

# Programming for today's physicist and engineer

ISOTDAQ 2015, Rio de Janeiro

January 28, 2015

Joschka Lingemann

CERN

RWTH Aachen - III. Physikalisches Institut B



Physics  
Institute III B



# Disclaimer

---

**Disclaimer:** This is more a collection of pointers\* than a tutorial  
it's a starting point... (Almost) no code

**Acknowledgment:** Slides are based on previous lectures by Erkcan Ozcan, see final slide for link

further reading and tips  
in these boxes

\* not in the C/C++ sense

# The Environment

---

## (Astro)particle, accelerator experiments and industry:

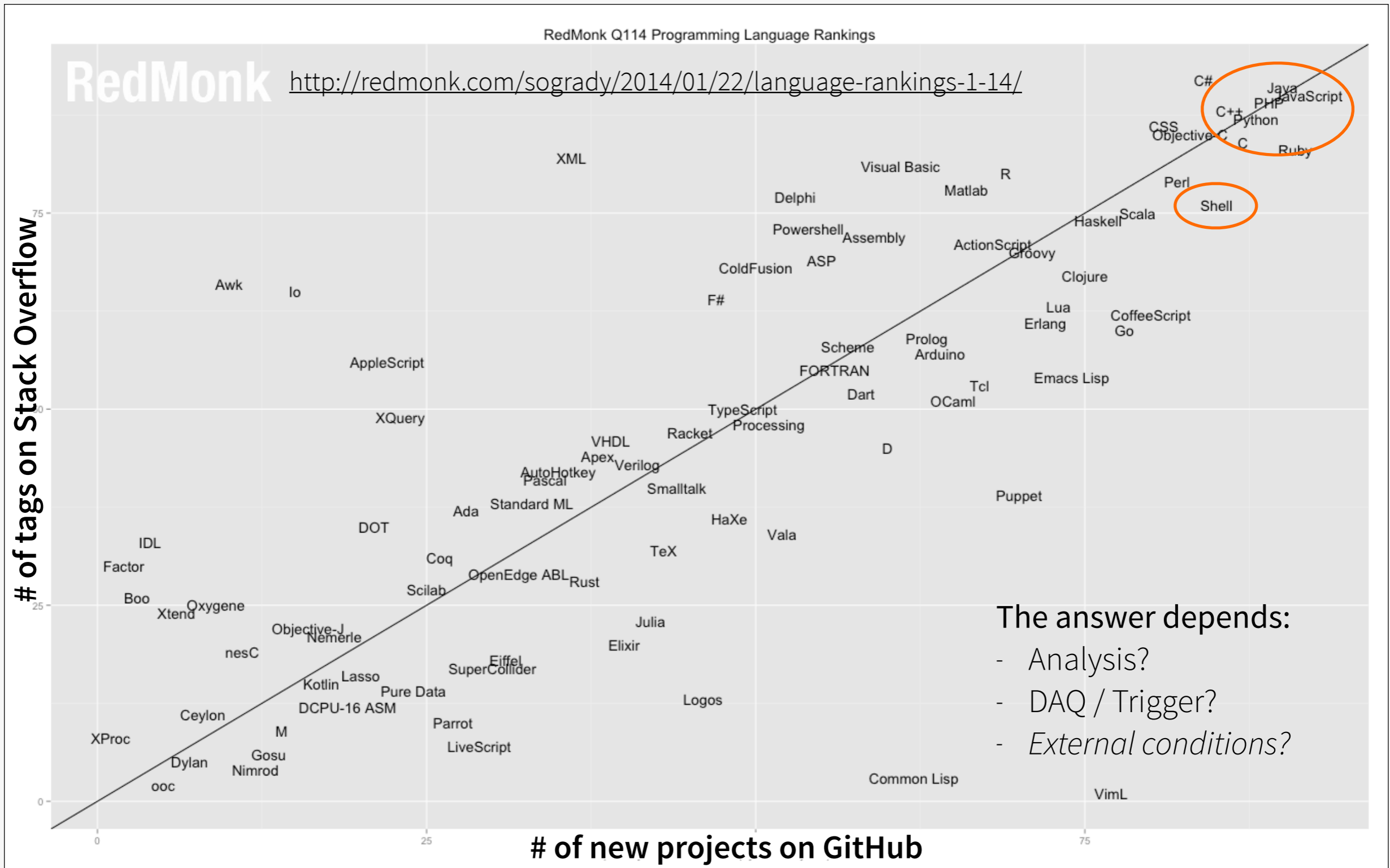
- (Large) collaborations, large sets of data, limited time
  - ▶ Code sharing / reuse
    - Revision control, reusability, portability
    - Learn new languages / libraries / tools
  - ▶ Code binding - framework integration
    - Often via (additional) scripting language
  - ▶ Documentation / visualisation
    - Doxygen, wikis, bug-tracking, UML, graphing, GUI
  - ▶ Working remotely
    - Batch, grid, cloud

**Part I:** How to avoid writing code twice  
(and how to avoid writing it at all)

---

# What Language should you learn? Language popularity?

N.b.: difficult to measure



# The C&P Technique — know what you're doing!

---

## Often you have existing code to start from

- Good: You know where to start and can repurpose
- Bad: It's easy to make mistakes if you don't understand the logic

## Short Example: Write a tool in C++ that gives you the number of days in a month

```
$ howmanydays april  
april has 30 days
```

You have an existing example that does something similar: Given a number it tells the user which month it corresponds to:

```
$ whichmonth 5  
The 5th month is may
```

# Getting to know the existing code

```
#include <iostream>
#include <unordered_map>

const char* suffix(unsigned int nm) {
    if (nm == 1) return "st";
    if (nm == 2) return "nd";
    if (nm == 3) return "rd";
    return "th";
}

using std::cout; using std::endl;
using std::unordered_map;

int main (int argc, char* argv[]) {
    if (argc != 2) return 1;
    int month = atoi(argv[1]);
    if (month < 1 || month > 12) return 1;
    unordered_map< int, const char* > months;
    months [1] = "january";
    months [2] = "february";
    months [3] = "march";
    months [4] = "april";
    months [5] = "may";
    months [6] = "june";
    months [7] = "july";
    months [8] = "august";
    months [9] = "september";
    months [10] = "october";
    months [11] = "november";
    months [12] = "december";

    cout << "The " << month << suffix(month);
    cout << " month is " << months[month] << endl;
    return 0;
}
```

You received the source:

- May learn something new...
- Try it out! Does it even work?

