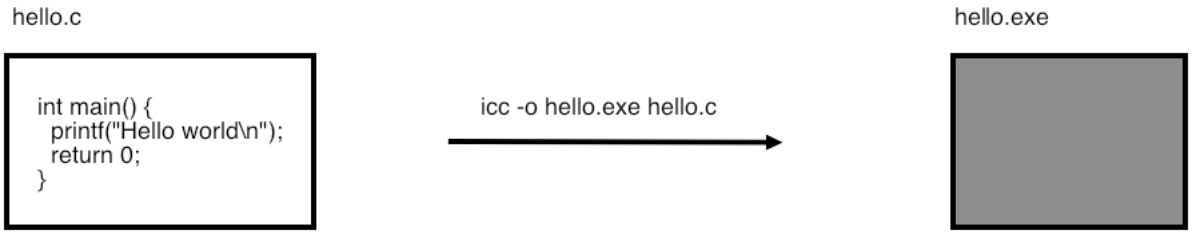


Figure 2.2: Compiling a single source file.

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

One file: `hello.c`

Compile this program with your favorite compiler; we will use `gcc` in this tutorial, but substitute your own as desired.

TACC note. On TACC clusters, the Intel compiler `icc` is preferred.

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
```

This process is illustrated in figure 2.2.

2.2.3 Multiple files: compile and link

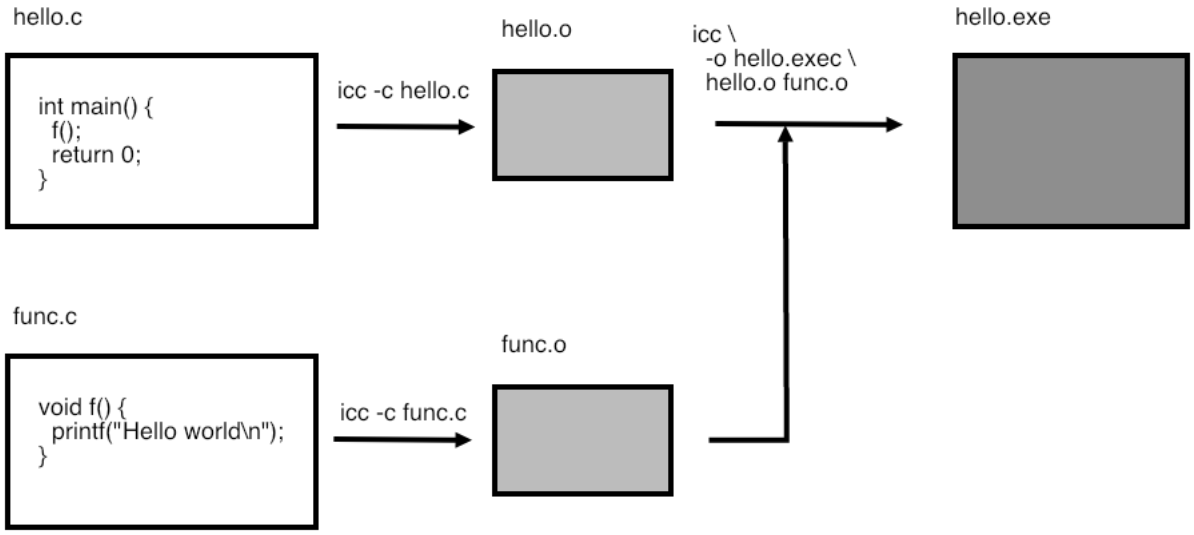


Figure 2.3: Compiling a program from multiple source files.

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

Main program: `fooprog.c`

```
// fooprog.c
extern void bar(char*);

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

Subprogram: `foosub.c`

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

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

Output
[code/compile] makeoneprogram:

```
clang -o oneprogram fooprog.c
     foosub.c
./oneprogram
hello world
```

However, you can also do it in steps, compiling each file separately and then linking them together. This is illustrated in figure 2.3.

Output
[code/compile]
makeseparatecompile:

```
clang -g -O2 -o oneprogram
    fooprog.o foosub.o
./oneprogram
hello world
```

The `-c` option tells the compiler to compile the source file, giving an *object file*. The third command then 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 3, for a way of dealing with this.)

Exercise 2.3.

Exercise for separate compilation. Structure:

Main program: `fooprog.c`

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

//codesnippet foomain
extern void bar(char*);

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

Subprogram: `foosub.c`

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

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

Add a second subroutine in a second file.

- Compile in one:
  ```
  icc -o program fooprog.c foosub.c
  ```
- Compile in steps:
  ```
  icc -c fooprog.c
  icc -c foosub.c
  icc -o program fooprog.o foosub.o
  ```

What files are being produced each time?
Can you write a shell script to automate this?

2.2.4 Looking into binary files: `nm`

Most of a binary file consists of the same instructions that you coded in C or Fortran, just in machine language, which is much harder to understand. Fortunately, you don't need to look at machine language often. What often interests you about object files is what functions are defined in it, and what functions are used in it.

For this, we use the *`nm`* command.

Each object file defines some routine names, and uses some others that are undefined in it, but that will be defined in other object files or system libraries. Use the *`nm`* command to display this:

```
[c:264] nm foosub.o
0000000000000000 T _bar
                 U _printf
```

Lines with `T` indicate routines that are defined; lines with `U` indicate routines that are used but not define in this file. In this case, `printf` is a system routine that will be supplied in the linker stage.

Sometimes you will come across *stripped binary* file, and *`nm`* will report `No symbols`. In that case `nm -D` may help, which displays 'dynamic symbols'.

2.2.5 Compiler options and optimizations

Above you already saw some *compiler options*:

- Specifying `-c` tells the compiler to only compile, and not do the linking stage; you would do this in case of separate compilation.
- The option `-o` gives you the opportunity to specify the name of the output file; without it, the default name of an executable is `a.out`.

There are many other options, some of them a *de facto* standard, and others specific to certain compilers.

2.2.5.1 Symbol table inclusion

The `-g` option tells the compiler to include the *symbol table* in the binary. This allows you to use an interactive debugger (section 10) since it relates machine instructions to lines of code, and machine addresses to variable names.

2.2.5.2 Optimization level

Compilers can apply various levels of *optimization* to your code. The typical optimization levels are specified as `-O0` 'minus-oh-zero', `-O1`, `-O2`, `-O3`. Higher levels will typically give faster execution, as the compiler does increasingly sophisticated analysis on your code.

The following is a fairly standard set of options:

```
icc -g -O2 -c myfile.c
```

As an example, let's look at *Given's rotations*:

```
// rotate.c
void rotate(double *x,double *y,double alpha) {
  double x0 = *x, y0 = *y;
  *x = cos(alpha) * x0 - sin(alpha) * y0;
  *y = sin(alpha) * x0 + cos(alpha) * y0;
  return;
}
```

Run with optimization level 0,1,2,3 we get:

```
Done after 8.649492e-02
Done after 2.650118e-02
Done after 5.869865e-04
Done after 6.787777e-04
```

Exercise 2.4. From level zero to one we get (depending on the run) an improvement of 2× to 3×. Can you find an obvious factor of two?
Use the optimization report facility of your compiler to see what other optimizations are applied. One of them is a good lesson in benchmark design!

Many compilers can generate a report of what optimizations they perform.

compiler	reporting option
clang	`-Rpass=.*`
gcc	`-fopt-info`
intel	`-qopt-report`

Generally, optimizations leave the semantics of your code intact. (Makes kinda sense, not?) However, at higher levels, usually level 3, the compiler is at liberty to make transformations that are not legal according to the language standard, but that in the majority of cases will still give the right outcome. For instance, the C language specifies that arithmetic operations are evaluated left-to-right. Rearranging arithmetic expressions is usually safe, but not always. Be careful when applying higher optimization levels!

2.3 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.

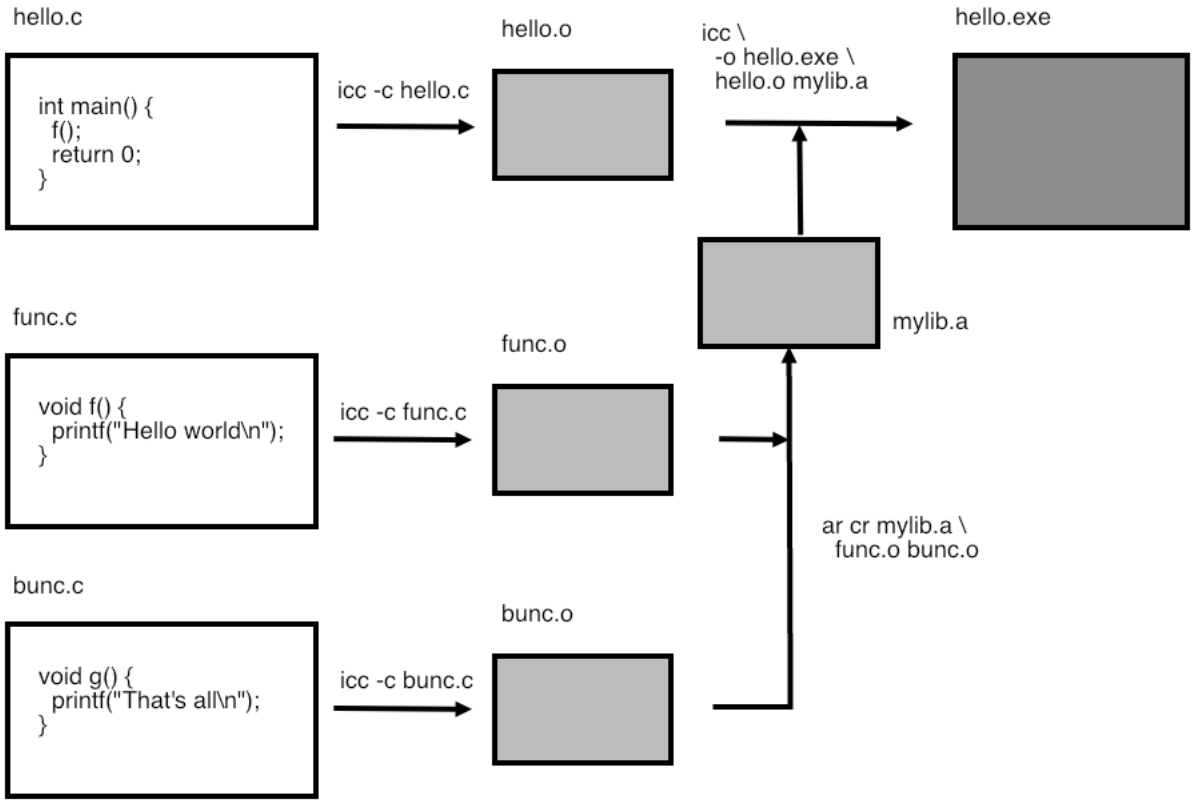


Figure 2.4: Compiling a single source file.

2.3.1 Static libraries

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.

The use of a library to build a program is illustrated in figure 2.4.

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.

Output
[code/compile] staticprogram:

```
==== Use of static library
    ====

for o in foosub.o ; do \
          ar cr libs/libfoo.a
     ${o} ; \
        done
clang  -o staticprogram
    fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff
      49984 Nov 28 12:00
    staticprogram

.. running:

hello world
```

The *nm* command tells you what's in the library, just like it did with object files, but now it also tells you what object files are in the library:

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

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

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

Output
[code/compile] staticprogram:

```
==== Use of static library
    ====

for o in foosub.o ; do \
          ar cr libs/libfoo.a
     ${o} ; \
        done
clang  -o staticprogram
    fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff
      49984 Nov 28 12:00
    staticprogram

.. running:

hello world
```

2.3.2 Shared libraries

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:

Output
[code/compile] makedynamiclib:

```
%%%%
Demonstration: making shared
    library
%%%%
clang -o libs/libfoo.so -
    shared foosub.o
```

You can again use *nm*:

```
%% 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 needs the information where the library is to be found at execution time. One way to do this is with `LD_LIBRARY_PATH`:

Output
[code/compile] dynamicprogram:

```
Linking to shared library

clang -o libs/libfoo.so -
    shared foosub.o
clang -o dynamicprogram
    fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff
      49720 Nov 28 12:00
    dynamicprogram

.. note the size of the
    program

-rwxr-xr-x  1 eijkhout  staff
      49720 Nov 28 12:00
    dynamicprogram

.. note unresolved link to a
    library

otool -L dynamicprogram |
    grep libfoo
        libs/libfoo.so (
    compatibility version
    0.0.0, current version
    0.0.0)

.. running by itself:

clang -o libs/libfoo.so -
    shared foosub.o
clang -o dynamicprogram
    fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff
      49720 Nov 28 12:00
    dynamicprogram
hello world

.. note resolved with
    LD_LIBRARY_PATH

LD_LIBRARY_PATH=${
    LD_LIBRARY_PATH}:./libs
    otool -L dynamicprogram |
     grep libfoo
        libs/libfoo.so (
    compatibility version
    0.0.0, current version
    0.0.0)

.. running with updated
    library path:
clang -o libs/libfoo.so -
    shared foosub.o
clang -o dynamicprogram
    fooprog.o -Llibs -lfoo
-rwxr-xr-x  1 eijkhout  staff
      49720 Nov 28 12:00
    dynamicprogram
```

Another solution is to have the path be included in the executable:

```
%% 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.

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

Chapter 3

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 behavior, for instance on the various flavors 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 [14].

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 3.6.

3.1 A simple example

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

The files for this section can be found in the repository.

3.1.1 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 prerequisite 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.

3.1.2 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.

3.1.3 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`.

3.2 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.

3.2.1 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 happens 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 3.2.2) 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.

3.2.2 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.

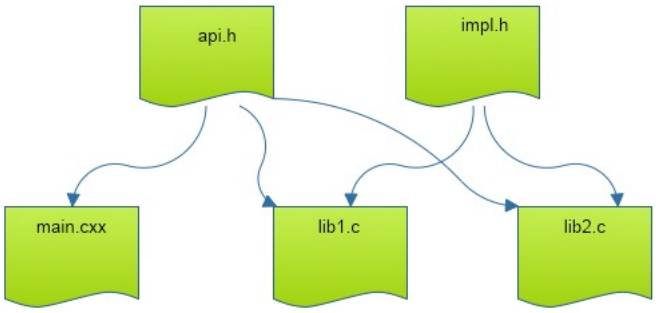


Figure 3.1: File structure with main program and two library files.

Exercise 3.1. Write a makefile for the following structure:

- There is one main file `libmain.cxx`, and two library files `libf.cxx` `libg.cxx`;
- There is a header file `libapi.h` that gives the prototypes for the functions in the library files;
- There is a header file `libimpl.h` that gives implementation details, only to be used in the library files.

This is illustrated in figure 3.1.

Here is how you can test it:

Changing a source file only recompiles that files:

```
clang++ -c libf.cxx
clang++ -o main libmain.o libf.o libg.o
```

Changing the implementation header only recompiles the library:

```
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o libg.o
```

Changing the `libapi.h` recompiles everything:

```
clang++ -c libmain.cxx
clang++ -c libf.cxx
clang++ -c libg.cxx
clang++ -o main libmain.o libf.o libg.o
```

For Fortran we don't have header files so we use *modules* everywhere; figure 3.3. If you know how to use *submodules*, a *Fortran2008* feature, you can make the next exercise as efficient as the C version.

Exercise 3.2. Write a makefile for the following structure:

- There is one main file `libmain.f90`, that uses a module `api.f90`;
- There are two low level modules `libf.f90` `libg.f90` that are used in `api.f90`.

Source file `mainprog.cxx`:

```
#include <cstdio>
#include "api.h"

int main() {
    int n = f() + g();
    printf("%d\n",n);
    return 0;
}
```

Source file `libf.cxx`:

```
#include "impl.h"
#include "api.h"

int f() {
    struct impl_struct foo;
    return 1;
};
```

Source file `libg.cxx`:

```
#include "impl.h"
#include "api.h"

int g() {
    struct impl_struct foo;
    return 2;
};
```

Header file `api.h`:

```
int f();
int g();
```

Header file `impl.h`:

```
struct impl_struct {};
```

Figure 3.2: Source files for exercise 3.1.