Hash maps: Associating unique identifiers “keys” with some values

- Used for fast searches, caching data
- Examples in Trigger / DAQ:
  - ▶ trigger algorithms
  - ▶ routing in networks

# Adaptation of the code

```
#include <iostream>
#include <unordered_map>

using std::cout; using std::endl;
using std::unordered_map;

int main (int argc, char* argv[]) {
    // if (argc != 2) return 1;
    // int month = atoi(argv[1]);
    // if (month < 1 || month > 12) return 1;

    unordered_map< const char*, int > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    cout << "march : ndays = " << months["march"] << endl;
    cout << "june  : ndays = " << months["june"] << endl;
    cout << "april : ndays = " << months["april"] << endl;

    return 0;
}
```

Checked documentation for hasher  
of char\*

As seen in code: Checked with  
examples

```
$ clang++ howmanydays.cxx
$ ./a.out
march : ndays = 31
june  : ndays = 30
april : ndays = 30
```

All I need now is to accept user input.



# Alright. Finished program, let's bring it to the people

```
#include <iostream>
#include <unordered_map>

using std::cout; using std::endl;
using std::unordered_map;

int main (int argc, char* argv[]) {
    if (argc != 2) return 1;

    unordered_map< const char*, int > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    cout << argv[1] << " has " << months[argv[1]];
    cout << " days" << endl;

    return 0;
}
```

One problem: assuming enduser is well behaved..

- Never, ever do that! Not even if you are the enduser...

# Alright. Finished program, let's bring it to the people

```
#include <iostream>
#include <unordered_map>

using std::cout; using std::endl;
using std::unordered_map;

int main (int argc, char* argv[]) {
    if (argc != 2) return 1;

    unordered_map< const char*, int > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    cout << argv[1] << " has " << months[argv[1]];
    cout << " days" << endl;

    return 0;
}
```

One problem: assuming enduser is well behaved..

- Never, ever do that! Not even if you are the enduser...

One other problem:

- It does not work:

```
$ g++ test.cxx
```

```
./a.out june
```

```
june has 0 days
```

# Ah, right... C-strings, comparison is problematic

```
#include <iostream>
#include <unordered_map>

using std::cout; using std::endl;
using std::unordered_map;

struct string_equal {
    bool operator() (const char* str1, const char* str2) const
    {
        return std::strcmp(str1, str2) == 0;
    }
};

int main (int argc, char* argv[]) {
    if (argc != 2) return 1;

    unordered_map< const char*, int,
        std::hash< const char* >, string_equal > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    cout << argv[1] << " has " << months[argv[1]];
    cout << " days" << endl;

    return 0;
}
```

## Check documentation:

- Can provide a comparison function
- Needs additional template argument: hash-function
  - ▶ We can use default std::hash

```
$ g++ test.cxx
```

```
./a.out june
```

```
june has 31 days
```

Finally done...

# Ah, right... C-strings, comparison is problematic

```
#include <iostream>
#include <unordered_map>

using std::cout; using std::endl;
using std::unordered_map;

struct string_equal {
    bool operator() (const char* str1, const char* str2) const
    {
        return std::strcmp(str1, str2) == 0;
    }
};

int main (int argc, char* argv[]) {
    if (argc != 2) return 1;

    unordered_map< const char*, int,
        std::hash< const char* >, string_equal > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    cout << argv[1] << " has " << months[argv[1]];
    cout << " days" << endl;

    return 0;
}
```

## Check documentation:

- Can provide a comparison function
- Needs additional template argument: hash-function
  - ▶ We can use default `std::hash`

```
$ g++ test.cxx
```

```
./a.out june
```

```
june has 30 days
```

## Finally done... Try it with your colleagues:

```
june has 0 days
```

why? see additional material

# Avoid mixing C and C++ styles: Stick to the STL

```
#include <iostream>
#include <unordered_map>
#include <string>

using std::cout; using std::endl;
using std::unordered_map;
using std::string;

int main (int argc, char* argv[]) {
    if (argc != 2) return 1;

    unordered_map< string, int > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    cout << argv[1] << " has " << months[argv[1]];
    cout << " days" << endl;

    return 0;
}
```

## Clean solution with STL:

- Know and use the STL consistently
- Invest some time to learn about it:  
Saves you in the long run
- Containers: Many corner cases covered for you
- They are safer - buffer overflows, thread-safety and more

Finally we are really done: Short code, portable (no dependencies) and works correctly

# Final Code: Don't forget to document!

```
#include <iostream>
#include <unordered_map>
#include <string>
using std::cout; using std::endl;
using std::unordered_map;
using std::string;

int main (int argc, char* argv[]) {
    // don't do anything if no argument is given
    if (argc != 2) return 1;

    // use string instead of const char*!
    // workarounds for const char*: need to implement
    // specialised hash-function.
    unordered_map< string, int > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    // assuming user is well behaved:
    // not implemented catches! i.e. non-existing months
    cout << argv[1] << " has " << months[argv[1]];
    cout << " days" << endl;

    return 0;
}
```

## Documentation is vital!

- Internal documentation
  - ▶ Describe your interfaces
  - ▶ Describe your reasoning
- External documentation
  - ▶ Give use-cases, examples
  - ▶ Describe the “big picture”

It will also save you when you re-use this in a few months / years

# Pre-summary: Make your code easy to understand

---

## External conditions:

- We have little time
- Likely someone else will have to take over at some point
- Work together with others

## It all boils down to one basic principal:

- Make your code easy to understand
  - ▶ Therefore easy to re-use
  - ▶ Easier to maintain

# Make re-using your code easy: Documentation

Document your code while you write it! For your own benefit.

---

## Generally two sides of the same coin: Internal and external documentation

- Both are necessary to make your programs easy to use
- They have different purpose!

### Internal documentation:

- Explain interfaces, i.e. function signatures
- Make note of possible future problems (better: prevent them)
- Sometimes might be good to document your reasoning
- Do not “over-comment”

```
def isolation(pt, et_sums, cut, rel=False):  
    """  
    Calculates if particle is isolated  
    TAKES: pt - pt of the particle (float)  
           et_sums - calorimeter transverse energy (L  
           cut - the maximum isolation value allowed  
           rel - calculate relative iso (= et/pt)? (b  
    RETURNS:  
           bool - whether particle is isolated  
    """
```

```
if a > b: # when a is greater than b do:
```

### External documentation:

- Again: Explain your interfaces (can be derived from internal, e.g. [doxygen.org](http://doxygen.org))
- For large projects: The big picture
  - ▶ Wiki pages with use-cases and examples
  - ▶ Consider using UML (unified modelling language)



# Make re-using your code easy: Documentation

Document your code while you write it! For your own benefit.