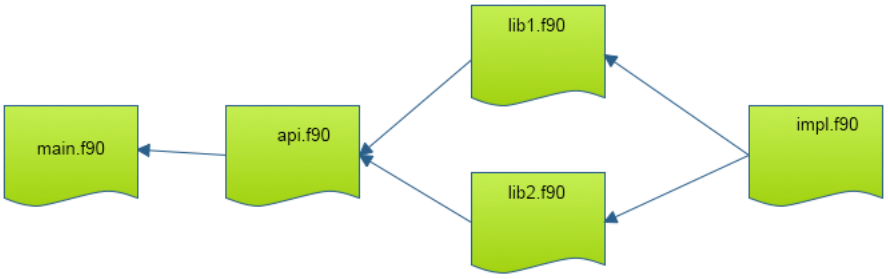


Figure 3.3: File structure with main program and two library files.

If you use modules, you'll likely be doing more compilation than needed. For the optimal solution, use submodules.

3.2.3 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}}
```

3.2.4 More functions

GNU make has more function that you can call inside the makefile. Some examples:

```
HOSTNAME := $(shell hostname -f)
SOURCES  := $(wildcard *.c)
OBJECTS  := $(patsubst %.c,%.o,${SOURCES})
RECURSIVE := $(foreach d,${DIRECTORIES},$(wildcard ${d}/*.c))
```

For the full list see https://www.gnu.org/software/make/manual/html_node/Functions.html.

3.2.5 Conditionals

There are various ways of making the behavior 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 "${HOME}" "/home/thisisme"
  # case where the executing user is me
else ifeq "${HOME}" "/home/buddyofmine"
  # case for other user
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`.

3.3 Miscellania

3.3.1 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, you use the *`.PHONY`* keyword:

```
.PHONY : clean
```

Most of the time, the makefile will actually work fine without this declaration, but the main benefit of declaring a target to be phony is that the *Make* rule will still work, even if you have a file (or folder) named `clean`.

3.3.2 Directories

It's a common strategy to have a directory for temporary material such as object files. So you would have a rule

```
obj/%.o : %.c
    ${CC} -c $< -o $@
```

and to remove the temporaries:

```
clean ::
    rm -rf obj
```

This raises the question how the `obj` directory is created. You could do:

```
obj/%.o : %.c
    mkdir -p obj
    ${CC} -c $< -o $@
```

but a better solution is to use *order-only prerequisite*s exist.

```
obj :
    mkdir -p obj
obj/%.o : %.c | obj
    ${CC} -c $< -o $@
```

This only tests for the existence of the object directory, but not its timestamp.

3.3.3 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[2]:

```
.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.

3.3.4 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.

3.4 Shell scripting in a Makefile

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

2. 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.

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 1.5.2, 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:

```
#### This Script Has An ERROR!
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.)

3.5 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.

3.5.1 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[3], 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

3. 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`.

```
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
```

3.6 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 exit 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 4

The Cmake build system

4.1 CMake as build system

CMake is a general build system that uses other systems such as *Make* as a back-end. The general workflow is:

1. The configuration stage. Here the *`CMakeLists.txt`* file is parsed, and `build` directory populated. This typically looks like:
   ```
   mkdir build
   cd build
   cmake <source location>
   ```
   Some people create the build directory in the source tree, in which case the cmake command is
   ```
   cmake ..
   ```
   Other put the build directory next to the source, in which case:
   ```
   cmake ../src_directory
   ```

2. The build stage. Here the installation-specific compilation in the build directory is performed. With Make as the 'generator' this would be
   ```
   cd build
   make
   ```
   but more generally
   ```
   cmake --build <build directory>
   ```
   Alternatively, you could use generators such as *ninja*, *Visual Studio*, or *XCode*:
   ```
   cmake -G ninja
     ## the usual arguments
   ```

3. The install stage. This can move binary files to a permanent location, such as putting library files in `/url/lib`:
   ```
   make install
   ```
   or

General directives	
`cmake_minimum_required`	specify minimum cmake version
`project`	name and version number of this project
`install`	specify directory where to install targets
Project building directives	
`add_executable`	specify executable name and source files for it
`add_library`	specify library name and files to go into it
`add_subdirectory`	specify subdirectory where cmake also needs to run
`target_link_libraries`	specify executable and libraries to link into it
`target_include_directories`	specify include directories, privately or publicly
`find_package`	other package to use in this build
Utility stuff	
`target_compile_options`	literal options to include
`target_compile_features`	things that will be translated by cmake into options
`target_compile_definitions`	macro definitions to be set private or publicly
`file`	define macro as file list
`message`	Diagnostic to print, subject to level specification
Control	
`if() else() endif()`	conditional

Table 4.1: Cmake commands.

```
cmake --install <build directory>
```

The directory structure is illustrated in figure 4.1.

Figure 4.1: In-source (left) and out-of-source (right) build schemes.

4.1.1 Target philosophy

Modern Cmake works through declaring targets and their requirements. For requirements during building:

```
target_some_requirement( the_target PRIVATE the require ments )
```

Usage requirements:

```
target_some_requirement( the_target PUBLIC the require ments )
```

4.1.2 Languages

Cmake is largely aimed at C++, but it easily supports C as well. For *Fortran* support, first to

```
enable_language(Fortran)
```

Note that capitalization: this also holds for all variables such as `CMAKE_Fortran_COMPILER`.

4.1.3 Script structure

Commands learned in this section	
`cmake_minimum_required`	declare minimum required version for this script
`project`	declare a name for this project

CMake is driven by the `CMakeLists.txt` file. This needs to be in the root directory of your project. (You can additionally have files by that name in subdirectories.)

Since cmake has changed quite a bit over the years, and is still evolving, it is a good idea to start each script with a declaration of the (minimum) required version:

```
cmake_minimum_required( VERSION 3.12 )
```

You can query the version of your cmake executable:

```
$ cmake --version
cmake version 3.19.2
```

You also need to declare a project name and version, which need not correspond to any file names:

```
project( myproject VERSION 1.0 )
```

4.2 Examples cases

4.2.1 Executable from sources

Commands learned in this section	
`add_executable`	declare an executable and its sources
`install`	indicate location where to install this project
PROJECT_NAME	macro that expands to the project name

If you have a project that is supposed to deliver an executable, you declare in your `CMakeLists.txt`:

```
add_executable( myprogram program.cxx )
```

Often, the name of the executable is the name of the project, so you'd specify:

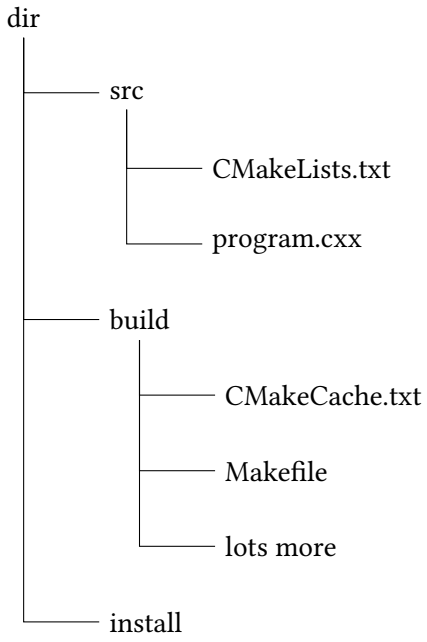
```
add_executable( ${PROJECT_NAME} program.cxx )
```

In order to move the executable to the install location, you need a clause

```
install( TARGETS myprogram DESTINATION . )
```

Without the `DESTINATION` clause, a default `bin` directory will be created; specifying `DESTINATION foo` will put the program in a `foo` directory.

In the figure on the right we have also indicated the `build` directory, which from now on we will not show again. It contains automatically generated files that are hard to decyper.or debug. Yes, there is a `Makefile`, but even for simple projects this is too complicated to debug by hand if your cmake installation misbehaves.



Here is the full `CMakeLists.txt`:

```
cmake_minimum_required( VERSION 3.12 )
project( singleprogram VERSION 1.0 )

add_executable( program program.cxx )
install( TARGETS program DESTINATION . )
```

4.2.2 Making libraries

Commands learned in this section	
`add_library`	declare a library and its sources
`target_link_libraries`	indicate that the library belong with an executable

If there is only one source file, this is all you need. However, often you will build libraries. You declare those with an `add_library` clause:

```
add_library( auxlib aux.cxx aux.h )
```

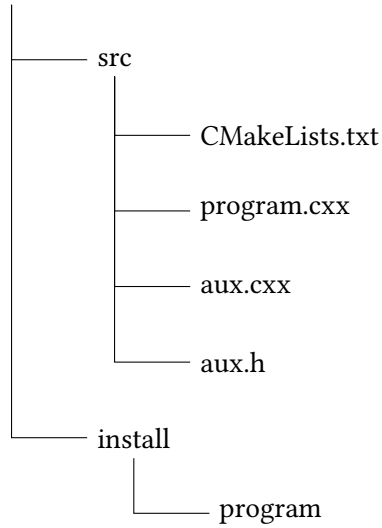
Next, you need to link that library into the program:

```
target_link_libraries( program PRIVATE auxlib )
```

The `PRIVATE` clause means that the library is only for purposes of building the executable. (Use `PUBLIC` to have the library be included in the installation; we will explore that in the next section.)

The full `CMakeLists.txt`:

```
cmake_minimum_required( VERSION 3.12 )
project( cmakeprogram VERSION 1.0 )

add_executable( program program.cxx )
add_library( auxlib STATIC
             aux.cxx aux.h )
target_link_libraries( program PRIVATE auxlib )
install( TARGETS program DESTINATION . )
```



Note that private shared libraries make no sense, as they will give runtime unresolved references.

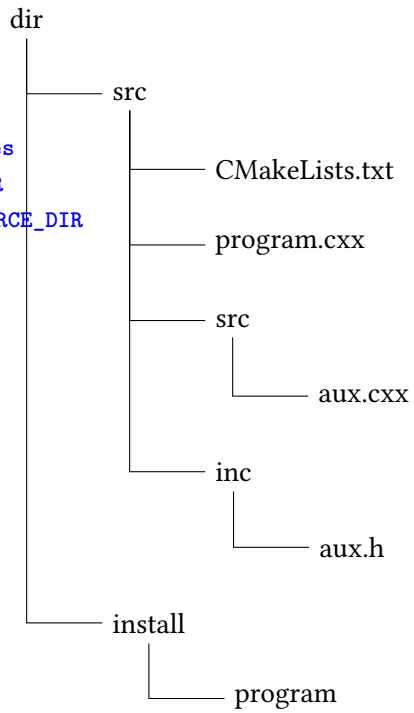
4.2.3 Finding subdirectories for the build

Commands learned in this section	
`target_include_directories`	indicate include directories needed
`target_sources`	specify more sources for a target
`CMAKE_CURRENT_SOURCE_DIR`	variable that expands to the current directory

If your program needs to be built using header files from a subdirectory, meaning:

```
cc -c yourprogram.c -I./some/dir
```

you can specify that include path with `target_include_directories`. It is best to make such paths relative to `CMAKE_CURRENT_SOURCE_DIR`, or the source root `CMAKE_SOURCE_DIR`, or equivalently `PROJECT_SOURCE_DIR`



Usually, when you start making such directory structure, you will also have sources in subdirectories. If you only need to compile them into the main executable, you could list them into a variable

```
set( SOURCES program.cxx src/aux.cxx )
```

and use that variable. However, this is deprecated practice; it is recommended to use `target_sources`:

```
target_sources( program PRIVATE src/aux.cxx )
```

Complete cmake file:

```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

add_executable( program program.cxx )
target_sources( program PRIVATE src/aux.cxx )
target_include_directories(
        program PRIVATE
        "${CMAKE_CURRENT_SOURCE_DIR}/inc" )

install( TARGETS program DESTINATION . )
```

4.2.4 Making a library

Commands learned in this section	
`add_library`	declare a library and its sources
`target_link_libraries`	declare a library to link into the target
`SHARED`	indicated to make shared libraries

In order to create a library we use `add_library`, and we link it into the target program with `target_link_libraries`.

By default the library is build as a static `.a` file, but adding

```
add_library( auxlib SHARED aux.cxx aux.h )
```

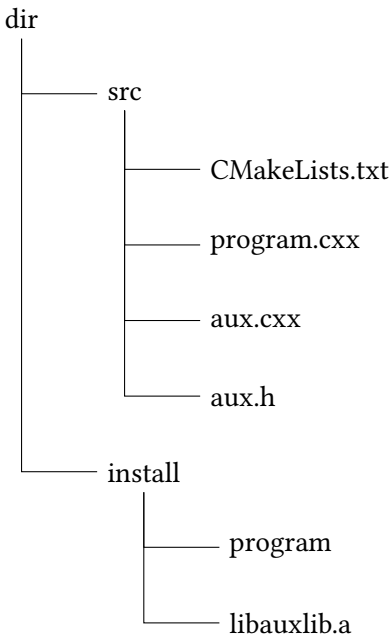
or adding a runtime flag

```
cmake -D BUILD_SHARED_LIBS=TRUE
```

changes that to a shared `.so` type.

The full cmake file:

```
cmake_minimum_required( VERSION 3.12 )
project( cmakeprogram VERSION 1.0 )

add_executable( program program.cxx )
add_library( auxlib
             aux.cxx aux.h )
## target_sources( aux.h PUBLIC FILE_SET headers )
target_link_libraries( program PUBLIC auxlib )
install( TARGETS program auxlib DESTINATION . )
```

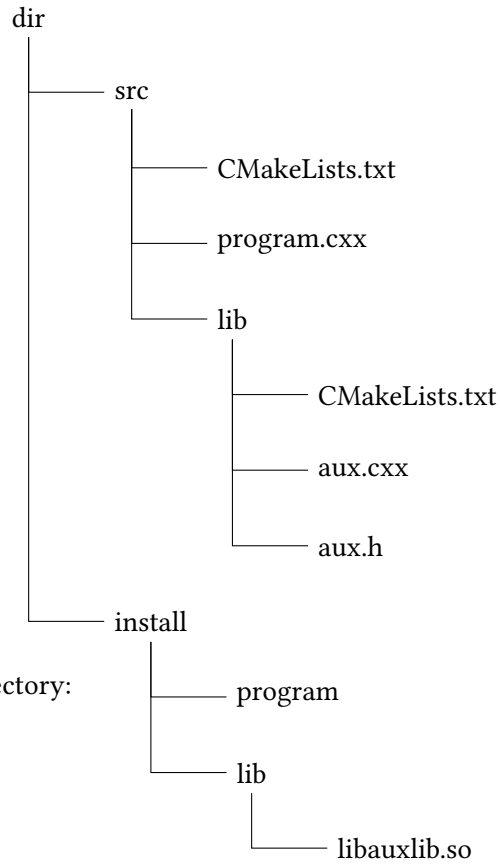


4.2.5 Sub directories in the installation

Commands learned in this section	
`add_subdirectory`	declare a subdirectory where cmake needs to be run
`CMAKE_CURRENT_BINARY_DIR`	
`LIBRARY_OUTPUT_PATH`	

If your sources are spread over multiple directories, there needs to be a `CMakeLists.txt` file in each, and you need to declare the existence of those directories. Let's start with the obvious choice of putting library files in a `lib` directory:

```
add_subdirectory( lib )
```



For instance, a library directory would have a `CMakeLists.txt` file:

```
cmake_minimum_required( VERSION 3.14 )
project( auxlib )

add_library( auxlib SHARED
             aux.cxx aux.h )
target_include_directories(
        auxlib PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}" )
install( TARGETS auxlib DESTINATION lib )
install( FILES aux.h DESTINATION include )
```

to build the library file from the sources indicated, and to install it in a `lib` subdirectory.

We also add a clause to install the header files in an include directory:

```
install( FILES aux.h DESTINATION include )
```

For installing multiple files, use

```
install(DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
    DESTINATION ${LIBRARY_OUTPUT_PATH}
    FILES_MATCHING PATTERN "*.h")
```

One problem is to tell the executable where to find the library. For this we use the *rpath* mechanism. By default, Cmake sets it so that the executable in the build location can find the library. If you use a non-trivial install prefix, the following lines work:

```
set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )
```

Note that these have to be specified before the target.

The whole file:

```
cmake_minimum_required( VERSION 3.14 )
project( cmakeprogram VERSION 1.0 )

set( CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib" )
set( CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE )

add_executable( program program.cxx )
add_subdirectory( lib )
target_include_directories(
        auxlib PUBLIC "${CMAKE_CURRECT_SOURCE_DIR}" )
target_link_libraries(
        program PUBLIC auxlib )

install( TARGETS program DESTINATION . )
```

4.2.6 Finding and using external packages

If your program depends on other libraries, there is a variety of ways to let cmake find them.

4.2.6.1 Cmake commandline options

You can indicate the location of your external library explicitly on the commandline; see section 4.3.3 for the basics.

```
cmake -D OTHERLIB_INC_DIR=/some/where/include
      -D OTHERLIB_LIB_DIR=/somewhere/lib
```

Example cmake file:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

# with environment variables
# set( AUX_INCLUDE_DIR $ENV{TACC_AUX_INC} )
# set( AUX_LIBRARY_DIR $ENV{TACC_AUX_LIB} )

# with cmake -D options
option( AUX_INCLUDE_DIR "include dir for auxlib" )
option( AUX_LIBRARY_DIR   "lib dir for auxlib" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${AUX_INCLUDE_DIR} )
target_link_libraries( program PUBLIC auxlib )
target_link_directories(
        program PUBLIC
        ${AUX_LIBRARY_DIR} )
install( TARGETS program DESTINATION . )
```

4.2.6.2 *Package finding through 'find library' and 'find package'*

Commands learned in this section	
`find_library`	find a library with a `FOOConfig.cmake` file
`CMAKE_PREFIX_PATH`	location for `FOOConfig.cmake` files
`find_package`	find a library with a `FindFOO` module
`CMAKE_MODULE_PATH`	location for `FindFOO` modules

The `find_package` command looks for files with a name `FindXXX.cmake`, which are searched on the `CMAKE_MODULE_PATH`. Unfortunately, the working of `find_package` depend somewhat on the specific package. For instance, most packages set a variable *FooFound* that you can test

```
find_package( Foo )
if ( FooFound )
    # do something
else()
    # throw an error
endif()
```

Some libraries come with a `FOOConfig.cmake` file, which is searched on the `CMAKE_PREFIX_PATH` through `find_library`. If it is found, you can test the variable it is supposed to set:

```
find_library( FOOLIB foo )
if (FOOLIB)
  target_link_libraries( myapp PRIVATE ${FOOLIB} )
else()
  # throw an error
endif()
```

4.2.6.3 *OpenMP*

```
find_package(OpenMP)
if(OpenMP_C_FOUND) # or CXX
else()
        message( FATAL_ERROR "Could not find OpenMP" )
endif()
# for C:
add_executable( ${program} ${program}.c )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_C )
# for C++:
add_executable( ${program} ${program}.cxx )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_CXX)
# for Fortran
enable_language(Fortran)
# test: if( OpenMP_Fortran_FOUND )
add_executable( ${program} ${program}.F90 )
target_link_libraries( ${program} PUBLIC OpenMP::OpenMP_Fortran )
```

4.2.6.4 *MKL*

Intel compiler installations come with cmake support: there is a file `MKLConfig.cmake`.

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

## https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/
    top/getting-started/cmake-config-for-onemkl.html

find_package( MKL CONFIG REQUIRED )

add_executable( program program.cxx )
target_compile_options(
        program PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_COMPILE_OPTIONS> )
target_include_directories(
        program PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_INCLUDE_DIRECTORIES> )

install( TARGETS program DESTINATION . )
```

4.2.7 Use of other packages through 'pkg config'

These days, many package support the *pkgconfig* mechanism.

1. Suppose you have a library `mylib`, installed in `/opt/local/mylib`.
2. If `mylib` supports `pkgconfig`, there is most likely a path `/opt/local/mylib/lib/pkgconfig`, containing a file `mylib.pc`.
3. Add the path that contains the *.pc* file to the *PKG_CONFIG_PATH* environment variable.

Cmake is now able to find `mylib`:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( MYLIBRARY REQUIRED mylib )
```

This defines variables

```
MYLIBRARY_INCLUDE_DIRS
MYLIBRARY_LIBRARY_DIRS
MYLIBRARY_LIBRARIES
```

which you can then use in the `target_include_directories` command.

4.2.7.1 *Example with pkg config: fmtlib*

In the following example, we use the *fmtlib*. The main cmake file:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes: ${FMTLIB_INCLUDE_DIRS}" )

add_executable( program program.cxx )
```

```
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

install( TARGETS program DESTINATION . )
```

We continue using the *fmtlib* library, but now the generated library also has references to this library, so we use `target_link_directories` and `target_link_library`.

Main file:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
message( STATUS "fmtlib includes : ${FMTLIB_INCLUDE_DIRS}" )
message( STATUS "fmtlib lib dirs : ${FMTLIB_LIBRARY_DIRS}" )
message( STATUS "fmtlib libraries: ${FMTLIB_LIBRARIES}" )

add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${FMTLIB_INCLUDE_DIRS})

add_subdirectory( prolib )
target_link_libraries( program PUBLIC prolib )

install( TARGETS program DESTINATION . )
```

Library file:

```
project( prolib )

add_library( prolib SHARED aux.cxx aux.h )
target_include_directories(
        prolib PUBLIC
        ${FMTLIB_INCLUDE_DIRS})
target_link_directories(
        prolib PUBLIC
        ${FMTLIB_LIBRARY_DIRS})
target_link_libraries(
        prolib PUBLIC fmt )
```

4.2.7.2 *Common packages: MPI*

This cmake setup searches for `mpich.pc`:

```
cmake_minimum_required( VERSION 3.12 )
project( pkgconfiglib VERSION 1.0 )

find_package( PkgConfig REQUIRED )
pkg_check_modules( MPI REQUIRED mpich )
message( STATUS "mpi includes: ${MPI_INCLUDE_DIRS}" )
```

```
add_executable( program program.cxx )
target_include_directories(
        program PUBLIC
        ${MPI_INCLUDE_DIRS} )
target_link_directories(
        program PUBLIC
        ${MPI_LIBRARY_DIRS} )
target_link_libraries(
        program PUBLIC mpich )

install( TARGETS program DESTINATION . )
```

4.3 Customizing the compilation process

Commands learned in this section	
`target_compile_features`	compiler-independent specification of compile flags
`target_compile_definitions`	pre-processor flags
`option`	query commandline option

4.3.1 Custominzing the compilation

It's probably a good idea to tell cmake explicitly what compiler you are using, otherwise it may find some default gcc version that came with your system. Use the variables `CMAKE_CXX_COMPILER`, `CMAKE_C_COMPILER`, `CMAKE_LINKER`.

4.3.1.1 Universal flags

Certain flags have a universal meaning, but compiler-dependent realization. For instance, to specify the C++ standard:

```
target_compile_features(arias_demo PRIVATE cxx_std_17)
```

The variable `CMAKE_CXX_COMPILE_FEATURES` contains the list of all features you can set.

Optimization flags can be set by specifying the `CMAKE_BUILD_TYPE`:

- *Debug* corresponds to the `-g` flag;
- *Release* corresponds to `-O3 -DNDEBUG`;
- *MinSizeRel* corresponds to `-Os -DNDEBUG`
- *RelWithDebInfo* corresponds to `-O2 -g -DNDEBUG`.

This variable will often be set from the commandline:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

4.3.1.2 *Custom*

Set the variable `CMAKE_CXX_FLAGS` or `CMAKE_C_FLAGS`; also `CMAKE_LINKER_FLAGS` (but see section 4.2.5 for the popular *rpath* options.)

4.3.2 Macro definitions

Cmake can provide macro definitions:

```
target_compile_definitions
  ( programname PUBLIC
    HAVE_HELLO_LIB=1 )
```

and your source could test these:

```
#ifdef HAVE_HELLO_LIB
#include "hello.h"
#endif
```

4.3.3 Commandline options

Commandline options to cmake:

```
cmake -DSOME_FLAG=somevalue
```

(with an optional space after the `-D`) can be queried by the cmake script by the `option` command:

```
option( SOME_FLAG "A flag that has some function" defaultvalue )
```

4.4 CMake scripting

Commands learned in this section	
`option`	query a commandline option
`message`	trace message during cmake-ing
`set`	set the value of a variable
CMAKE_SYSTEM_NAME	variable containing the operating system name

The *CMakeLists.txt* file is a script, though it doesn't much look like it.

- Instructions consist of a command, followed by a parenthesized list of arguments.
- (All arguments are strings: there are no numbers.)
- Each command needs to start on a new line, but otherwise whitespace and line breaks are ignored.

Comments start with a hash character.

4.4.1 System dependencies

```
if (CMAKE_SYSTEM_NAME STREQUALS "Window")
  target_compile_options( myapp PRIVATE /W4 )
elseif (CMAKE_SYSTEM_NAME STREQUALS -Wall -Wextra -Wpedantic)
  target_compile_options( myapp PRIVATE /W4 )
endif()
```

4.4.2 Messages, errors, and tracing

The `message` command can be used to write output to the console. This command has two arguments:

```
message( STATUS "We are rolling!")
```

Instead of `STATUS` you can specify other logging levels (this parameter is actually called 'mode' in the documentation); running for instance

```
cmake --log-level=NOTICE
```

will display only messages of 'notice' status or higher.

The possibilities here are: `FATAL_ERROR`, `SEND_ERROR`, `WARNING`, `AUTHOR_WARNING`, `DEPRECATION`, `NOTICE`, `STATUS`, `VERBOSE`, `DEBUG`, `TRACE`.

The `NOTICE`, `VERBOSE`, `DEBUG`, `TRACE` options were added in CMake-3.15.

For a complete trace of everything Cmake does, use the commandline option `--trace`.

You can get a verbose make file by using the option

```
-D CMAKE_VERBOSE_MAKEFILE=ON
```

on the cmake invocation. You still need `make V=1`.

4.4.3 Variables

Variables are set with `set`, or can be given on the commandline:

```
cmake -D MYVAR=myvalue
```

Using the variable by itself gives the value, except in strings, where a shell-like notation is needed:

```
set(SOME_ERROR "An error has occurred")
message(STATUS "${SOME_ERROR}")
set(MY_VARIABLE "This is a variable")
message(STATUS "Variable MY_VARIABLE has value ${MY_VARIABLE}")
```

Some variables are set by other commands. For instance the `project` command sets *PROJECT_NAME* and *PROJECT_VERSION*.

Environment variables can be queried with the *ENV* command:

```
set( MYDIR $ENV{MYDIR} )
```

4.4.3.1 *Numerical variables*

```
math( EXPR lhs_var "math expr" )
```

4.4.4 Control structures

4.4.4.1 *Conditionals*

```
if ( MYVAR MATCHES "value$" )
    message( NOTICE "Variable ended in 'value'" )
elseif( stuff )
    message( stuff )
else()
    message( NOTICE "Variable was otherwise" )
endif()
```

4.4.4.2 *Looping*

```
while( myvalue LESS 50 )
    message( stuff )
endwhile()

foreach ( var IN ITEMS item1 item2 item3 )
  ## something wityh ${var}
endforeach()
foreach ( var IN LISTS list1 list2 list3 )
  ## something wityh ${var}
endforeach()
```

Integer range, with inclusive bounds, upper bound zero by default:

```
foreach ( idx RANGE 10 )
foreach ( idx RANGE 5 10 )
foreach ( idx RANGE 5 10 2 )
endforeach()
```

4.4.4.3 *All the reasons I do not like the Professional Cmake book*

1. The `make install` idiom which I encounter in 100 percent of packages is not discussed.
2. The use of `option` with commandline option is not discussed, in fact the term 'commandline' does not even appear in the index.
3. Does not mention `cmake --build`.

Chapter 5

Source code control through Git

In this tutorial you will learn *git*, the currently most popular *version control* (also *source code control* or *revision control*) systems. Other similar systems are *Mercurial* and *Microsoft Sharepoint*. Earlier systems were *SCCS*, *CVS*, *Subversion*, *Bitkeeper*.

Version control is a system that tracks the history of a software project, by recording the successive versions of the files of the project. These versions are recorded in a *repository*, either on the machine you are working on, or remotely.

This has many practical advantages:

- It becomes possible to undo changes;
- Sharing a repository with another developer makes collaboration possible, including multiple edits on the same file.
- A repository records the history of the project.
- You can have multiple versions of the project, for instance for exploring new features, or for customization for certain users.

The use of a version control system is industry standard practice, and git is by far the most popular system these days.

5.1 Concepts and overview

Older systems were based on having one *central repository*, that all developers coordinated with. These days a setup is popular where each developer (or a small group) has a *local repository*, which gets synchronized to a *remote*repository. In so-called *distributed version control* systems there can even be multiple remote repositories to synchronize with.

It is possible to track the changes of a single file, but often it makes sense to bundle a group of changes together in a *commit*. That also makes it easy to roll back such a group of changes.

If a project is in a state that constitutes some sort of a milestone, it is possible to attach a *tag*, or mark the state as a *release*.

Modern version control systems allow you to have multiple *branch*es, even in the same local repository. This way you can have a main branch for the release version of the project, and one or more development branches for exploring new features.

5.2 Git

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.

Best practices for distributed version control: https://homes.cs.washington.edu/~mernst/advice/version-control.html

5.3 Create and populate a repository

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

You can create a repository two different ways:

1. Create the remote repository and do a clone.
2. Create the repository locally, and then connect it to a remote; this is a lot more work.

5.3.1 Create a repository by cloning

If you want to work on a repository that you have found (or created!) on some site such as *github.com*, you can use

```
git clone URL [ localname ]
```

This gives you a directory with the contents of the repository. If you leave out the local name, the directory will have the name of the repository.

```
Cmd >> git clone https://github.com/TACC/empty.git
    ↪empty
Out >>
Cloning into 'empty'...
warning: You appear to have cloned an empty repository.
Cmd >> cd empty
Cmd >> ls -a
Out >>
.
..
.git
Cmd >> git status
Out >>
On branch main
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

Clone an empty repository and check that it is indeed empty

5.3.2 Create a repository locally

You can also create a directory for your repository, and connect it to a remote site later. For this you do `git init` in the directory that is either empty, or already contains the material that you want to add later. Here, we start with an empty directory.

```
Cmd >> mkdir newrepo
Cmd >> cd newrepo
Cmd >> git init
Out >>
Initialized empty Git repository in
    ↪/users/demo/git/newrepo/.git/
Cmd >> ls -a
Out >>
.
..
.git
Cmd >> git status
Out >>
On branch master
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

Create a directory, and nake it into a repository

5.3.3 Main vs master

It used to be that the default branch (yes, I know, we haven't discussed branches yet) was called 'master'. In a shift of terminology, the preferred name is now 'main'. Sites such as *github* and *gitlab* may already create this name by default; the git software does not do this, as of this writing in 2022.

Renaming a branch is possible. Use `git status` to see what branch you are on,

```
==== Branch renaming
%%
%% See what the main branch is
%%
Cmd >> git status
Out >>
On branch master
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
%%
%% Rename this branch
%%
Cmd >> git branch -m main
Cmd >> git status
Out >>
On branch main
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

See what the main branch is

Move the current branch with

```
git branch -m newbranchname
```

and for good measure check with `git status`.

```
Cmd >> git branch -m main
Cmd >> git status
Out >>
On branch main
No commits yet
nothing to commit (create/copy files and use "git add"
    ↪to track)
```

Rename this branch.

5.4 Adding and changing files

5.4.1 Creating a new file

If you create a file it does not automatically become part of your repository. This takes a sequence of steps. If you create a new file, and you run `git status`, you'll see that the file is listed as 'untracked'.

```
Cmd >> echo foo > firstfile
Cmd >> git status
Out >>
On branch main
No commits yet
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)
```

Create a file; it will initually be untracked

You need to `git add` on your file to tell git that the file belongs to the repository. (You can add a single file, or use a wildcard to add multiple.) However, this does not actually add the file: it moves it to the *staging area*. The status now says that it is a change to be committed.

```
Cmd >> git add firstfile
Cmd >> git status
Out >>
On branch main
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file:   firstfile
```

Add the file to the local repository

Use `git commit` to add these changes to the repository.

```
Cmd >> git commit -m "adding first file"
Out >>
[main (root-commit) f968ac6] adding first file
1 file changed, 1 insertion(+)
create mode 100644 firstfile
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

Commit these changes

5.4.2 Changes to a file in the repository

The `git add` and `git commit` commands need to be repeated if you make any changes to a file in the repository:

When you make changes to a file that has previous been added and committed, the `git status` command will list it as 'modified'.

```
Cmd >> echo bar >> firstfile
Cmd >> cat firstfile
Out >>
foo
bar
Cmd >> git status
Out >>
On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be
    ↪committed)
(use "git restore <file>..." to discard changes in
    ↪working directory)
modified:   firstfile
no changes added to commit (use "git add" and/or "git
    ↪commit -a")
```

Make changes to a file that is tracked.