---

## Generally two sides of the same coin: Internal and external documentation

- Both are necessary to make your programs easy to use
- They have different purpose!

### Internal documentation:

- Explain interfaces, i.e. function signatures
- Make note of possible future problems (better: prevent them)
- Sometimes might be good to document your reasoning
- Do not “over-comment”

```
def isolation(pt, et_sums, cut, rel=False):  
    """  
    Calculates if particle is isolated  
    TAKES:  pt - pt of the particle (float)  
           et_sums - calorimeter transverse energy (L  
           cut - the maximum isolation value allowed  
           rel - calculate relative iso (= et/pt)? (b  
    RETURNS:  
           bool - whether particle is isolated  
    """
```

```
if a > b: # when a is greater than b do:
```

### External documentation:

- Again: Explain your interfaces (can be derived from internal, e.g. [doxygen.org](http://doxygen.org))
- For large projects: The big picture
  - ▶ Wiki pages with use-cases and examples
  - ▶ Consider using UML (unified modelling language)

# Make re-using your code easy: Sharing

\* paraphrasing Linus Torvalds

Revision control: Important for you, important for colleagues

Basic: CVS and Subversion (“CVS done right”\*)

Distributed revision control: Must for personal use (for working on the go)

- Your local copy has everything (including history)

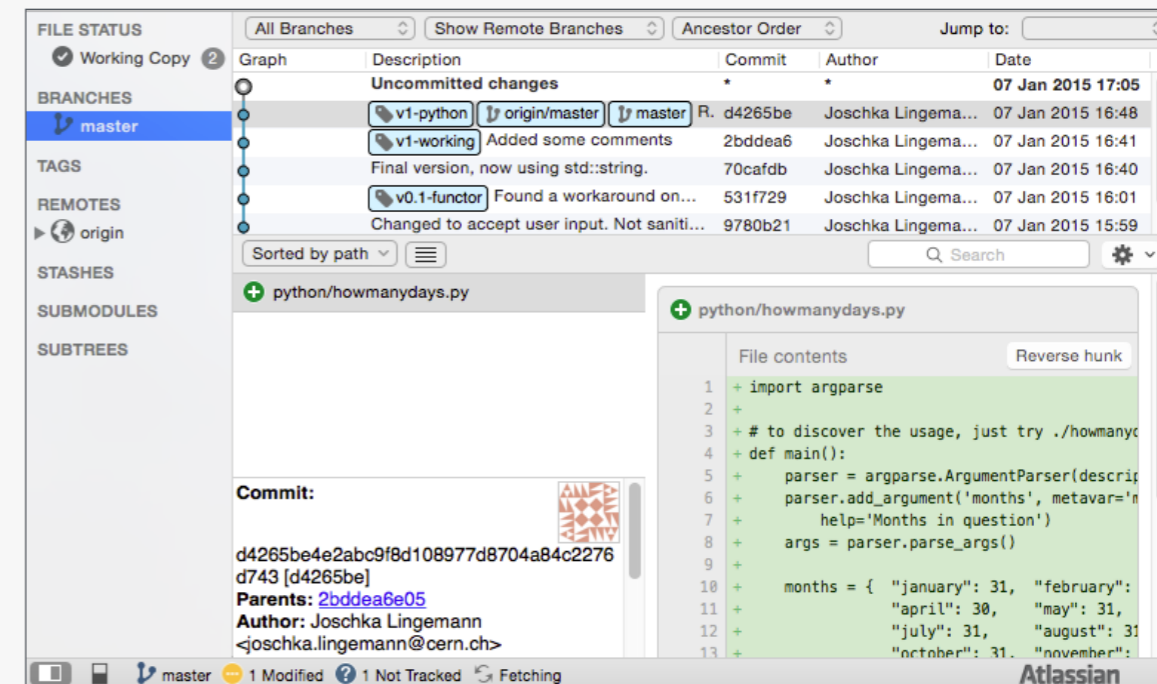
Gaining ever more popularity “git”: [git-scm.com](http://git-scm.com)  
 (“there is no way to do CVS right”\*)

- Other solutions are: Mercurial, bazaar and more
- Easy to learn:

```
$ git init
```

```
$ git add test.cc
```

```
$ git commit -m “initial commit”
```



visualisation with SourceTree, often available gitk

<http://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>  
<http://pcottle.github.io/learnGitBranching/>

# Make re-using your code easy: Sharing (cont.)

---

## Git is widely used, examples:

- CMSSW (CMS software), ROOT
- Microsoft, Apple, Amazon, Google, LG
- Linux Kernel, GNOME, Android, KDE
- and many more



## Easy to host & share your projects:

- Setting up a shared repo can be done via any cloud service, e.g. dropbox
- many open-source hosting sites, biggest: [github.com](https://github.com)
  - ▶ Includes fairly usable bug-tracking

**The more you learn the more you'll like it!**

possibly interesting:  
<https://education.github.com/pack>

# Make re-using your code easy: Organise your code

---

## Have a meaningful directory structure (especially in repos)

- if you use github: try playing around with the markdown for README.md

## Separate projects that do not belong together!

## When compiling software:

- Use meaningful folder names — version numbers
- `./configure --help` to learn how to install into custom folders
- Use softlinks if you like to store multiple versions of code

```
drwxr-xr-x 49 joschka staff 1.6K Dec 17 15:44 root6.02.02
lrwxr-xr-x  1 joschka staff  11B Jan  6 11:42 root -> root6.02.02
dr-xr-xr-x 49 joschka staff 1.6K Jan 13 09:50 root5.34.04
```