If you need to check what changes you have made, `git diff` on that file will tell you the differences the between the edited, but not yet added or committed, file and the previous commit version.

```
Cmd >> git diff firstfile
Out >>
diff --git a/firstfile b/firstfile
index 257cc56..3bd1f0e 100644
--- a/firstfile
+++ b/firstfile
@@ -1 +1,2 @@
foo
+bar
```

See what the changes were wrt the previously commit version.

You now need to repeat `git add` and `git commit` on that file.

```
Cmd >> git add firstfile
Cmd >> git commit -m "changes to first file"
Out >>
[main b1edf77] changes to first file
1 file changed, 1 insertion(+)
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

Commit the changes to the local repo.

Doing `git log` will give you the history of the repository, listing the commit numbers, and the messages that you entered on those commits.

```
Cmd >> git log
Out >>
commit b1edf778c17b7c7e6cb1a8ac73fa9b61464eba14
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:40 2022 -0600
changes to first file
commit f968ac6c05dd877db84705a4dcdadbc0bed2c535
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:39 2022 -0600
adding first file
```

Get the log of all commits so far.

5.5 Undoing changes

There are various levels of undo, depending on whether you have added or committed those chaqnges. Or worse; see below.

If you have made a change, and you did `git add` on the file, and you've done `git diff` to make sure;

```
Cmd >> echo bar >> firstfile
Cmd >> cat firstfile
Out >>
foo
bar
bar
Cmd >> git status
Out >>
On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be
    ↪committed)
(use "git restore <file>..." to discard changes in
    ↪working directory)
modified:   firstfile
no changes added to commit (use "git add" and/or "git
    ↪commit -a")
Cmd >> git diff firstfile
Out >>
diff --git a/firstfile b/firstfile
index 3bd1f0e..58ba28e 100644
--- a/firstfile
+++ b/firstfile
@@ -1,2 +1,3 @@
foo
bar
+bar
```

Make regrettable changes.

Do `git checkout` on that file. This gets the last committed version and puts it back in your working directory.

```
Cmd >> git checkout firstfile
Out >>
Updated 1 path from the index
Cmd >> cat firstfile
Out >>
foo
bar
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

Restore previously committed version.

A more complicated scenario is where you have committed the change. Then you need to find the commit id.

You can use `git log` to get the ids of all commits. This is useful if you want to roll back to pretty far in the past. However, if you only want to roll back the last commit, use `git show HEAD` to get a description of just that last commit.

```
Cmd >> git log
Out >>
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
commit 63d6ad16beb4e2d12574fb238c29e8ba11fc6732
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:41 2022 -0600
adding first file
Cmd >> git show HEAD
Out >>
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
diff --git a/firstfile b/firstfile
index 257cc56..3bd1f0e 100644
--- a/firstfile
+++ b/firstfile
@@ -1 +1,2 @@
foo
+bar
```

Find the commit id that you want to roll back.

Then do `git revert sdlksdfkl2343` (with the right id). This will normally open an editor for you to leave comments; you can prevent this with the `--no-edit` option.

```
Cmd >> git revert $commit --no-edit
Out >>
[main 3dca724] Revert "changes to first file"
Date: Sat Jan 29 14:14:42 2022 -0600
1 file changed, 1 deletion(-)
```

Use 'git revert' to roll back.

This will restore the file to its state before the last add and commit, and it will in generally leave the repository back in the state it was before that commit.

```
Cmd >> cat firstfile
Out >>
foo
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

See that we have indeed undone the commit.

However, the log will show that you have reverted a certain commit.

```
Cmd >> git log
Out >>
commit 3dca724a1902e8a5e3dba007c325542c6753a424
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
Revert "changes to first file"

This reverts commit
    ↪e411fad261fd82eb93c328c44978699e946abc0d.
commit e411fad261fd82eb93c328c44978699e946abc0d
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:42 2022 -0600
changes to first file
commit 63d6ad16beb4e2d12574fb238c29e8ba11fc6732
Author: Victor Eijkhout <eijkhout@tacc.utexas.edu>
Date:   Sat Jan 29 14:14:41 2022 -0600
adding first file
```

But there will be an entry in the log.

The `git reset` command can also be used for various types of undo.

5.6 Remote repositories and collaboration

The repository where you have been adding files and changes with `git commit` is the *local repository*. This is great for tracking changes, and reverting them when needed, but a major reason for using source code control is collaboration with others. For this you need a *remote repository*. This involves a set of commands

```
git remote [ other keywords ] [ arguments ]
```

We have some changes, added to the local repository with `git add` and `git commit`

```
Cmd >> git add newfile && git commit -m "adding first
    ↪file"
Out >>
[main 8ce1de4] adding first file
1 file changed, 1 insertion(+)
create mode 100644 newfile
```

Committed changes.

We connect to some remote repository with

```
git remote add servername url
```

If you want to see what your remote is, do

```
git remote -v
```

```
Cmd >> git remote add mainserver
    ↪git@github.com:TACC/tinker.git
Cmd >> git remote -v
Out >>
mainserver      git@github.com:TACC/tinker.git (fetch)
mainserver      git@github.com:TACC/tinker.git (push)
```

Connect local repo to a remote one.

Finally, you can `git push` committed changes to this remote. Git doesn't just push everything here: since you can have multiple branches locally, and multiple *upstream*s remotely, you intially specify both:

```
git push -u servername branchname
```

```
Cmd >> git push -u mainserver main
Out >>
To github.com:TACC/tinker.git
8333bc1..8ce1de4  main -> main
Branch 'main' set up to track remote branch 'main'
    ↪from 'mainserver'.
```

Push changes.

5.6.1 Multiple local repositories

Let's see how changes from one local repository propagate to another. This can be because more than one person is working on a project, or because one person is working from more than one machine.

We make one local repository in directory `person1`.

```
Cmd >> git clone git@github.com:TACC/tinker.git person1
Out >>
Cloning into 'person1'...
```

Person 1 makes a clone.

Create another clone in `person2`. Normally the cloned repositories would be two user accounts, or the accounts of one user on two machines.

```
Cmd >> git clone git@github.com:TACC/tinker.git person2
Out >>
Cloning into 'person2'...
```

Person 2 makes a clone.

Now the first user creates a file, adds, commits, and pushes it. (This of course requires an upstream to be set, but since we did a `git clone`, this is automatically done.)

```
Cmd >> ( cd person1 && echo 123 >> p1 && git add p1 &&
    ↪git commit -m "add p1" && git push )
Out >>
[main 6f6b126] add p1
1 file changed, 1 insertion(+)
create mode 100644 p1
To github.com:TACC/tinker.git
8863559..6f6b126  main -> main
```

Person 1 adds a file and pushes it.

The second user now does

```
git pull
```

to get these changes. Again, because we create the local repository by `git clone` it is clear where the pull is coming from. The pull message will tell us what new files are created, or how many other files were changes.

```
Cmd >> ( cd person2 && git pull )
Out >>
From github.com:TACC/tinker
8863559..6f6b126  main       -> origin/main
Updating 8863559..6f6b126
Fast-forward
p1 | 1 +
1 file changed, 1 insertion(+)
create mode 100644 p1
```

Person 2 pulls, getting the new file.

5.6.2 Merging changes

If you work with someone else, or even if you work solo on a project, but from more than one machine, it may happen that there will be multiple changes on a single file. That is, two local repositories have changes committed, and are now pushing to the same remote.

In the following script we start with the same situation of the previous example, where we have two local repositories, as a stand-in for two users, or two different machines.

We have a file of four lines.

```
Cmd >> cat person1/fourlines
Out >>
1
2
3
4
```

We have a four line file.

The first user makes an edit on the first line; we confirm the state of the file;

```
Cmd >> ( cd person1 && sed -i -e '1s/1/one/' fourlines
    ↪&& cat fourlines )
Out >>
one
2
3
4
```

Person 1 makes a change.

This user pushes the change.

```
Cmd >> ( cd person1 && git add fourlines && git commit
    ↪-m "edit line one" && git push )
Out >>
[main 6767e3f] edit line one
1 file changed, 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
fdd70b7..6767e3f  main -> main
```

Person 1 pushes the change.

The other user also makes a change, but on line 4, so that there is no conflict;

```
Cmd >> ( cd person2 && sed -i -e '4s/4/four/'
    ↪fourlines && cat fourlines )
Out >>
1
2
3
four
```

Person 2 makes a different change to the same file.

This change is added with `git add` and `git commit`, but we proceed more cautiously in pushing: first we pull any changes made by others with

```
git pull --no-edit
git push
```

```
Cmd >> ( cd person2 && git add fourlines && git commit
    ↪-m "edit line four" && git pull --no-edit && git
    ↪push )
Out >>
[main 27fb2b2] edit line four
1 file changed, 1 insertion(+), 1 deletion(-)
From github.com:TACC/tinker
fdd70b7..6767e3f  main       -> origin/main
Auto-merging fourlines
Merge made by the 'recursive' strategy.
fourlines | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
6767e3f..62bd424  main -> main
```

This change does not conflict, we can pull/push.

Now if the first user does a pull, they see all the merged changes.

```
Cmd >> ( cd person1 && git pull && cat fourlines )
Out >>
From github.com:TACC/tinker
6767e3f..62bd424  main       -> origin/main
Updating 6767e3f..62bd424
Fast-forward
fourlines | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
one
2
3
four
```

Person 1 pulls to get all the changes.

5.6.3 Conflicting changes

There can be various reasons for git to report a conflict.

- You are trying to pull changes from another developer, but you have changes yourself that you haven't yet committed. If you don't want to commit your changes (maybe you're still busy editing) you can use `git stash` to set your edits aside. You can later retrieve them with `git stash pop`, or decide to forget all about them with `git stash drop`.

- There is a conflict between your changes, and those you are trying to pull from another developer, or from yourself on another machine. This is the case we will look at here.
- Very similar, there can also be conflicts between two branches you try to merge. We will look at that in section 5.7.1.

As before, we have directories `person1` and `person2` containing independent clones of a repository. We have a file of four lines long, but contrary to above, we now make edits that are too close together for git's auto-merge to deal with them.

```
Cmd >> cat person1/fourlines
Out >>
1
2
3
4
```

The original file.

Now developer 1 makes a change on line 1.

```
Cmd >> ( cd person1 && sed -i -e '1s/1/one/' fourlines
    ↪&& git add fourlines && git commit -m "edit line
    ↪one" && git push )
Out >>
[main a10216d] edit line one
1 file changed, 1 insertion(+), 1 deletion(-)
To github.com:TACC/tinker.git
4955e50..a10216d  main -> main
```

With apologies for some scripting trickery, we use an edit by *sed*, changing 1 on line 1 to `one`. We add, commit, and push this change.

In the meantime, developer 2 makes another change, to the original file. This change can be added and committed to the local repository without any problem.

```
Cmd >> ( cd person2 && sed -i -e '2s/2/two/' fourlines
    ↪&& cat fourlines  && git add fourlines && git
    ↪commit -m "edit line two" )
Out >>
1
two
3
4
[main c9b6ded] edit line two
1 file changed, 1 insertion(+), 1 deletion(-)
```

Change the `2` on line two to `two`. We add and commit this to the local repository.

However, if we try to `git push` this change to the remote repository, we get an error that the remote is ahead of the local repository. So we first pull the state of the remote repository. In the previous section this led to an automatic merge; not so here.

```
Cmd >> ( cd person2 && git pull --no-edit || echo )
Out >>
From github.com:TACC/tinker
4955e50..a10216d  main       -> origin/main
Auto-merging fourlines
CONFLICT (content): Merge conflict in fourlines
Automatic merge failed; fix conflicts and then commit
    ↪the result.
```

The `git pull` call results in a message that the automatic mere failed, indicating what file was the problem.

You can now edit the file by hand, or using some merge tool.

```
Cmd >> ( cd person2 && cat fourlines )
Out >>
<<<<<<< HEAD
1
two
=======
one
2
>>>>>>> a10216da358649df80aaaeb94f1ceef909c2ed83
3
4
```

In between the chevron'ed lines you first get the `HEAD`, that is the local state, followed by the pulled remote state. Edit this file, commit the merge and push.

5.7 Branching

With a *branch* you can keep a completely separate version of all the files in your project.

Initially we have a file on the `main` branch.

```
Cmd >> cat firstfile
Out >>
foo
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

We have a file, committed and all.

We create a new branch, named `dev` and check it out

```
git branch dev
git checkout dev
```

This initially has the same content.

```
Cmd >> git branch dev && git checkout dev
Out >>
Switched to branch 'dev'
Cmd >> cat firstfile
Out >>
foo
```

Make a development branch.

We make changes, and commit them to the current branch.

```
Cmd >> echo bar > firstfile && cat firstfile
Out >>
bar
Cmd >> git status
Out >>
On branch dev
Changes not staged for commit:
(use "git add <file>..." to update what will be
    ↪committed)
(use "git restore <file>..." to discard changes in
    ↪working directory)
modified:   firstfile
no changes added to commit (use "git add" and/or "git
    ↪commit -a")
Cmd >> git add firstfile && git commit -m "dev changes"
Out >>
[dev 933b924] dev changes
1 file changed, 1 insertion(+), 1 deletion(-)
```

Make changes and commit them to the dev branch.

If we switch back to the `main` branch, everything is as before when we made the `dev` branch.

```
Cmd >> git checkout main && cat firstfile && git status
Out >>
Switched to branch 'main'
foo
On branch main
nothing to commit, working tree clean
```

The other branch is still unchanged.

We can inspect differences between branches with

```
git diff branch1 branch2
```

```
Cmd >> git diff main dev
Out >>
diff --git a/firstfile b/firstfile
index 257cc56..5716ca5 100644
--- a/firstfile
+++ b/firstfile
@@ -1 +1 @@
-foo
+bar
```

We can check differences between branches.

5.7.1 Branch merging

One of the points of having branches is to merge them after you have done some development in one branch.

We start with the four line file from before.

```
Cmd >> cat fourlines
Out >>
1
2
3
4
Cmd >> git status
Out >>
On branch main
nothing to commit, working tree clean
```

The main branch is up to date.

Also as before, we have a `dev` branch that contains these contents.

We switch back to the `main` branch and make a change. This will not be visible in the `dev` branch.

```
Cmd >> git checkout main
Out >>
Switched to branch 'main'
Cmd >> sed -i -e '1s/1/one/' fourlines && cat fourlines
Out >>
one
2
3
4
Cmd >> git add fourlines && git commit -m "edit line 1"
Out >>
[main c51d4ff] edit line 1
1 file changed, 1 insertion(+), 1 deletion(-)
```

On line 1, change `1` to `one`.

We switch to the `dev` branch and make another file. The change in the main branch is indeed not here.

```
Cmd >> git checkout dev
Out >>
Switched to branch 'dev'
Cmd >> sed -i -e '4s/4/four/' fourlines && cat
    ↪fourlines
Out >>
1
2
3
four
Cmd >> git add fourlines && git commit -m "edit line 4"
Out >>
[dev dbb0c03] edit line 4
1 file changed, 1 insertion(+), 1 deletion(-)
```

On line 4, change `4` to `four`. This change is far enough away from the other change, that there should be no conflict.

Switching back to the `main` branch, we use

```
git merge dev
```

to merge the `dev` changes into `main`.

```
Cmd >> git checkout main
Out >>
Switched to branch 'main'
Cmd >> git merge dev
Out >>
Auto-merging fourlines
Merge made by the 'recursive' strategy.
fourlines | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
Cmd >> cat fourlines
Out >>
one
2
3
four
```

Merge the dev branch into the main one with `git merge`. Note the 'auto-merge' message, and confirm that both changes to the file are there.

If two developers make changes on the same line, or on adjacent lines, git will not be able to merge and you have to edit the file as in section 5.6.3.

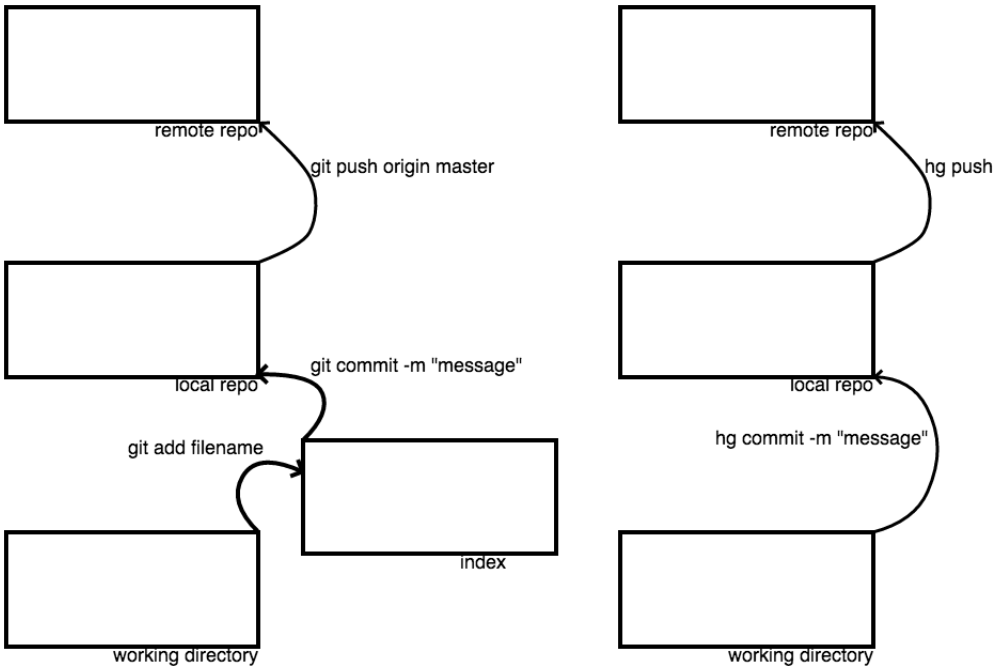


Figure 5.1: Add local changes to the remote repository.

5.8 Conflicts

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

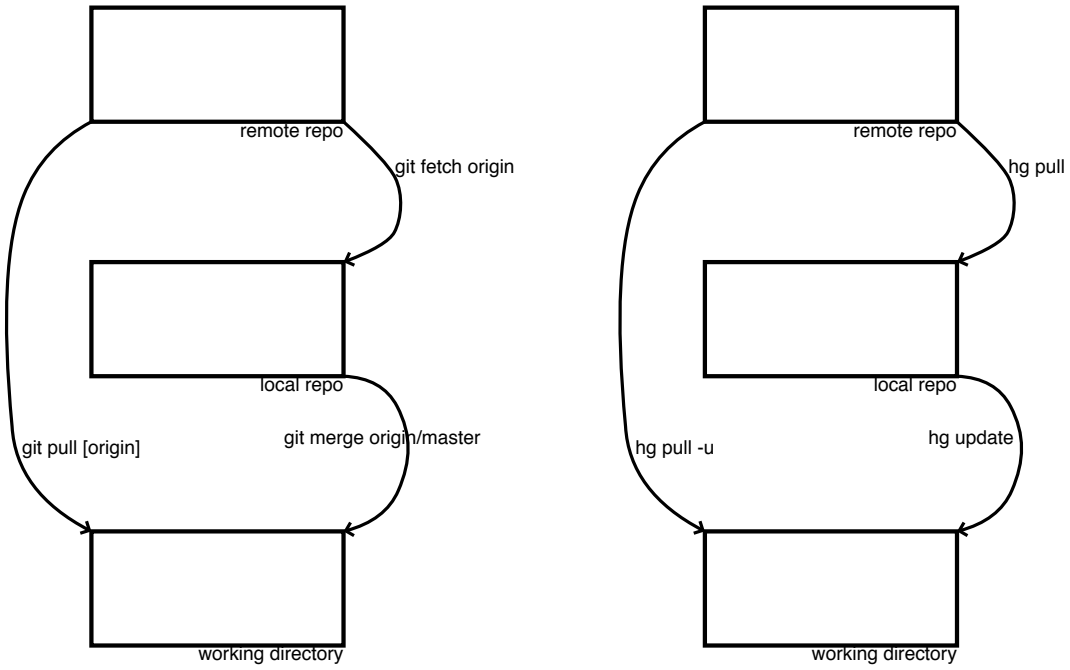


Figure 5.2: Get changes that were made to the remote repository.

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 **??**.

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, version control 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
```

For git:

```
CONFLICT (content): Merge conflict in <name of some file>
Automatic merge failed; fix conflicts and then commit the result.
[solutions-mpi-c:955] emacs <name of some file>
[solutions-mpi-c:956] git add !$ && git commit -m "fix conflict" && git pull && git pus
```

There are now the following options:

1. There is usually a way to indicate whether to use the local or the remote version.
2. There are graphical programs to resolve conflicts. They will typically show you 3 colums, for the two versions, and for your resolution. You can then indicate 'take this from the local version, and this from the remote'.

3. You can also edit the file to resolve the conflicts yourself. We will discuss that shortly.

Both 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
```

or

```
git add <name of that file>
git commit -m "fixed conflict in <name of that file>"
```

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

For conflicts in binary files, git has a flag where you can indicate which version to use:

```
git checkout --ours path/to/file
git checkout --theirs path/to/file
```

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.

5.9 Inspecting the history

Purpose. In this section, you will learn how to view and compare files in 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.

mercurial	git
`hg diff <file>`	`git diff HEAD <file>`
`hg diff -r A -r B <file>`	`git diff A..B <file>`

For example:

```
[src:1151] git pull
[...]
From github.com:Username/Reponame
   c5aaa43..2f5ce0b  main       -> origin/main
Updating c5aaa43..2f5ce0b
Fast-forward
 src/net.cpp | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
[src:1156] git diff c5aaa43..2f5ce0b net.cpp
```

- The `pull` command tells you what the previous and current commit identifiers are; and
- With `diff`, specifying these identifiers, you get the changes in *`diff`* format.

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.

5.10 Branching

Create branch:

`git checkout -b newbranch`

List available:

`git branch`

After that switch:

`git checkout branchname`

`git push --set-upstream origin newbranch`

See what branches there are: `hg brances`

See what branch you are working on: `hg branch`

Switch to a branch (this undoes local changes): `hg update -C branchname`

5.11 Pull requests and forks

Suppose you want to contribute changes to a repository, but you don't have write permissions on a repository. (Of course, you need to have read permissions to the repository in order to make the changes.) Lack of write permission means that you can not create a branch, and push and merge it. A *pull request* is then a way to communicate your changes so that they can be pulled, rather than you pushing them.

1. In github, creating a fork is made with a press of the 'Fork' button.
2. Clone your fork.
   ```
   git clone <original-url> repo-fork
   cd repo-fork
   ```
3. Make your edits in the cloned fork.
4. You need to keep your fork up to date with the original repo. First you connect the original repo as 'upstream' to your fork:
   ```
   git remote add upstream <original-url>
   git remote -v # check on the state of things
   ```
   (the name 'upstream' is the conventional name; it is not a keyword.)

5. Optionally, check out a branch. If you `git status`, you should see
   ```
   On branch YourBranch
   Your branch is up to date with 'origin/YourBranch.
   ```

6. Keep your fork updated:
   ```
   git fetch upstream
   git merge upstream/master master # or specific branch
   ```

To make pull request:

1. Create a new branch
   ```
   git checkout -b YourChangesBranch
   ```

2. Make your changes and commit them:
   ```
   git add file1 file2
   git commit -m "changes summary"
   ```

3. Push your changes to the origin, that is, your repo on github:
   ```
   git push --set-upstream origin YourChangesBranch
   ```

   (remember that you didn't have write permission to the original repo!)
4. You probably now get a message with URL to visit for creating a pull request.

5.12 Other issues

5.12.1 Transport

Mercurial and git can use either `ssh` or `http` as *transport*. With Git you may need to redefine the transport for a `push`:

```
git remote rm origin
git remote add origin git@github.com:TACC/pylauncher.git
```

You can change the transport with `git remote set-url`:

```
git remote -v # check current transport
git remote set-url origin git@hostname:USERNAME/REPOSITORY.git
git remote -v # verify changes
```

Chapter 6

Dense linear algebra: BLAS, LAPACK, SCALAPACK

In this section we will discuss libraries for dense linear algebra operations.

Dense linear algebra, that is linear algebra on matrices that are stored as two-dimensional arrays (as opposed to sparse linear algebra; see section HPC book, section-5.4, as well as the tutorial on PETSc *Parallel Programming book, section-III*) has been standardized for a considerable time. The basic operations are defined by the three levels of *Basic Linear Algebra Subprograms (BLAS)*:

- Level 1 defines vector operations that are characterized by a single loop [13].
- Level 2 defines matrix vector operations, both explicit such as the matrix-vector product, and implicit such as the solution of triangular systems [7].
- Level 3 defines matrix-matrix operations, most notably the matrix-matrix product [6].

The name 'BLAS' suggests a certain amount of generality, but the original authors were clear [13] that these subprograms only covered dense linear algebra. Attempts to standardize sparse operations have never met with equal success.

Based on these building blocks libraries have been built that tackle the more sophisticated problems such as solving linear systems, or computing eigenvalues or singular values. *Linpack*[1] and *Eispack* were the first to formalize these operations involved, using Blas Level 1 and Blas Level 2 respectively. A later development, *Lapack* uses the blocked operations of Blas Level 3. As you saw in section HPC book, section-1.6.1, this is needed to get high performance on cache-based CPUs. (Note: the reference implementation of the BLAS [3] will not give good performance with any compiler; most platforms have vendor-optimized implementations, such as the *MKL* library from Intel.)

With the advent of parallel computers, several projects arose that extended the Lapack functionality to distributed computing, most notably *Scalapack* [4, 2], *PLapack* [20, 19], and most recently Elemental [17]. These packages are harder to use than Lapack because of the need for a two-dimensional cyclic distribution; sections HPC book, section-6.2.2 and HPC book, section-6.3.2. We will not go into the details here.

1. The linear system solver from this package later became the *Linpack benchmark*; see section HPC book, section-2.11.4.

6.0.1 Some general remarks

6.0.1.1 The Fortran heritage

The original BLAS routines were written in Fortran, and the reference implementation is still in Fortran. For this reason you will see the routine definitions first in Fortran in this tutorial It is possible to use the Fortran routines from a C/C++ program:

- You typically need to append an underscore to the Fortran name;
- You need to include a prototype file in your source, for instance `mkl.h`;
- Every argument needs to be a 'star'-argument, so you can not pass literal constants: you need to pass the address of a variable.
- You need to create a column-major matrix.

There are also C/C++ interfaces:

- The C routine names are formed by prefixing the original name with `cblas_`; for instance `dasum` becomes `cblas_dasum`.
- Fortran character arguments have been replaced by enumerated constants, for instance `CblasNoTrans` instead of the `'N'` parameter.
- The Cblas interface can accommodate both row-major and column-major storage.
- Array indices are 1-based, rather than 0-based; this mostly becomes apparent in error messages and when specifying pivot indices.

6.0.1.2 Routine naming

Routines conform to a general naming scheme: `XYYZZZ` where

X precision: `S,D,C,Z` stand for single and double, single complex and double complex, respectively.
YY storage scheme: general rectangular, triangular, banded.
ZZZ operation. See the manual for a list.

6.0.1.3 Data formats

Lapack and Blas use a number of data formats, including

GE General matrix: stored two-dimensionally as `A(LDA,*)`
SY/HE Symmetric/Hermitian: general storage; `UPLO` parameter to indicate upper or lower (e.g. `SPOTRF`)
GB/SB/HB General/symmetric/Hermitian band; these formats use column-major storage; in `SGBTRF` overallocation needed because of pivoting
PB Symmetric of Hermitian positive definite band; no overallocation in `SPDTRF`

6.0.1.4 Lapack operations

For Lapack, we can further divide the routines into an organization with three levels:

- Drivers. These are powerful top level routine for problems such as solving linear systems or computing an SVD. There are simple and expert drivers; the expert ones have more numerical sophistication.

- Computational routines. These are the routines that drivers are built up out of[2]. A user may have occasion to call them by themselves.
- Auxiliary routines.

Expert driver names end on 'X'.

- Linear system solving. Simple drivers: `-SV` (e.g., `DGESV`) Solve $AX = B$, overwrite A with LU (with pivoting), overwrite B with X.
  Expert driver: `-SVX` Also transpose solve, condition estimation, refinement, equilibration
- Least squares problems. Drivers:
  `xGELS` using QR or LQ under full-rank assumption
  `xGELSY` "complete orthogonal factorization"
  `xGELSS` using SVD
  `xGELSD` using divide-conquer SVD (faster, but more workspace than `xGELSS`)
  Also: LSE & GLM linear equality constraint & general linear model
- Eigenvalue routines. Symmetric/Hermitian: `xSY` or `xHE` (also `SP`, `SB`, `ST`)
  simple driver `-EV`
  expert driver `-EVX`
  divide and conquer `-EVD`
  relative robust representation `-EVR`
  General (only `xGE`)
  Schur decomposition `-ES` and `-ESX`
  eigenvalues `-EV` and `-EVX`
  SVD (only `xGE`)
  simple driver `-SVD`
  divide and conquer `SDD`
  Generalized symmetric (`SY` and `HE`; `SP`, `SB`)
  simple driver `GV`
  expert `GVX`
  divide-conquer `GVD`
  Nonsymmetric:
  Schur: simple `GGES`, expert `GGESX`
  eigen: simple `GGEV`, expert `GGEVX`
  svd: `GGSVD`

6.0.1.5 BLAS matrix storage

There are a few points to bear in mind about the way matrices are stored in the BLAS and LAPACK[3]:

6.0.1.5.1 **Array indexing** Since these libraries originated in a Fortran environment, they use 1-based indexing. Users of languages such as C/C++ are only affected by this when routines use index arrays, such as the location of pivots in LU factorizations.

2. Ha! Take that, Winston.
3. We are not going into band storage here.

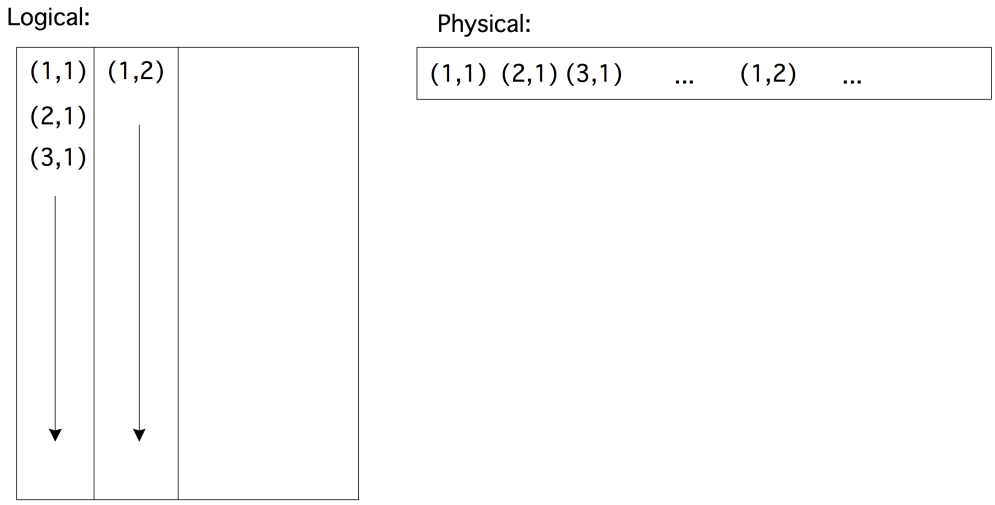


Figure 6.1: Column-major storage of an array in Fortran.

6.0.1.5.2 **Fortran column-major ordering** Since computer memory is one-dimensional, some conversion is needed from two-dimensional matrix coordinates to memory locations. The *Fortran* language uses *column-major* storage, that is, elements in a column are stored consecutively; see figure 6.1. This is also described informally as 'the leftmost index varies quickest'.

Arrays in C, on the other hand, are laid out in *row-major* order. How to create a C array that can be handled by Blas routines is discussed in section 12.4.

6.0.1.6 Submatrices and the `LDA` *parameter*

Using the storage scheme described above, it is clear how to store an $m \times n$ matrix in mn memory locations. However, there are many cases where software needs access to a matrix that is a subblock of another, larger, matrix. As you see in figure 6.2 such a subblock is no longer contiguous in memory. The way to describe this is by introducing a third parameter in addition to `M,N`: we let `LDA` be the 'leading dimension of `A`', that is, the allocated first dimension of the surrounding array. This is illustrated in figure 6.3. To pass the subblock to a routine, you would specify it as

```
call routine( A(3,2), /* M= */ 2, /* N= */ 3, /* LDA= */ Mbig, ... )
```

6.0.2 Performance issues

The collection of BLAS and LAPACK routines are a *de facto* standard: the Application Programmer Interface (API) is fixed, but the implementation is not. You can find reference implementations on the *netlib* website (`netlib.org`), but these will be very low in performance.