- Permissions: Minimum rights as usual

github markdown:  
<https://help.github.com/articles/markdown-basics/>

# Make re-using your code easy: Portability

## Makefiles — makes compilation easier

- Makefiles might look complex
- More than one source file: Useful!
  - ▶ Again: Think about compiling it in 2 years
- Write your own for a small project
- Automatically allows parallel compilation (option `-j`)

```
CC=clang++
CFLAGS=-Wall -pedantic -std=c++11
SOURCES=src/howmanydays.cc
OBJECTS=$(SOURCES:.cc=.o)
EXE=howmanydays

all: $(SOURCES) $(EXE)

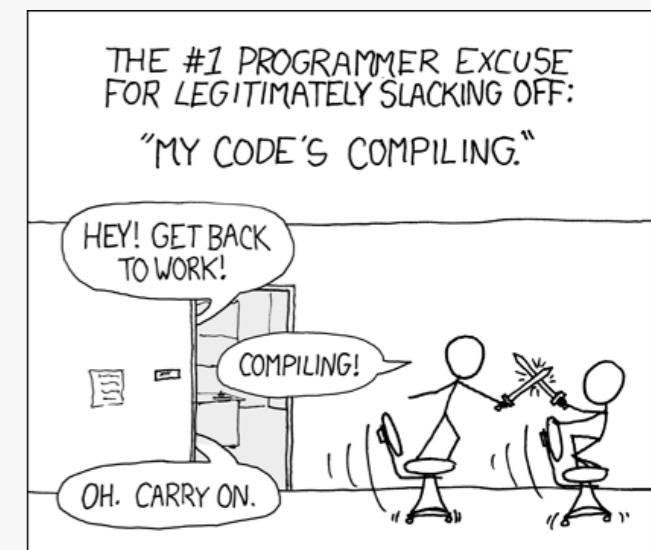
$(EXE): $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) -o bin/$@

%.o: %.cc
    $(CC) $(CFLAGS) -c -o $@ $<

.PHONY: clean all
clean:
    rm -f $(OBJECTS) bin/$(EXE)
```

## Alternatives and improvements to makefiles: CMake, autoconf and more

- Might look like overkill; Makes things easier in the long run
- CMake is easier to read and better documented
- Improved **portability**
- At least you should learn how to compile with them



“Compiling” by Randall Munroe  
[xkcd.com](http://xkcd.com)

# Make re-using your code easy: Conciseness

## Use the right tool

---

### Make your code as short as possible while maintaining readability

- For some solutions that means to use the right language
- Often quicker and nicer to use interpreted languages: python, perl, ruby, tcl, lua
- Often used as binding languages: Performance critical code in C/C++ modules instantiated within python (e.g. in CMSSW) — best of both worlds
- Personal choice: Python has a large standard library and is very expressive!

- dict example in python:  
more compact code, does more stuff
- have a look at ipython

#### try it yourself:

```
$ ipython
In [1]: run howmanydays.py -h
In [2]: run howmanydays.py may
In [3]: help(months)
```

```
import argparse

parser = argparse.ArgumentParser(description='Get the number of days in a month.')
parser.add_argument('months', metavar='month', type=str, nargs='+',
                    help='Months in question')
args = parser.parse_args()

months = { "january": 31, "february": 28, "march": 31,
           "april": 30, "may": 31, "june": 30,
           "july": 31, "august": 31, "september": 30,
           "october": 31, "november": 30, "december": 31 }

for usermonth in args.months:
    if usermonth in months:
        print ("{month} has {n} days.".format(month=usermonth, n=months[usermonth]))
    else:
        print ("sorry month '{month}' not known.".format(month=usermonth))
```

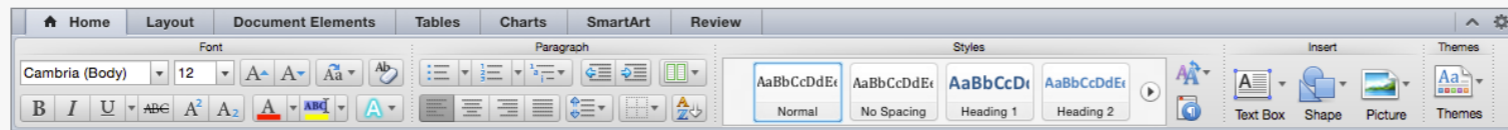
# Make re-using your code easy: Conciseness

## Avoid feature creeps

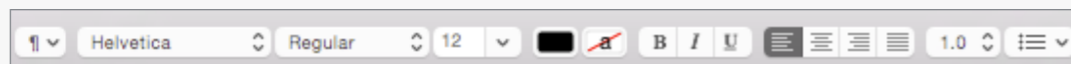
---

### If you try to do everything at once:

- You'll probably end up doing nothing right



MS Office '11 Word



Apple TextEdit

- Generalising a problem before solving it: Probably not a good idea
  - ▶ Only do it when you have a use case
- Write dedicated tools / libraries
- Be pragmatic:
  - ▶ Only do the abstract cases when it is likely that they will be used
  - ▶ Try to make everything as concise as possible (maintain readability)
  - ▶ Keep it simple!