On the other hand, many LAPACK routines can be based on the matrix-matrix product (BLAS routine `gemm`), which you saw in section HPC book, section-6.4.1 has the potential for a substantial fraction of peak performance. To achieve this, you should use an optimized version, such as

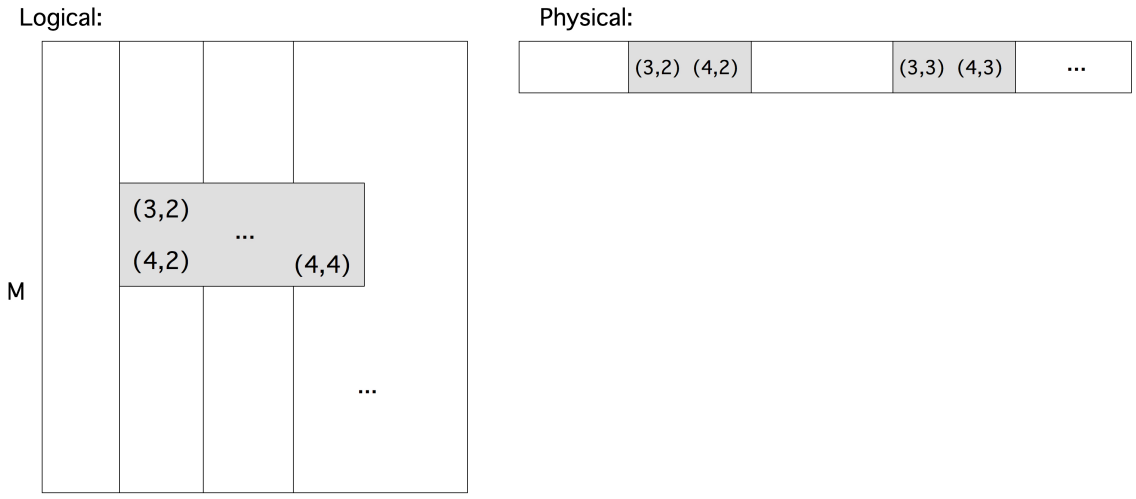


Figure 6.2: A subblock out of a larger matrix.

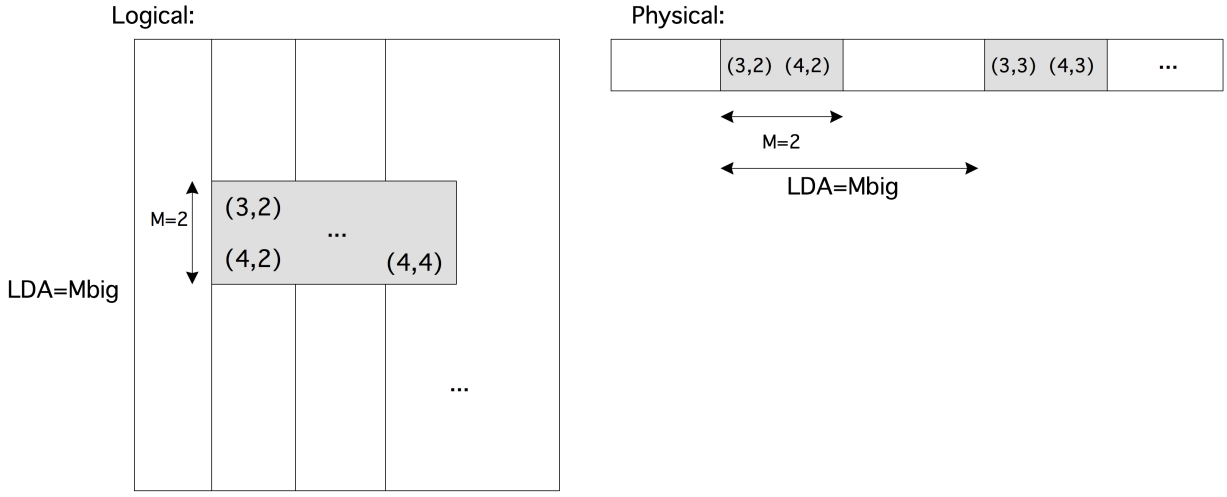


Figure 6.3: A subblock out of a larger matrix, using `LDA`.

- *MKL*, the Intel math-kernel library;
- OpenBlas (`http://www.openblas.net/`), an open source version of the original *Goto BLAS*; or
- *blis* (`https://code.google.com/p/blis/`), a BLAS replacement and extension project.

6.0.3 Some simple examples

Let's look at some simple examples.

The routine `xscal` scales a vector in place.

```
! Fortran
```

```
subroutine dscal(integer N, double precision DA,
    double precision, dimension(*) DX, integer INCX )
// C
void cblas_dscal (const MKL_INT n, const double a,
    double *x, const MKL_INT incx);
```

A simple example:

```
// example1.F90
  do i=1,n
     xarray(i) = 1.d0
  end do
  call dscal(n,scale,xarray,1)
  do i=1,n
     if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
  end do
```

The same in C:

```
// example1c.cxx
  xarray = new double[n]; yarray = new double[n];

  for (int i=0; i<n; i++)
    xarray[i] = 1.;
  cblas_dscal(n,scale,xarray,1);
  for (int i=0; i<n; i++)
    if (!assert_equal( xarray[i],scale ))
      printf("Error in index %d",i);
```

Many routines have an increment parameter. For `xscale` that's the final parameter:

```
// example2.F90
  integer :: inc=2
  call dscal(n/inc,scale,xarray,inc)
  do i=1,n
     if (mod(i,inc)==1) then
        if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
     else
        if (.not.assert_equal( xarray(i),1.d0 )) print *,"Error in index",i
     end if
  end do
```

The matrix-vector product `xgemv` computes $y \leftarrow \alpha Ax + \beta y$, rather than $y \leftarrow Ax$. The specification of the matrix takes the `M,N` size parameters, and a character argument `'N'` to indicate that the matrix is not transposed. Both of the vectors have an increment argument.

```
subroutine dgemv(character TRANS,
    integer M,integer N,
    double precision ALPHA,
    double precision, dimension(lda,*) A,integer LDA,
    double precision, dimension(*) X,integer INCX,
    double precision BETA,double precision, dimension(*) Y,integer INCY
    )
```

An example of the use of this routine:

```
// example3.F90
  do j=1,n
     xarray(j) = 1.d0
     do i=1,m
        matrix(i,j) = 1.d0
     end do
  end do

  alpha = 1.d0; beta = 0.d0
  call dgemv( 'N',M,N, alpha,matrix,M, xarray,1, beta,yarray,1)
  do i=1,m
     if (.not.assert_equal( yarray(i),dble(n) )) &
          print *,"Error in index",i,":",yarray(i)
  end do
```

The same example in C has an extra parameter to indicate whether the matrix is stored in row or column major storage:

```
// example3c.cxx
  for (int j=0; j<n; j++) {
    xarray[j] = 1.;
    for (int i=0; i<m; i++)
      matrix[ i+j*m ] = 1.;
  }

  alpha = 1.; beta = 0.;
  cblas_dgemv(CblasColMajor,
      CblasNoTrans,m,n, alpha,matrix,m, xarray,1, beta,yarray,1);

  for (int i=0; i<m; i++)
    if (!assert_equal( yarray[i],(double)n ))
      printf("Error in index %d",i);
```

Chapter 7

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.

7.1 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);
```

7.2 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.

```
/*
 * 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.

7.3 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
      }
   }
}
}
```

7.4 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.

7.5 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 8

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.

8.1 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
```

8.2 Plotting

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

8.2.1 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`.

8.2.2 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
```

8.2.3 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
```

8.3 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 9

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.

9.1 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 likelihood 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 3.
- Source code management with 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.

9.1.1 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 occurrence 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 stop.

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*'.

9.1.1.1 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 stopped. 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.

9.1.1.2 *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")
```

9.1.2 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.)

9.2 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 exit 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 stop 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 behavior. 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.

9.2.1 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.

9.2.2 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 10.6.2.1.

9.2.3 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 behavior 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.)

9.2.4 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.

9.3 Testing

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

- Correctness proving: the programmer draws up predicates that describe the intended behavior of code fragments and proves by mathematical techniques that these predicates hold [10, 5].
- 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 behavior. 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 HPC book, section-5.5.1), but others have no such built-in error correction (nonstationary methods; section HPC book, section-5.5.8). As a result, the same iterative process can take different numbers of iterations depending on how many processors are used.

9.3.1 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 10

Debugging

> Debugging is like being the detective in a crime movie where you are also the murderer.
> (Filipe Fortes, 2013)

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 processors, 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 behavior of a running program. In this section you will familiarize yourself with *gdb* and *lldb*, the open source debuggers of the *GNU* and *clang* projects respectively. 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 found in the repository in the directory `code/gdb`.

10.1 Invoking the debugger

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.

Table 10.1: List of common gdb / lldb commands.

gdb	lldb
Starting a debugger run	
`$ gdb program` `(gdb) run`	`$ lldb program` `(lldb) run`
Displaying a stack trace	
`(gdb) where`	`(lldb) thread backtrace`
Investigate a specific frame	
`frame 2`	`frame select 2`
Run/step	
`run / step / continue`	thread continue / step-in/over/out
Set a breakpoint at a line	

`break foo.c:12`
`break foo.c:12 if n>0`
`info breakpoints`
	`breakpoint set [ -f foo.c ] -l 12`

Set a breakpoint for exceptions	
`catch throw`	`break set -E C++`

Starting a debugger run	
gdb	lldb
`$ gdb program` `(gdb) run`	`$ lldb program` `(lldb) run`

Here is an example 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
```

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.

```
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
```

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
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.
```

10.2 Finding errors: where, frame, print

Let us now consider some programs with errors.

10.2.1 C programs

The following code has several errors. We will use the debugger to uncover them.

```
// 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 exit. 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:

Displaying a stack trace

gdb	lldb
`(gdb) where`	`(lldb) thread backtrace`

```
(gdb) where
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7
```

We inspect the actual problem:

Investigate a specific frame

gdb	clang
`frame 2`	`frame select 2`

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. The following program fills an array, forward, and reads it out, backward. However, there is an indexing error in the second loop.

```
// 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
```

You see that the index where the debugger finally complains is quite a bit larger than the size of the array.

Exercise 10.1. Can you think of a reason why indexing out of bounds is not immediately fatal? What would determine where it does become a problem? (Hint: how is computer memory structured?)

In section 10.6.2.1 you will see a tool that spots any out-of-bound indexing.

10.2.2 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 end prematurely 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.

10.3 Stepping through a program

Stepping through a program		
gdb	lldb	meaning
`run`		start a run
`cont`		continue from breakpoint
`next`		next statement on same level
`step`		next statement, this level or next

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() {
  feenableexcept(FE_INVALID | FE_OVERFLOW);
  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;
```

Most of the time you will set a breakpoint at a specific line:

Set a breakpoint at a line	
gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`

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 `step`s 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`.

10.4 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?

10.5 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.

Set a breakpoint	
gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`
`break foo.c:12 if n>0`	

Remark 6 *You can break on* `NaN` *with the following trick:*

```
break foo.c:12 if x!=x
```

using the fact that `NaN` *is the only number not equal to itself.*

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`.

Information about breakpoints:

- 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.
- In languages with *exceptions*, such as *C++*, you can set a *catchpoint*:

Set a breakpoint for exceptions	
gdb	clang
`catch throw`	`break set -E C++`

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
```

10.6 Memory debugging

Many problems in programming stem from memory errors. We start with a sort description of the most common types, and then discuss tools that help you detect them.

10.6.1 Type of memory errors

10.6.1.1 Invalid pointers

Dereferencing a pointer that does not point to an allocated object can lead to an error. If your pointer points into valid memory anyway, your computation will continue but with incorrect results.

However, it is more likely that your program will probably exit with a *segmentation violation* or a *bus error*.

10.6.1.2 Out-of-bounds errors

Addressing outside the bounds of an allocated object is less likely to crash your program and more likely to give incorrect results.

Exceeding bounds by a large enough amount will again give a segmentation violation, but going out of bounds by a small amount may read invalid data, or corrupt data of other variables, giving incorrect results that may go undetected for a long time.

10.6.1.3 Memory leaks

We speak of a *memory leak* if allocated memory becomes unreachable. Example:

```
if (something) {
  double *x = malloc(10*sizeofdouble);
  // do something with x
}
```

After the conditional, the allocated memory is not freed, but the pointer that pointed to has gone away.

This last type especially can be hard to find. Memory leaks will only surface in that your program runs out of memory. That in turn is detectable because your allocation will fail. It is a good idea to always check the return result of your `malloc` or `allocate` statement!

10.6.2 Memory tools

10.6.2.1 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 `freed`.

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?

10.6.2.2 Electric fence

The *electric fence* library is one of a number of tools that supplies a new *`malloc`* with debugging support. These are linked instead of the `malloc` of the standard *`libc`*.

```
cc -o program program.c -L/location/of/efence -lefence
```

Suppose your program has an out-of-bounds error. Running with gdb, this error may only become apparent if the bounds are exceeded by a large amount. On the other hand, if the code is linked with *`libefence`*, the debugger will stop at the very first time the bounds are exceeded.

10.7 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 HPC book, section-2.6.3.6.

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 );
```

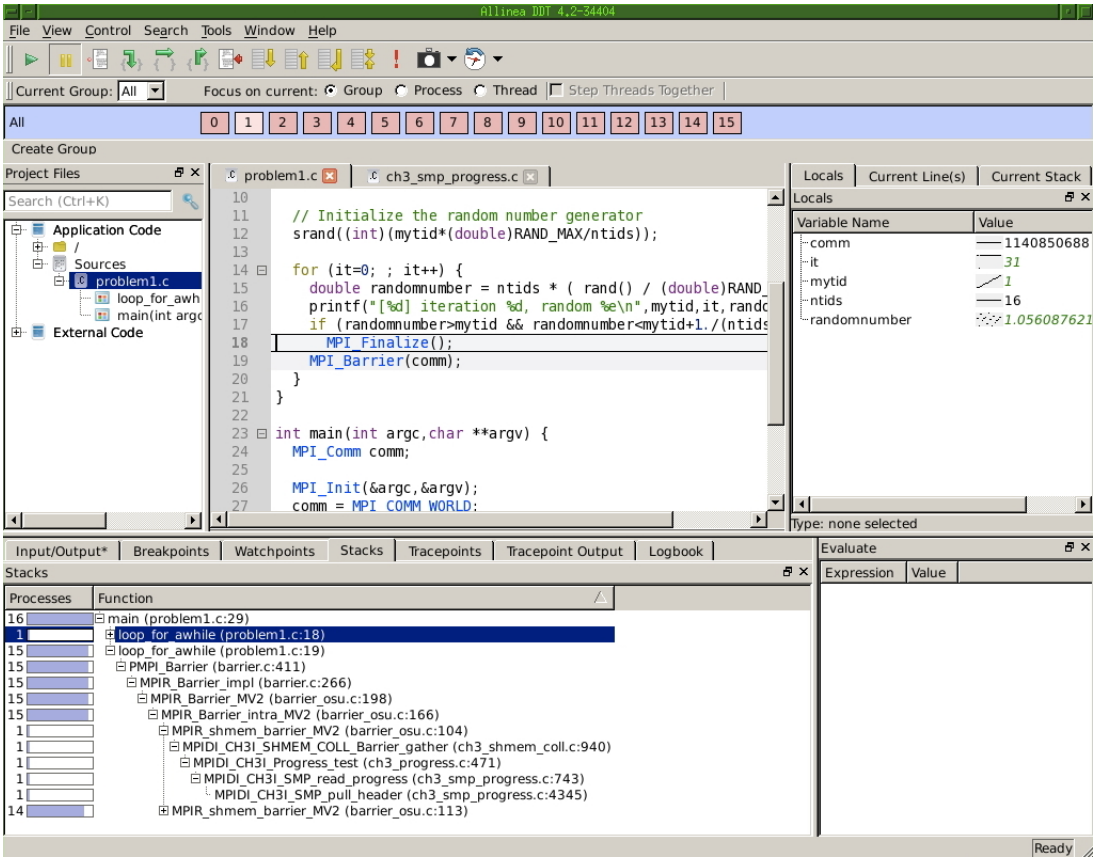


Figure 10.1: Display of 16 processes in the DDT debugger.

```
  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):

```
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 10.1 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 calling `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 10.2. 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 10.3. 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.

10.8 Further reading

A good tutorial: `http://www.dirac.org/linux/gdb/`.

Reference manual: `http://www.ofb.net/gnu/gdb/gdb_toc.html`.

Chapter 11

Language 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 or Python.

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.

For the issues of arrays, see chapter 12.

11.1 C/Fortran interoperability

11.1.1 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:
void foo() {
  return;
}
! Fortran
      Subroutine foo()
      Return
      End Subroutine
```

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)
{
  // this is the actual function
}
int SomeCFunction_(int i,float f)
{
  return SomeCFunction(i,f);
}
```

11.1.2 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);
```

11.1.3 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
```

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.

11.2 C/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.

11.3 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.

11.4 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;
}
```

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.

11.5 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.

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.

11.6 Python calling C code

Because of its efficiency of computing, C is a logical language to use for the lowest layers of a program. On the other hand, because of its expressiveness, Python is a good candidate for the top layers. It is then a logical thought to want to call C routines from a python program. This is possible using the python *ctypes* module.

1. You write your C code, and compile it to a dynamic library as indicated above;
2. The python code loads the library dynamically, for instance for *libc*:
```
path_libc = ctypes.util.find_library("c")
libc = ctypes.CDLL(path_libc)
libc.printf(b"%s\n", b"Using the C printf function from Python ... ")
```
3. You need to declare what the types are of the C routines in python:
```
test_add = mylib.test_add
test_add.argtypes = [ctypes.c_float, ctypes.c_float]
test_add.restype = ctypes.c_float
test_passing_array = mylib.test_passing_array
test_passing_array.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int]
test_passing_array.restype = None
```
4. Scalars can be passed simply; arrays need to be constructed:
```
data = (ctypes.c_int * Nelements)(*[x for x in range(numel)])
```

11.6.1 Boost

Another way to let C and python interact is through the *Boost* library.

Let's start with a C/C++ file that was written for some other purpose, and with no knowledge of Python or interoperability tools:

```
char const* greet()
{
  return "hello, world";
}
```

With it, you should have a `.h` header file with the function signatures.

Next, you write a C++ file that uses the Boost tools:

```
#include <boost/python.hpp>

#include "hello.h"

BOOST_PYTHON_MODULE(hello_ext)
{
  using namespace boost::python;
  def("greet", greet);
}
```

The crucial step is compiling both C/C++ files together into a *dynamic library*:

```
icpc -shared -o hello_ext.so hello_ext.o hello.o \
    -Wl,-rpath,/pythonboost/lib  -L/pythonboost/lib  -lboost_python39 \
    -Wl,-rpath,/python/lib       -L/python/lib       -lpython3
```

You can now import this library in python, giving you access to the C function:

```
import hello_ext
print(hello_ext.greet())
```

Chapter 12

Arrays

12.1 C arrays

12.1.1 Static arrays

The easiest way to create arrays is with the 'square bracket notation':

```
int x[5];
int y[6][7];
```

The same square brackets are used for indexing:

```
for (int row=0; row<nrows; row++)
  for (int col=0; col<ncols; col++)
    moments[row][col] = pow( coefficient[row],(double)col );
```

In these examples we used a constant for the array bound. See section 12.1.3 for using a variable.

12.1.1.1 Allocation

What we are calling 'static' arrays are actually technically called *automatic array*s. Static arrays are arrays with the keyword `static`:

```
static float x[5]; // a truly `static' array
int main() {
  float y[6]; // this one is `automatic'
}
```

Automatic arrays, which we will from now on call 'static', are usually allocated on the *stack*, because they have static scope:

```
// variables `x' and `y' don't exist
if (whatever) {
  int x;
  float y[2];
  // variables `x' and `y' exist for the duration of the conditional
  ....
}
// variables `x' and `y' don't exist anymore
```

Thus, creating too many of these may lead to *stack overflow*. Check out the `limit` command.

12.1.1.2 Passing to functions

You can pass an array to a function, indicating in the function prototype that it is an array. However, the function can not query the array length, so that has to be passed separately if this information is needed.

Code:

```
void set_last( int array[],int len, int value ) {
  array[len-1] = value;
}
  int x[5];
  set_last(x,5,3);
  printf("%d\n",x[4]);
```

Output
[code/array] set1d:

```
make[1]: *** No rule to make
    target `cansi', needed by
     `run_set1d'.  Stop.
```

You can also use the equivalence of arrays and pointers:

Code:

```
// cansi.c
void set_pointer( int *array, int loc, int value ) {
  array[loc] = value;
}
  set_pointer(x,2,1);
  printf("%d\n",x[2]);
```

Output
[code/array] setstar:

```
make[1]: *** No rule to make
    target `cansi', needed by
     `run_setstar'.  Stop.
```

12.1.2 Multi-dimensional arrays

Declaring a multi-dimensional array:

```
int y[6][7];
```

Initialization:

```
int z[2][2] = { {1,2},{3,4} };
```

Such arrays are stored in *row-major* ordering, meaning that rows are contiguous in memory.

Code:

```
  int y[2][2];
  y[0][0] = 1; y[0][1] = 2; y[1][0] = 3; y[1][1] = 4;
  int *yloc = (int*)y;
  for (int i=0; i<4; i++)
    printf("%d ",*(yloc++));
  printf("\n");
```

Output
[code/array] set2d:

```
make[1]: *** No rule to make
    target `cansi', needed by
     `run_set2d'.  Stop.
```

Passing them to functions is a little more tricky: all dimensions except the first have to be pass explicitly.

Code:

```
  int y[2][2];
  y[0][0] = 1; y[0][1] = 2; y[1][0] = 3; y[1][1] = 4;
  int *yloc = (int*)y;
  for (int i=0; i<4; i++)
    printf("%d ",*(yloc++));
  printf("\n");
```

Output
[code/array] pass2d:

```
make[1]: *** No rule to make
    target `cansi', needed by
     `run_pass2d'.  Stop.
```

12.1.3 Variable-length arrays

In *Ansi C*, arrays had to be declared with compile-time bounds, as above. The *C99* standard has added *variable-length array*s:

Code:

```
// c99.c
  int len;
  scanf("%d",&len);
  int x[len];
```

Output
[code/array] set99:

```
make[1]: *** No rule to make
    target `c99', needed by `
    run_set99'.  Stop.
```

- These arrays are still 'automatic' so they only live in the scope where they are defined.
- The term 'variable-length' does not mean that the length is variable; only that it is specified with a variable. Resizable arrays only exist in C++; see section 12.2.2.)
- Such declarations can also be done in multi-D. However, they are pointless for passing arrays to functions.
- Note: this mechanism was already available as an extension on many compilers before the C99 standard.

12.1.4 Dynamically allocated arrays

Before C99's variable-length arrays, the only way to create arrays with a size determined at run-time was through *dynamic allocation*, using the `malloc` keyword. This

1. takes an argument that is a number of bytes, and
2. returns the address of a block of memory of that size.
3. It returns zero if the block could not be created.

```
int arraysize = 500;
float *x;
x = (float*) malloc( arraysize*sizeof(float) );
if (!x) printf("Could not allocate x\n");
```

Indexing, both in the scope where the array is created, and in subprograms it is passed to, is exactly the same as above. For passing such arrays to functions, the type is now `float*`:

Code:

```
// cmalloc.c
void set_last( float *x,int len,float value ) {
  x[len-1] = value;
}
  float *x;
  x = (float*) malloc( len*sizeof(float) );
  set_last(x,len,3.14);
  printf("%e\n",x[len-1]);
```

Output
[code/array] cmalloc:

```
make[1]: *** No rule to make
    target `cmalloc', needed
    by `run_cmalloc'.  Stop.
```

12.1.5 Dynamic arrays, scope, memory leaks

The memory allocated for dynamic arrays is not subject to scope:

```
void create( float **x,int len ) {
  float *x_space = (float*) malloc(len*sizeof(float));
  *x = s_space;
}
int main() {
  float *x;
  create(&x,50);
}
```

Thus, this allocation has to happen on the *heap* This is good for flexible programming, but may lead to *memory leaks*.

12.1.6 Multi-dimensional dynamic arrays

Solution 1:

```
// cmalloc2d.c
  int (*numbers)[cols] = malloc( sizeof(int[rows][cols]) );
```

This allocates a single block, consisting of arrays of length `cols`:

Code:

```
long row2row = (long)(numbers[1])-(long)(numbers[0]);
printf("row length bytes=%ld, ints=%ld\n",row2row,
 row2row/sizeof(int));
```

Output [code/array] rowlength:

```
make[1]: *** No rule to make
    target `cmalloc2d',
    needed by `run_rowlength
    '.  Stop.
```

Solution 2:
create a one-dimensional array, and convert multi-dimensional indices to one-dimensional.

Code:

```
#define INDEX(i,j,m,n) (i)*(n)+(j)
void set_corner( float *array,int m,int n,float v ) {
  array[ INDEX(m-1,n-1,m,n) ] = v;
}
  float *y;
  y = (float*) malloc( len*len*sizeof(float) );
  set_corner( y,len,len,5.12 );
  printf("%e\n",y[ INDEX(len-1,len-1,len,len) ]);
  printf("malloc2d\n");
```

Output [code/array] cmallocpass:

```
make[1]: *** No rule to make
    target `cmalloc', needed
    by `run_cmallocpass'.
    Stop.
```

12.1.7 Type theory of arrays

After declaring

```
int x[5][6];
```

the expression `x[3]` stands for an array of length 6, compatible with being an `int*`.

Does that mean that `x` itself is of type `int**`? No, it is not. Careful parsing of the standard shows that `x` itself is also of type `int*`! Above you already saw that multi-dimensional arrays are contiguous in memory, so you can step through their content with pointer arithmetic.

Let's explore the notion of `int**` and arrays a little more. You can in fact write:

```
int **x;
x = (int**)malloc( nrows*sizeof(int*) );
for (int irow=0; irow<nrows; irow++)
  x[irow] = (int*)malloc( ncolumns*sizeof(int) );
```

and the resulting object can also be indexed with `x[i][j]`. However, this has significant disadvantages:

1. The 'array' is no longer contiguous, so you can not easily step through the contents.
2. Copying its contents is harder than copying contiguous data.
3. The performance of operations may suffer because of the lack of locality and regularity.

12.2 C++ arrays

The C arrays described above are available in C++, with exception of variable-length arrays. However, better mechanisms exist.

12.2.1 Array

The C++ `std::array` is close to the C 'static arrays' (section 12.1.1) in that it requires bounds that are known at compile time.

```
std::array<float,8> eight_floats;
```

Note that the size is a template parameter, not a parameter of the constructor.

While the compile-time bound is a severe restriction on flexibility, this is the most efficient array variant in C++:

- There is no storage overhead in addition to the element storage;
- Nested arrays of this type are form contiguous memory;
- The compiler can optimize these arrays as much as the C-style static arrays.

However, storage for these arrays happens on the heap, rather than on the stack as for C static arrays.

The convenience of having methods such as `size` and bound checking through `at` makes these arrays preferable over the C variant.

12.2.2 Vector

The `std::vector` also creates space on the heap. By contrast with `std::array`

- the size can be specified as a dynamically determined quantity; and
- the size can be changed.

This first difference means that the vector now needs a control block that contains the size and the allocated capacity. The second difference implies that certain operations can be computationally inefficient.

12.3 Fortran

12.3.1 Static

Arrays can be created with compile-time bounds:

```
Integer,dimension(5) :: x
Integer,parameter :: size = 6
Integer,dimension(size,size) :: y
```

These arrays are scoped.

Unlike in C, Fortran arrays can have a specified lower bound:

```
Integer,dimension(-1:7) :: x
```

12.3.2 Dynamic

```
Real*4,dimension(:),allocatable :: x
Integer :: n

Read *,n
Allocate(x(n))
```

Such arrays are also scoped, and can therefore not lead to memory leaks, unlike `malloc`'ed arrays in C.

12.3.3 Memory layout

Fortran arrays are column-major: the columns are stored contiguously. For higher dimensions this is also phrased as 'the first index varies quickest'.

12.4 Layout in memory

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 extension: 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.

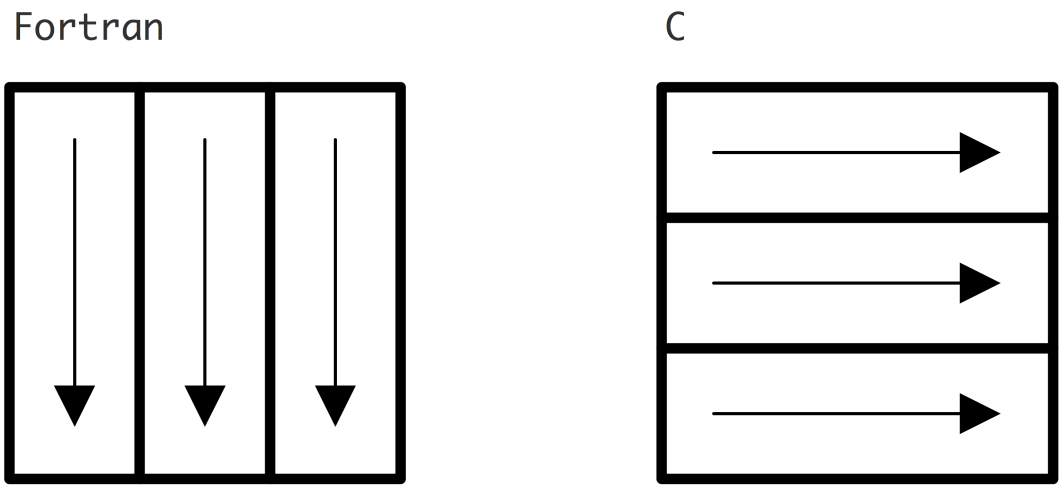


Figure 12.1: Fortran and C array storage by columns and rows respectively.

- 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.

12.4.1 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
```

Chapter 13

Bit operations

In most of this book we consider numbers, such as integer or floating point representations of real numbers, as our lowest building blocks. Sometimes, however, it is necessary to dig deeper and consider the actual representation of such numbers in terms of bits.

Various programming languages have support for bit operations. We will explore the various options.

13.1 Construction and display

13.1.1 C/C++

The built-in possibilities for display are limited:

```
printf("Octal: %o",i);
printf("Hex  : %x",i);
```

gives octal and hexadecimal representation, but there is no *format specifier* for binary. Instead use the following bit of magic:

```
void printBits(size_t const size, void const * const ptr)
{
  unsigned char *b = (unsigned char*) ptr;
  unsigned char byte;
  for (int i=size-1; i>=0; i--) {
      for (int j=7; j>=0; j--) {
          byte = (b[i] >> j) & 1;
          printf("%u", byte);
        }
    }
}
/* ... */
printBits(sizeof(i),&i);
```

13.1.2 Python

- The python `int` function converts a string to int. A second argument can indicate what base the string is to be interpreted in:

```
five     = int('101',2)
maxint32 = int('0xffffffff',16)
```

- Function `bin hex` convert an int to a string in base 2,8,16 respectively.
- Since python integers can be of unlimited length, there is a function to determine the bit length (Python version 3.1): `i.bit_length()`.

13.2 Bit operations

Boolean operations are usually applied to the boolean datatype of the programming language. Some languages allow you to apply them to actual bits.

	boolean	bitwise (C)	bitwise (Py)
and	&&	&	&
or	||	|	|
not	!		~
xor		^	

Additionally, there are operations on the bit string as such:

left shift	<<
right shift	>>

Exercise 13.1. Use bit operations to test whether a number is odd or even.

The shift operations are sometimes used as an efficient shorthand for arithmetic. For instance, a left shift by one position corresponds to a multiplication by two.

Exercise 13.2. Given an integer n, find the largest multiple of 8 that is $\leq n$.

This mechanism is sometimes used to allocate aligned memory. Write a routine

```
aligned_malloc( int Nbytes, int aligned_bits );
```

that allocates `Nbytes` of memory, where the first byte has an address that is a multiple of `aligned_bits`.

Chapter 14

LaTeX for scientific documentation

14.1 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 favorite 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 be generated automatically, for instance from a database. Also, you can edit them with whatever your favorite editor happens to be.

Another point in favor 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 then 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 14.4.

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.

14.1.1 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.

14.1.2 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 14.1: A minimal LaTeX document.

Exercise 14.1. Create a text file `minimal.tex` with the content as in figure 14.1. 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.

Things to watch out for. 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.

14.2 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 [16].

14.2.1 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.

14.2.2 Some simple text

Purpose. In this section you will learn some basics of text formatting.

Exercise 14.2. Create a file `first.tex` with the content of figure 14.1 in it. Type some text in the preamble, that is, before the `\begin{document}` line and run `pdflatex` on your file.

Intended 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 14.3. 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.

Intended 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 14.4. 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.

Intended outcome. TEX interprets one or more blank lines as the separation between paragraphs.

Exercise 14.5. Add `\usepackage{pslatex}` to the preamble and rerun `pdflatex` on your document. What changed in the output?

Intended outcome. This should have the effect of changing the typeface from the default to Times Roman.

Things to watch out for. 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 14.6. Replace `article` by `artikel3` in the documentclass declaration line and reformat your document. What changed?

Intended 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.

Things to watch out for. 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 14.2.9.

14.2.3 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 14.7. Put `$x+y$` somewhere in a paragraph and format your document. Put `\[x+y\]` somewhere in a paragraph and format.

Intended 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 14.8. Take the last example and typeset it as display math. Do you see a difference with inline math?

Intended 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.

14.2.4 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 14.9. 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?

Intended 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.

Things to watch out for. 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 14.10. 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.

Intended 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.

14.2.5 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 14.11. Add some lists to your document, including nested lists. Inspect the output.

Intended outcome. Nested lists will be indented further and the labeling and numbering style changes with the list depth.

Exercise 14.12. Add a label to an item in an `enumerate` list and refer to it.

Intended outcome. Again, the `\label` and `\ref` commands work as before.

14.2.6 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}
```

14.2.7 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.