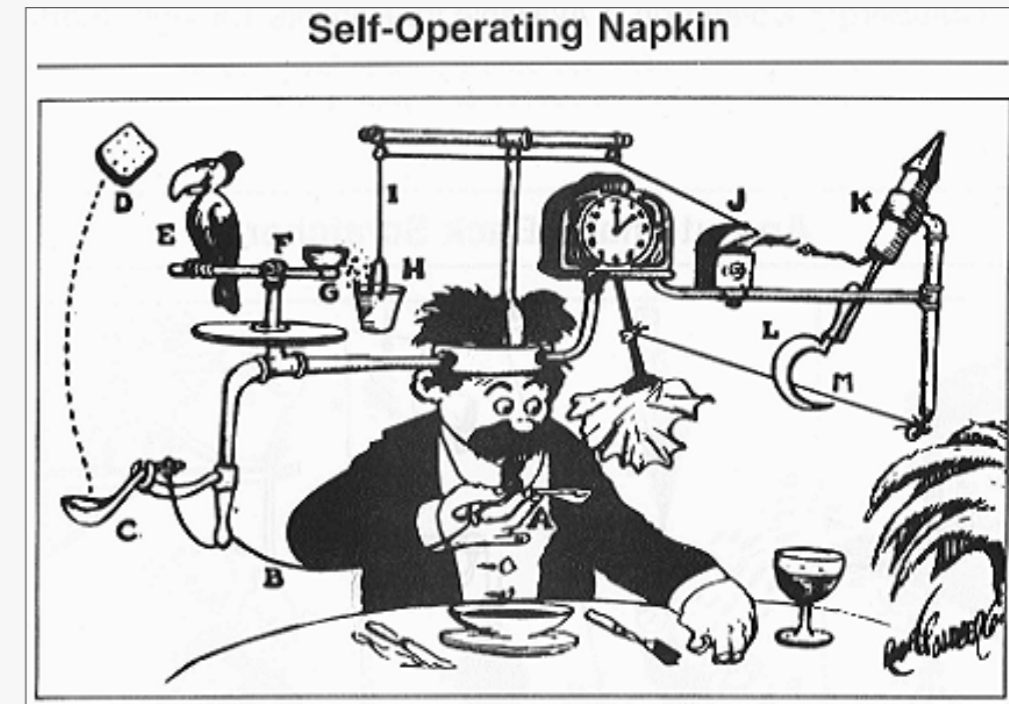
# Re-using other's code

## Do not reinvent the wheel

- Many problems have already been solved
- (Sometimes necessary — avoid dependencies)
- In languages with big standard lib: Look there!
- When using external libraries, look out for:
  - ▶ Active community? Well maintained?
  - ▶ Tested?

## Getting to know new frameworks:

- Before asking for advice: try the simple tools
  - ▶ Read the docs
    - Investing time in the beginning will pay off
  - ▶ StackOverflow is your friend
  - ▶ Are there TWikis?
  - ▶ python packages: try the ipython “help”



“Prof. Lucifer Butts and his Self-Operating Napkin”,  
by Rube Goldberg

- Start with a simple test (work your way from the existing examples)
  - ▶ Does the code do what you expect?



**Part II:** When you have to write code  
(tools to make your life easy)

---

# Be efficient — know your tools

---

## One side of using the right tool for the right problem:

- Interpreted vs compiled languages
- Existing toolkit vs implementing yourself

## Whatever you do, you'll end up using (at least)

- Editor
  - ▶ Know\* at least one “always” present editor: nano, vi(m), emacs, etc.
  - ▶ But also have a look at more modern solutions, they may have some benefits
  - ▶ Depending on the language / platform, IDEs are a better choice (Java, Python(?))
- Terminal
  - ▶ Learn about shortcuts (`ctrl+r`, `ctrl+e`, `ctrl+a` ... have a look)
  - ▶ Knowing about some basic command line-tools can come in handy

\* at least know how to save and exit :)  
for the more daring: try `ed`

A few words on editors: Many choices  
Whichever you choose, learn to be efficient!

---

### The choice of editor is yours...

- Do you want “a great operating system, lacking only a decent editor”
- Or one with two modes: “beep constantly” and “break everything” \*

Both are versatile and learning them is worthwhile



VS



### However: Alternatives exist that have a less steep learning-curve

- Most of them have been closed and expensive (TextMate, Sublime Text)
- Open alternative: Atom, <https://atom.io/>
  - ▶ Con: footprint, startup time
  - ▶ Pro: Integrated git diffs, active community, many plugins...

### Once you decided which one is best for you:

- Spend some time learning about it's features and keybindings
- Many things that might require dozens of keystrokes can be done with 2 (5 in emacs ;))
- Learn about: Linters, extensibility — look at existing plugins

Atom on MacOS: Don't forget to Install Shell  
Commands (after moving to final dest)

\* from [http://en.wikipedia.org/wiki/Editor\\_war](http://en.wikipedia.org/wiki/Editor_war)

# The terminal: Biggest foe or best friend?

---

## At the beginning might think: Quicker with GUI, don't need terminal

- After learning about some command line tools... probably not
- What if you don't have a GUI?

## Searching for files / something in files: grep, find.. example:

```
$ find . -name "*.cc" -exec grep -A 3 "foo" {} +
```

- Displays all matches of “foo” (+3 lines below) in all .cc files from the current work dir

## Once you learn about some of the small wheels you can build big machines:

- **sed, head, tail, sort... awk** (a turing-complete interpreted language)
- At the beginning: note down often used commands...
- After a tutorial dump your history\* (increase cache size for max usage)

```
* dump the last 100 steps:  
$ history | tail -n 100 > steps.txt  
log the terminal “responses”:  
$ script # press ctrl+d to stop
```

## Shell-scripting:

- Alternative: Can solve most things more conveniently with an interpreted language
- Con: interpreters might not always be available

```
tune your bashrc / bash-profile  
see additional material
```

# After writing code:

## Debugging, profiling and static analysis

---

### While running your code:

- `cout` / `printf` statements: only suitable for small code base
- Sooner or later have to use a debugger: **gdb** (GNU debugger) — get a stack-trace
  - ▶ basic commands: `run`, `bt`, `info <*>`, `help`
- Most crashes from memory management: Only use raw pointers when you have to! (I.e. when you know what you're doing and you need the performance)
  - ▶ Look at smart pointers (part of C++11/14 standards, alternative: boost)
- Even if you don't have crashes: Memory Leaks. Try **valgrind** ([valgrind.org](http://valgrind.org))
- Python: `pdb` — `import pdb; pdb.set_trace()` #set a breakpoint

### While writing your code:

- There are static code analysis tools that can help you
- Try out a linter for your preferred editor (e.g. atom: <https://atom.io/packages/linter>)
  - ▶ Highlights potentially problematic code— your code will be more reliable

# Need GUI Solution?

Web interfaces might be the easiest solution...

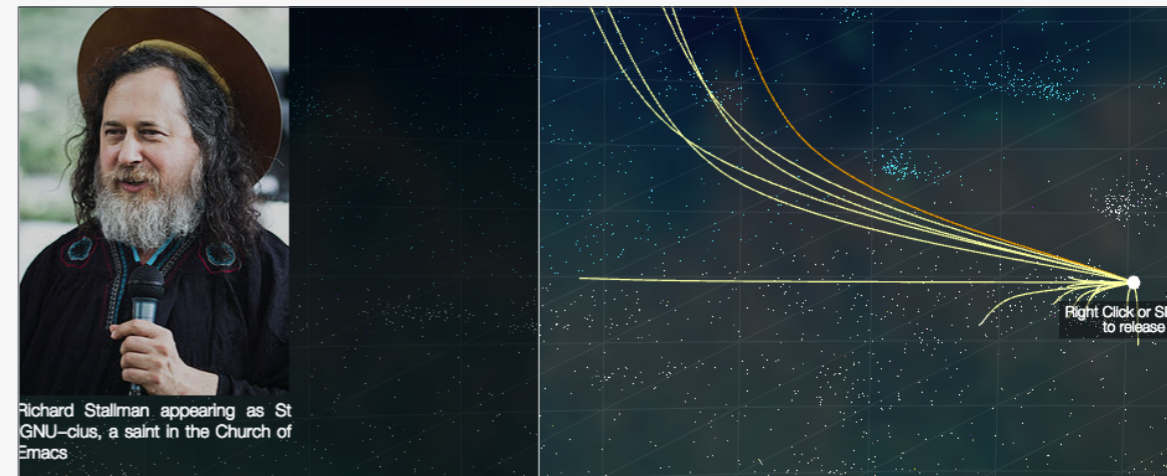
## When you need a graphical user interface:

- Web browsers are present everywhere
- If you do it right get phone / tablet support for free\*
- HTML5 - a lot of potential, a lot of frameworks
  - ▶ WebGL: GPU support (want to make a new event display?  
<http://ispy-webgl.web.cern.ch/ispy-webgl/>)
  - ▶ There are also tools to compile Java, Python, even C++ into JS

\* try <http://getbootstrap.com/>  
if you want to get something sleek quickly

## Some pointers to interesting / fun projects:

- <http://www.skulpt.org/> — python interpreter
- <http://gcc.godbolt.org/> — C++ compiler to assembly
- <http://webglplayground.net/> — play with shaders
- <http://wiki.polyfra.me/> — wikipedia visualisation
- <http://www.chromeexperiments.com/>
  - ▶ many more



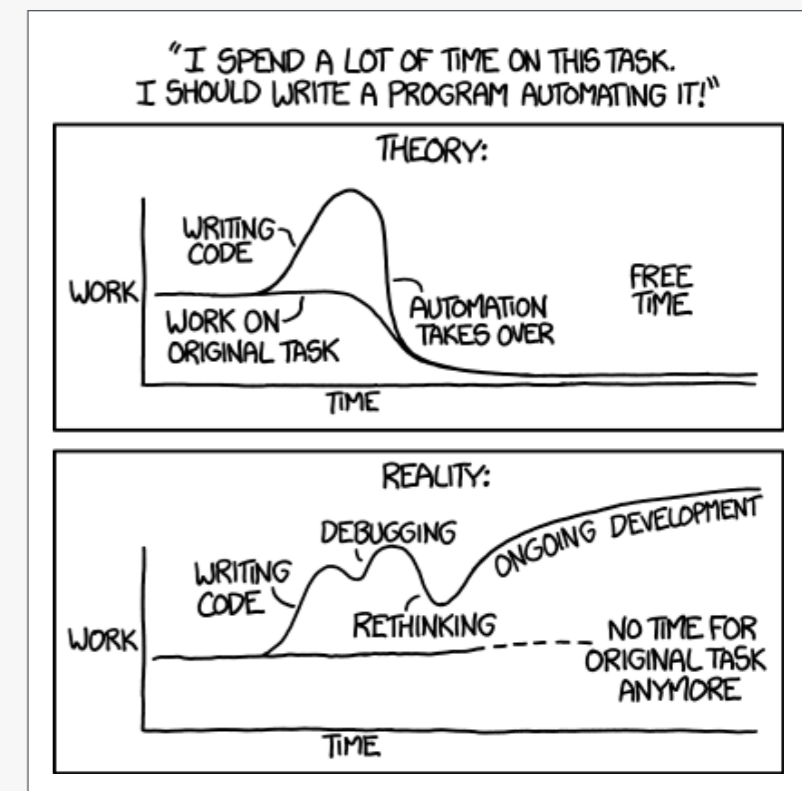
# Automation and batch systems..

If you have programs that have to run with slightly different parameters:

- Do not change the variables in the source code! (It's error prone, might need to recompile, etc)
- Introduce a way to configure from the terminal:
  - ▶ Command line options / arguments (a few variables)
  - ▶ Configuration file (many variables)
    - XML, JSON, CSV .. choices a plenty

Batch systems: Probably your home institute has one installed

- Learn about available resources
  - GRID — get a certificate
  - Your laptop has multiple cores.. Can also utilise without threading:
    - ▶ e.g. task spooler - <http://vicerveza.homeunix.net/~viric/soft/ts/>
- ```
$ export TS_MAXCONN=20
$ export TS_SLOTS=<# cores>
$ ts
$ ts <job>
```



"Automation" by Randall Munroe  
[xkcd.com](http://xkcd.com)

# Working remotely: No way around the terminal

## SSH — might be more versatile than you think:

- Tunneling
  - ▶ Secure connections to other machines
  - ▶ Should also be used when using **VNC** to **avoid man-in-the-middle vulnerability**
- Using public keys as authentication
  - ▶ Generate with `ssh-keygen -t dsa`, copy your public key (never the `id_dsa` key):  
Put public key into `~/.ssh/authorized_keys` on remote host
- Working through X-forwarding can be annoying if you have bad latency / shaky connection
- Alternative: mosh (<https://mosh.mit.edu/>)
  - ▶ allows intermittent connectivity, roaming and more...

## SSHFS — mounting your remote work directories:

- On MacOSX: MacFusion + OSXFuse
- Work locally but have files live in remote host

## AFS — even more convenient:

- Don't forget to set permissions correctly while sharing files
- OpenAFS also available for Mac OS X (Yosemite: Have to hack the installer if you need help, come see me)

### SSH tunnel for VNC connection:

```
ssh -L 5902:<VNCServerIP>5902 <user>@<remote>  
vncserver :<session> -geometry <width>x<height>  
-localhost -nolisten tcp
```

### SSH authentication via kerberos token. In `~/.ssh/config`:

```
GSSAPIAuthentication yes  
GSSAPIDelegateCredentials yes  
HOST lxplus*  
    GSSAPITrustDns yes
```

### Lots of things possible with the ssh-config:

```
HOST <host>  
    USER <remote-user>  
    ProxyCommand ssh <tunnel> nc <host> <port>
```

### more on (auto-)tunnelling:

[https://security.web.cern.ch/security/recommendations/en/ssh\\_tunneling.shtml](https://security.web.cern.ch/security/recommendations/en/ssh_tunneling.shtml)