14.2.8 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.

14.2.9 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 14.2.8).

14.3 A worked out example

The following example `demo.tex` contains many of the elements discussed above.

You also need the file `math.bib`:

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

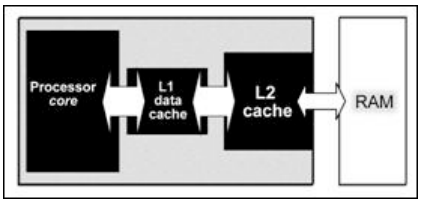


Figure 1: this is the only figure

As I showed in the introductory section 1, in the paper [1], it was shown that equation (1)

- There is an item.

- 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.

14.3.1 Listings

The 'listings' package is makes it possible to have source code included, with coloring and indentation automatically taken care of.

```
\documentclass{article}

\usepackage[pdftex]{hyperref}
\usepackage{pslatex}


%%%%
%%%% Import the listings package
%%%%
\usepackage{listings,xcolor}

%%%%
%%%% Set a basic code style
%%%% (see documentation for more options)
%%%%
\lstdefinestyle{reviewcode}{
  belowcaptionskip=1\baselineskip,
    breaklines=true, frame=L,
  xleftmargin=\parindent, showstringspaces=
    false,
  basicstyle=\footnotesize\ttfamily,
  keywordstyle=\bfseries\color{blue},
  commentstyle=\color{red!60!black},
  identifierstyle=\slshape\color{black},
  stringstyle=\color{green!60!black},
    columns=fullflexible,
  keepspaces=true,tabsize=8,
}
\lstset{style=reviewcode}

\lstset{emph={ %% MPI commands
    MPI_Init,MPI_Initialized,MPI_Finalize,
    MPI_Finalized,MPI_Abort,
    MPI_Comm_size,MPI_Comm_rank,
    MPI_Send,MPI_Isend,MPI_Rsend,MPI_Irsend,
    MPI_Ssend,MPI_Issend,
    MPI_Recv,MPI_Irecv,MPI_Mrecv,
    MPI_Sendrecv,MPI_Sendrecv_replace,
  },emphstyle={\color{red!70!black}\bfseries
    }
}
\lstset{emph={[2] %% constants
    MPI_COMM_WORLD,MPI_STATUS_IGNORE,
    MPI_STATUSES_IGNORE,MPI_STATUS_SIZE,
    MPI_INT,MPI_INTEGER,
  },emphstyle={[2]\color{green!40!black}}
}
\lstset{emph={[3] %% types
    MPI_Aint,MPI_Comm,MPI_Count,MPI_Datatype
    ,MPI_Errhandler,MPI_File,MPI_Group,
  },emphstyle={[3]\color{yellow!30!brown}\
    bfseries},
}

\begin{document}
\title{SSC 335: listings demo}
\author{Victor Eijkhout}
\date{today}
\maketitle

\section{C examples}

\lstset{language=C}
\begin{lstlisting}
int main() {
  MPI_Init();
  MPI_Comm comm = MPI_COMM_WORLD;
  if (x==y)
    MPI_Send( &x,1,MPI_INT,0,0,comm);
  else
    MPI_Recv( &y,1,MPI_INT,1,1,comm,
    MPI_STATUS_IGNORE);
  MPI_Finalize();
}
\end{lstlisting}

\section{Fortran examples}

\lstset{language=Fortran}
\begin{lstlisting}
Program myprogram
  Type(MPI_Comm) :: comm = MPI_COMM_WORLD
  call MPI_Init()
  if (.not. x==y ) then
    call MPI_Send( x,1,MPI_INTEGER,0,0,comm)
    ;
  else
    call MPI_Recv( y,1,MPI_INTEGER,1,1,comm,
    MPI_STATUS_IGNORE)
  end if
  call MPI_Finalize()
End Program myprogram
\end{lstlisting}

\end{document}
```

Output:

SSC 335: listings demo

Victor Eijkhout

today

1 C examples

```c
int main() {
  MPI_Init();
  MPI_Comm comm = MPI_COMM_WORLD;
  if (x==y)
    MPI_Send( &x,1,MPI_INT,0,0,comm);
  else
    MPI_Recv( &y,1,MPI_INT,1,1,comm,MPI_STATUS_IGNORE);
  MPI_Finalize();
}
```

2 Fortran examples

```fortran
Program myprogram
  Type(MPI_Comm) :: comm = MPI_COMM_WORLD
  call MPI_Init()
  if (.not. x==y ) then
    call MPI_Send( x,1,MPI_INTEGER,0,0,comm);
  else
    call MPI_Recv( y,1,MPI_INTEGER,1,1,comm,MPI_STATUS_IGNORE)
  end if
  call MPI_Finalize()
End Program myprogram
```

14.3.2 Native graphing

You have seen how to include graphics files, but it is also possible to let LaTeX do the drawing. For this, there is the *tikz* package. Here we show another package *pgfplots* that uses `tikz` to draw numerical plots.

```
\documentclass{artikel3}

\usepackage[pdftex]{hyperref}
\usepackage{pslatex}

\usepackage{wrapfig}
\usepackage{pgfplots}
\pgfplotsset{width=6.6cm,compat=1.7}

\usepackage{geometry}
\addtolength{\textwidth}{.75in}
\addtolength{\textheight}{.75in}

\begin{document}
\title{SSC 335: barchart demo}
\author{Victor Eijkhout}
\date{today}
\maketitle

\section{Two graphs}

\begin{wrapfigure}{l}{2in}
  \hrule width 3in height 0pt
  \begin{tikzpicture}
    \begin{axis}
      [
        ybar,
        enlargelimits=0.15,
        ylabel={\#Average Marks},
        xlabel={\ Students Name},
        symbolic x coords={Tom, Jack, Hary,
    Liza, Henry},
        xtick=data,
        nodes near coords,
        nodes near coords align={vertical},
      ]
      \addplot coordinates {(Tom,50) (Jack
    ,90) (Hary,70) (Liza,80) (Henry,60) };
    \end{axis}
  \end{tikzpicture}
\end{wrapfigure}
Lorem ipsum dolor sit amet, consectetur
    adipiscing elit, sed do eiusmod tempor
    incididunt ut labore et dolore magna
    aliqua. Pharetra massa massa ultricies
    mi quis hendrerit. Tempor nec feugiat
    nisl pretium fusce id velit ut tortor.
    Eget nulla facilisi etiam dignissim
    diam quis enim. Cursus sit amet dictum
    sit amet justo donec. Tortor consequat
    id porta nibh venenatis cras sed felis
    eget. Senectus et netus et malesuada
    fames ac turpis egestas integer.
    Ultricies mi quis hendrerit dolor magna
     eget est. A iaculis at erat
    pellentesque adipiscing. Sagittis orci
    a scelerisque purus. Quisque non tellus
     orci ac. Nisl nunc mi ipsum faucibus.
    Vivamus at augue eget arcu dictum
    varius duis. Maecenas ultricies mi eget
     mauris pharetra et ultrices neque
    ornare. Pulvinar neque laoreet
    suspendisse interdum consectetur. Nunc
    id cursus metus aliquam eleifend mi.
    Tristique sollicitudin nibh sit amet
    commodo nulla. Massa tincidunt nunc
    pulvinar sapien et ligula ullamcorper
    malesuada. Justo laoreet sit amet
    cursus sit. Laoreet id donec ultrices
    tincidunt arcu non sodales.

\begin{wrapfigure}{r}{2in}
  \hrule width 3in height 0pt % kludge:
    picture is not wide enough
  \begin{tikzpicture}
    \begin{axis}
      [
        ybar,
        enlargelimits=0.15,
        legend style={at={(0.4,-0.25)},
    anchor=north,legend columns=-1},
        ylabel={\#Annual Growth Percentage},
        symbolic x coords={2016, 2017,
    2018},
        xtick=data,
        nodes near coords,
        nodes near coords align={vertical},
      ]
      \addplot coordinates {(2016, 75)
    (2017, 78) (2018, 80)};
      \addplot coordinates {(2016, 70)
    (2017, 63) (2018, 68)};
      \addplot coordinates {(2016, 61)
    (2017, 55) (2018, 59)};
      \legend{Wheat, Tea, Rice}
    \end{axis}
```

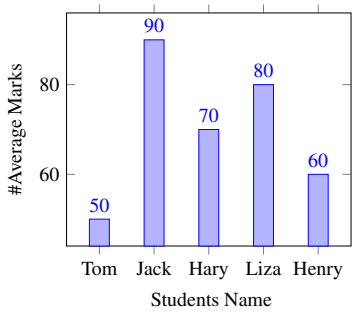
```
  \end{tikzpicture}
\end{wrapfigure}
Sem nulla pharetra diam sit amet. Vel
    pharetra vel turpis nunc eget.
    Vulputate dignissim suspendisse in est
    ante in nibh mauris cursus. Sem viverra
     aliquet eget sit amet tellus cras.
    Rhoncus aenean vel elit scelerisque
    mauris pellentesque pulvinar
    pellentesque. Fusce ut placerat orci
    nulla pellentesque. Vel risus commodo
    viverra maecenas accumsan lacus vel
    facilisis volutpat. Enim ut tellus
    elementum sagittis vitae et. In nibh
    mauris cursus mattis molestie.
    Curabitur gravida arcu ac tortor
    dignissim convallis aenean et tortor.
    Mauris commodo quis imperdiet massa.

\end{document}
```

SSC 335: barchart demo

Victor Eijkhout

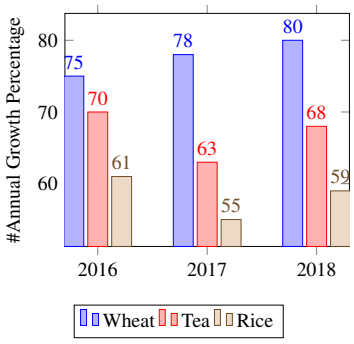
today

1 Two graphs



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Pharetra massa massa ultricies mi quis hendrerit. Tempor nec feugiat nisl pretium fusce id velit ut tortor. Eget nulla facilisi etiam dignissim diam quis enim. Cursus sit amet dictum sit amet justo donec. Tortor consequat id porta nibh venenatis cras sed felis eget. Senectus et netus et malesuada fames ac turpis egestas integer. Ultricies mi quis hendrerit dolor magna eget est. A iaculis at erat pellentesque adipiscing. Sagittis orci a scelerisque purus. Quisque non tellus orci ac. Nisl nunc mi ipsum faucibus. Vivamus at augue eget arcu dictum varius duis. Maecenas ultricies mi eget mauris pharetra et ultrices neque ornare. Pulvinar neque laoreet suspendisse interdum consectetur. Nunc id cursus metus aliquam eleifend mi. Tristique sollicitudin nibh sit amet commodo nulla. Massa tincidunt nunc pulvinar sapien et ligula ullamcorper malesuada. Justo laoreet sit amet cursus sit. Laoreet id donec ultrices tincidunt arcu non sodales.

Sem nulla pharetra diam sit amet. Vel pharetra vel turpis nunc eget. Vulputate dignissim suspendisse in est ante in nibh mauris cursus. Sem viverra aliquet eget sit amet tellus cras. Rhoncus aenean vel elit scelerisque mauris pellentesque pulvinar pellentesque. Fusce ut placerat orci nulla pellentesque. Vel risus commodo viverra maecenas accumsan lacus vel facilisis volutpat. Enim ut tellus elementum sagittis vitae et. In nibh mauris cursus mattis molestie. Curabitur gravida arcu ac tortor dignissim convallis aenean et tortor. Mauris commodo quis imperdiet massa.



14.4 Where to take it from here

This tutorial touched only briefly on some essentials of TEX and LATEX. You can find longer intros online [16], or read a book [12, 11, 15]. 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 [18].

14.5 Review questions

Exercise 14.13. 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 15

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. [Cited on page 35.]

[2] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammerling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997. [Cited on page 107.]

[3] Netlib.org BLAS reference implementation. http://www.netlib.org/blas. [Cited on page 107.]

[4] Yaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the fourth symposium on the frontiers of massively parallel computation (Frontiers '92), McLean, Virginia, Oct 19–21, 1992*, pages 120–127, 1992. [Cited on page 107.]

[5] Edsger W. Dijkstra. Programming as a discipline of mathematical nature. *Am. Math. Monthly*, 81:608–612, 1974. [Cited on page 134.]

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990. [Cited on page 107.]

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988. [Cited on page 107.]

[8] 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. [Cited on page 35.]

[9] Victor Eijkhout. *The Science of TEX and LATEX*. lulu.com, 2012. [Cited on page 39.]

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969. [Cited on page 134.]

[11] Helmut Kopka and Patrick W. Daly. *A Guide to LATEX*. Addison-Wesley, first published 1992. [Cited on page 185.]

[12] L. Lamport. *LATEX, a Document Preparation System*. Addison-Wesley, 1986. [Cited on page 185.]

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. [Cited on page 107.]

[14] 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. [Cited on page 51.]

[15] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LaTeX Companion, 2nd edition*. Addison-Wesley, 2004. [Cited on page 185.]

[16] Tobi Oetiker. The not so short introductino to LaTeX. http://tobi.oetiker.ch/lshort/. [Cited on pages 169 and 185.]

[17] Jack Poulson, Bryan Marker, Jeff R. Hammond, and Robert van de Geijn. Elemental: a new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. submitted. [Cited on page 107.]

[18] TeX frequently asked questions. [Cited on page 185.]

[19] R. van de Geijn, Philip Alpatov, Greg Baker, Almadena Chtchelkanova, Joe Eaton, Carter Edwards, Murthy Guddati, John Gunnels, Sam Guyer, Ken Klimkowski, Calvin Lin, Greg Morrow, Peter Nagel, James Overfelt, and Michelle Pal. Parallel linear algebra package (PLAPACK): Release r0.1 (beta) users' guide. 1996. [Cited on page 107.]

[20] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997. [Cited on page 107.]

[21] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best practices for scientific computing. *PLOS Biology*, 12(1):1–7, 01 2014. [Cited on page 7.]

Chapter 16

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
CPU Central Processing Unit
CRS Compressed Row Storage
CUDA Compute-Unified Device Architecture
DAG Directed Acyclic Graph
DL Deep Learning
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
IMP Integrative Model for Parallelism
IVP Initial Value Problem
LAPACK Linear Algebra Package
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
ML Machine Learning
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
SGD Stochastic Gradient Descent
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
SSOR Symmetric Successive Over-Relaxation
SP Streaming Processor
SPMD Single Program Multiple Data
SPD symmetric positive definite
SRAM Static Random-Access Memory
SSE SIMD Streaming Extensions
STL Standard Template Library
TLB Translation Look-aside Buffer
UMA Uniform Memory Access
UPC Unified Parallel C
WAN Wide Area Network

Chapter 17

Index

ISBN 978-1-257-99254-6
90000
9 781257 992546