# Both while working remotely or locally: Screen can come in very handy

## screen : part of GNU, present in almost all \*nix

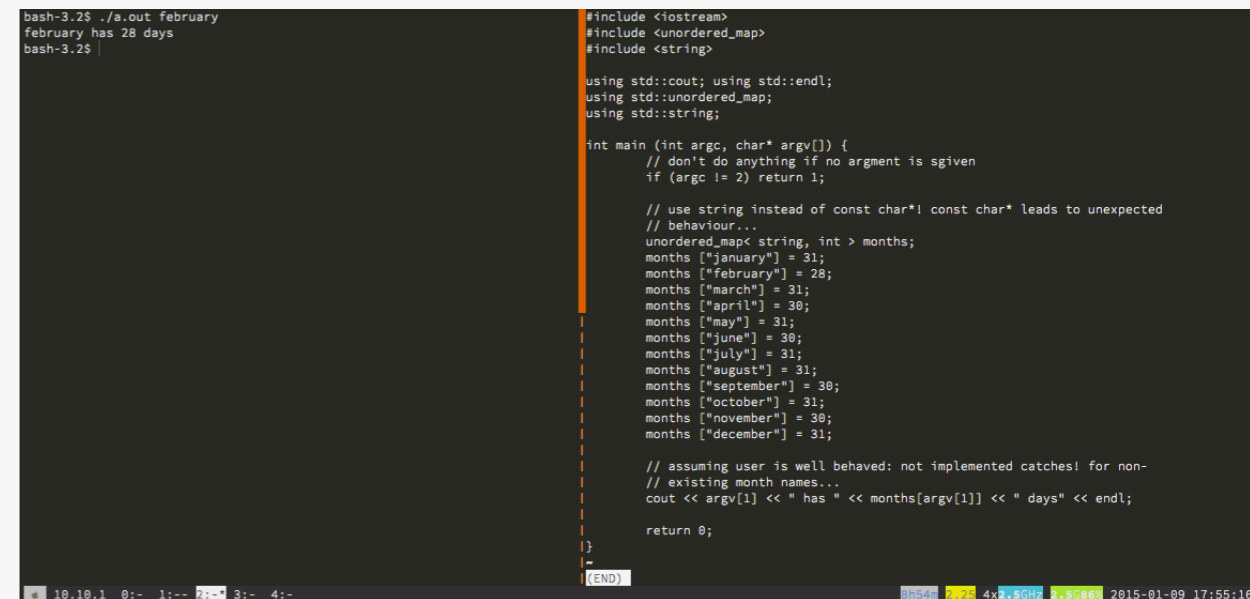
- Creates a virtual terminal
  - ▶ That do not die when connection is lost:
    - X session crashes, while working via ssh
  - ▶ The processes keep working after you logged off
- When you try it: Also try visualiser e.g. byobu (<http://byobu.co/>)
- Alternative: **tmux**

### Try it out — screen:

```
$ screen
> # start something
ctrl+a d
$ screen -ls
$ screen -r
> # continue where you were
> # get help:
ctrl+a ?
```

### Try it out — tmux:

```
$ tmux
> # start something
ctrl+b d
$ tmux ls
$ tmux a
> # continue where you were
> # get help:
ctrl+b ?
```



```
bash-3.2$ ./a.out february
february has 28 days
bash-3.2$

#include <iostream>
#include <unordered_map>
#include <string>

using std::cout; using std::endl;
using std::unordered_map;
using std::string;

int main (int argc, char* argv[]) {
    // don't do anything if no argument is given
    if (argc != 2) return 1;

    // use string instead of const char*! const char* leads to unexpected
    // behaviour...
    unordered_map< string, int > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    // assuming user is well behaved: not implemented catches! for non-
    // existing month names...
    cout << argv[1] << " has " << months[argv[1]] << " days" << endl;

    return 0;
}
```

**Part III:** More tools that can help you  
(maybe there is already a solution)

---

# Software collection: Analysis / Statistics

---

## In HEP probably no way around ROOT / RooFit

- Maintained at CERN, used in LHC experiments

## GNU R — [www.r-project.org](http://www.r-project.org)

- Used widely among statisticians (including finance and others)
- Interpreted language + software for analysis and graphical representation

## SciPy — <http://www.scipy.org/>

- Collection of python libraries for numerical computations, graphical representation and containing additional data structures

## Sci-kitlearn: — <http://scikit-learn.org/stable/>

- Python library for machine learning

# Software collection: Visualisation

---

## Data visualisation:

### Matplotlib (part of SciPy)

- histograms, power spectra, scatterplots and more.. extensive library for 2D/3D plotting

### ROOT

- Again, probably no way around it... Sometimes a little unintuitive

## Other:

### JaxoDraw — <http://jaxodraw.sourceforge.net/>

- Feynman graphs through “axodraw” latex package

### tex2im — <http://www.nought.de/tex2im.php>

- Need formulas in your favourite WYSIWYG presentation tool?

### GraphViz — <http://www.graphviz.org/> or MacOS: <http://www.pixelglow.com/graphviz/>

- Diagrams / Flowcharts with auto-layout

# Software collection: Math & more

---

## SAGE — [www.sagemath.org](http://www.sagemath.org)

- Open source alternative to Matlab, Maple and Mathematica

## GNUPlot — <http://www.gnuplot.info/>

- Quick graphing and data visualisation

## Wolfram Alpha — <http://www.wolframalpha.com/>

- Wolfram = Makers of Mathematica.. A... ask me anything?:
  - ▶ <http://www.wolframalpha.com/input/?i=how+much+does+a+goat+weigh>
  - ▶ Answer: Assuming “goat” is a species specification. Result: 61 kg

# Conclusion

---

**This lecture was full of starting points: You have to follow up to get something out of it.**

- Most of it are tools to make your life easier
  - ▶ Bonus: If you know them you'll have an easier time to follow nerd-talk
- Nothing is free
  - ▶ You'll have to invest some effort to learn
  - ▶ If you do that this week: We'll be here to help!

## Homework:

- Install git, start a repository. Try branching on the web
- Compile and play with examples given (python + C++)
  - ▶ See what doesn't work
- Run screen, kill the connection, reconnect and see if you can continue where you left off
- Tune your `.bashrc` / `.bash_profile` to get a more useful prompt
- Try out vim / emacs / atom and learn what suits you best — download a shortcut summary...  
Learn how to block-select, indent multiple lines, rename occurrences of text

# Advice

---

**Before you write trigger / DAQ software, you should know the ins and outs:**

- What is: compiler, interpreter, linker, terminal, object, class, pointer, reference
- If these concepts are not clear: Excellent material on the web (see next slide)

**Before (and while) implementing: Think**

- Smart solutions can take significant amount of time... put it on the back-burner if you have other things to work on

**Read! Read! Read!** The internet is full of information... Blogs, tutorials, StackOverflow, also Wikipedia can be very useful to get a grasp of new concepts

# Internet courses on programming (and more)

---

## Udacity — courses from industry (Google, Intel, Autodesk)

- <https://www.udacity.com/courses#!/all>
  - ▶ course material is free (videos + exercises), tutoring for monthly fees
- Growing catalogue beginner to advanced — mostly web-centric
  - ▶ JavaScript + HTML5 + AJAX courses etc
  - ▶ But also: Intro to git, data analysis with R, parallel programming ...

## Coursera — courses by universities (Caltech, Johns Hopkins, Stanford and more)

- <https://www.coursera.org/courses>
- Large variety of courses
  - ▶ Not only technology / programming
  - ▶ Also physics, biology, economics... and more
  - ▶ Also in different languages

## University Homepages — have a gander... many courses available through YouTube etc.

- i.e.: <https://www.youtube.com/watch?v=Ps8jOj7diA0&feature=PlayList&p=9D558D49CA734A02&index=0>

<http://ureddit.com/> — University of Reddit



# Random Quotes & Links

## 6 Stages of Debugging:

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?
  - <http://plasmasturm.org/log/6debug/>

Want to try your programming skills?  
Google code jam (registration 10.03.15):  
<https://code.google.com/codejam>

Also you can just practice  
by solving nice problems.

Go-language: Designed with threading in mind  
<http://tour.golang.org/welcome/1>

like the fonts in the presentation?  
<https://github.com/adobe-fonts/source-code-pro>  
<https://github.com/adobe-fonts/source-sans-pro>

Random github commit messages:  
<http://whatthecommit.com/>

Guru of the Week: Regular C++  
programming problems with solutions  
by Herb Sutter  
<http://www.gotw.ca/gotw/>

About JavaScript:  
<https://www.destroyallsoftware.com/talks/the-birth-and-death-of-javascript>  
<https://www.destroyallsoftware.com/talks/wat>

“Debugging is like being the  
detective in a crime novel where you  
are also the murderer.”

– @fortes

Last year's lecture may have some complementary stuff:  
<http://indico.cern.ch/event/274473/session/21/material/0/0.pdf>

# Additional Material

---

# ~/.bashrc : An example.

```
# tune your prompt:
if [ "$PS1" ]; then
    PS1="\[\033[1;29m\]\[\033[0;34m\] \u\[\033[0;34m\]@\[\033[1;34m\]\h : \[\033[0m\]: \w \[
\033[0;36m\] \$(git branch 2>/dev/null | grep '^*' | colrm 1 2) \[\033[0m\] ] \n \[\033[0;31m\]\$\[
\033[0m\] "
fi

# do not put duplicate lines into history:
export HISTCONTROL="ignoredups"

# default to human readable file sizes
alias df='df -h'
alias du='du -h'

# get some color
alias grep='grep --color'

# more file listing:
alias l='ls'
alias ll='ls -lt -h -G -c -r'

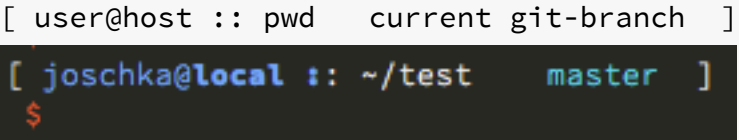
# fool proof cp - asks for each file, use fcp if you're sure
alias fcp='cp'
alias cp='cp -i -v'

# never remember those..
alias untgz='tar -xvzf'
alias tgz='tar -pczf'

#never install root:
source /path/to/your/working/root/bin/thisroot.sh
alias root='root -l'

# Mac OS stuff
alias wget='curl -O'
```

resulting prompt



```
[ user@host :: pwd current git-branch ]
[ joschka@local :: ~/test master ]
$
```

# Functional Fibonacci (Haskell)

---

```
fibonacci = 0:1:zipWith (+) fibonacci (tail fibonacci)
main = print (fibonacci !! 100)
```

And that's it — it's even fast

- we concatenate 0 and 1 with an infinite list of recursive sums
- At run-time the lazy evaluation makes sure the list is only created up to element 100

“I love functional programming. it takes smart people who would otherwise be competing with me and turns them into unemployable crazies”

— William Morgan (@wm)

# So.. Why does this not work?

```
#include <iostream>
#include <unordered_map>

using std::cout; using std::endl;
using std::unordered_map;

struct string_equal {
    bool operator() (const char* str1, const char* str2) const
    {
        return std::strcmp(str1, str2) == 0;
    }
};

int main (int argc, char* argv[]) {
    if (argc != 2) return 1;

    unordered_map< const char*, int,
        std::hash< const char* >, string_equal > months;
    months ["january"] = 31;
    months ["february"] = 28;
    months ["march"] = 31;
    months ["april"] = 30;
    months ["may"] = 31;
    months ["june"] = 30;
    months ["july"] = 31;
    months ["august"] = 31;
    months ["september"] = 30;
    months ["october"] = 31;
    months ["november"] = 30;
    months ["december"] = 31;

    cout << argv[1] << " has " << months[argv[1]];
    cout << " days" << endl;

    return 0;
}
```

## How does look up in unordered\_map work?

- keys are associated to buckets depending on the hash (bucket: each 0-1 elements, depending on implementation more than 1 is allowed)
- upon look-up: hash the key and go to bucket to retrieve value
- the **predicate is not used** here!

## When is the predicate used?

- The `string_equal` is actually only called when we add elements! (More precise: When re-hashing)

## The actual problem was the hash function which is not specialised for null-terminated (C-)strings

- the correct value is only returned if the pointers match (not guaranteed)

# Security

---

## Passwords

- Never re-use a password
  - ▶ Use password generators for one-use passwords
  - ▶ Store passwords in KDE KWallet, Apple Keychain, Gnome Keyring etc.
  - ▶ Don't forget to set a good master password

## Pub terminal: New session!

- Whenever you use a public terminal: Start a new session
  - ▶ Simple (`script`) to capture everything you type...