DON'T PANIC

David Bjergaard

January 23, 2017

Contents

A Hitchhiker's guide to High Energy Physics

Introduction

Welcome to High Energy Physics. Like the Galaxy, HEP is a wonderful and mysterious place filled with amazing things. The intent of this guide is to provide a jump-start to HEP research. Its purpose is to be entertaining and uninformative. The information in this guide was hard won, The Editor does not intend it to be different for our dear reader. This guide is not intended to make the reader's life easier. It is not intended to teach anything. It is intended to provide the dear reader with a glossary of concepts for further research.

If the reader finds this guide entertaining, but uninformative, then please send a bottle of Old Janx Spirit to The Editor. The Guide is not intended to be read linearly. Jump to the section in need, and if that section contains links to other sections, then please see above regarding edits. Prepare to be frustrated.

Navigating the guide

A note on key notation: Keys sequences follow the "emacs" style for notating keys. Therefore "`C-c`" means hold down the control key, while pressing the `c` key. "`M-a`" would mean hold down the "Meta" key and press the `a` key. On most keyboards the Meta key is the key labeled "Alt". There is also the "Super" key, on most PC keyboards this key has the "Windows icon" on it. The function keys 1-12 are notated as `F1` etc. The escape key is usually notated `ESC`, and the enter key is notated `RET` for Return. A sequence is denoted by a series of keys and spaces or dashes. A dash means hold the two

keys at the same time, a space means release the previous keys and continue with the next instruction. Some examples:

- "C-c C-a" Control-C release Control-A
- "C-c a" Control-c release A
- "C-M-f" Press Control, then Meta, then F without releasing

The online version of the guide includes a Table of Contents; simply mouse over it and a full list of topics will pop up. You can click any topic to jump to that section. If you read this in org-mode, the file will open folded. It will look like:

```
#+TITLE: DON'T PANIC
#+AUTHOR: David Bjergaard
#+EMAIL: david.b@duke.edu
#+OPTIONS:   H:5 num:nil toc:t \n:nil @:t ::t |:t ^:t -:t f:t *:t <:t
#+OPTIONS:   TeX:t LaTeX:t skip:nil d:nil todo:t pri:nil tags:not-in-toc
#+LaTeX_CLASS: article

 * A Hitchhiker's guide to High Energy Physics...
```

Place your cursor at the beginning of `* A Hitchhiker's...` and hit `TAB`, this will expand the topics allowing you to see and over-view of the document. Move to whichever topic you're interested in and hit `TAB` again to expand that section.

If you are reading this in Vim without Vim-OrgMode, then you will have no folding and the whole document will be expanded. Jump to various headlines by searching for "**" (two levels deep), "***" (three levels deep), "****" (four levels deep), etc.

If you are reading this as a PDF, there is no Table of Contents. Just scroll to the section you are interested in.

NOTE If you are reading this on GitHub, a link to another section will probably be broken due to a bug in how GitHub parses org-flavored markdown.

Disclaimer

All bottles of donated Old Janx Spirit are redirected to our lawyers, who are out enjoying them now. They insisted that we include this disclaimer:

> The Hitchhiker's Guide to High Energy Physics is a set of documents and software herein referred to as "The Guide"
>
> The Guide is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3, or (at your option) any later version.
>
> The Guide is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this software. If not, see `http://www.gnu.org/licenses/`.

Also, note that the content of this guide is satirical in nature, and is intended to be sarcastic. If you have a weak heart or are easily offended, it may be better to seek out other sources of information.

Obtaining a copy and supporting material

All of the sources for the guide are hosted on GitHub.

Here are some quick-links:

- The Guide the online version of the guide
- PDF of The Guide if you prefer that sort of thing, though it's of limited use in printed form.
- Org source of the online version
- GitHub Repo where source code is hosted
- Compiled Macros for Level 2 of ROOT enlightenment
- Compiled Programs for Level 3 of ROOT enlightenment
- Bug Reports/Feature Requests
- Pull Requests for submitting patches

To get a local copy:

```
git clone https://github.com/dbjergaard/hitchhikers-guide-to-hep.git
```

Then you'll have a copy of the org file, as well as the compiled macros and compiled programs.

For Windows Hitchhikers

Everyone should read For Linux Hitchhikers to understand what functionality they'll need (especially when working with or on remote machines)

While it is possible to practice HEP from the comfort of Bill Gates' brain child, it is not recommended by The Editor. (He doesn't run Windows anyway, daylight scares him.) If you insist on using Windows, the following is a list of useful software.

Software you will need

- For Linux Hitchhikers
- PuTTY (ssh client for Windows): Secure SHell is the standard way of accessing *nix machines remotely. PuTTY is the Windows client for this.
- `ROOT`: The industry standard for High Energy Physics analysis. Beware: this program uses an Infinite Improbability Drive to perform analysis.
- Xming an X11 server for Windows: This allows you to tunnel X11 applications (ROOT's histogram interface) to your Windows desktop, this way your data (and ROOT) can live on a remote machine, but you can still interact with them as if they were on your desktop. (You need a **fast** internet connection to do this). Xming comes in two flavors: the "web" version, which is locked behind a paywall for people who donate to the project, and a slightly less up-to-date public version available for free. Choose the public version. See Getting Started and Trouble shooting for tips on getting set up.
- Gow: A lightweight Cygwin alternative, this is probably for more adventurous hitchhikers only.
- Cygwin: Adds a substrate of the GNU system to Windows (in addition to an X11 server), you can use this to create a more Unix-like environment to work from.

- VirtualBox: Allows you to boot operating systems within operating systems (useful if you don't want to dual boot Ubuntu) see For Linux Hitchhikers after you've setup a working distro.

See here for a nice picture-book tutorial on installing Ubuntu through VirtualBox on Windows.

For Linux Hitchhikers

Software you will need

- Screen: This lets you pick up where you left off if your ssh connection drops, here is a good conceptual introduction. If you use `screen` on `lxplus`, you'll have to re-initialize your kerberos tokens after logging in with `kinit -5`, otherwise you won't have read access to your files.
- `ROOT`: The industry standard for High Energy Physics analysis. Beware: this program uses an Infinite Improbability Drive to perform analysis.
- BASH: The command shell of choice for ATLAS Physicists. You may think you could use ZSH, but it's better just to stick with what everyone else uses. CMS Physicists prefer TCSH for some weird reason.
- Editor: Choose you're religion wisely, it will eventually permeate your being and change the way you approach life in general.

The Terminal

You will, regardless of which operating system you use, be typing commands into a terminal. It's inevitable, powerful, and intimidating to new users. HEP hitchhikers should feel at home. Proficiency with the command line is essential to being a functioning HEP researcher.

The terminal is like the Galaxy Hitchhiker's utility towel. Every hitchhiker needs a terminal, and each hitchhiker customizes his or her towel to their needs.

If you've never touched a terminal before, and don't know what the command line is, there are two options:

1. The great pedagogical introduction (12 page PDF) by Carl Mason
2. "An Introduction to Unix" (a comprehensive, modern take on the unix ecosystem) by Oliver Elliott

You should read both, Carl Mason's tutorial should be read "cover-to-cover", where as Oliver's is written very much in the spirit of this guide, so bookmark it and refer to it after you've read Carl Mason's tutorial.

Line Editing Most modern operating systems' default shell is bash. Be aware that bash's line editor is set up to respect emacs keybindings, this means "C-a" is beginning of line "C-e" the end, etc. You can change to vi bindings by typing:

```
set -o vi
```

If you forget which mode your in, check it by typing:

```
set -o
```

If you want these changes to be permanent, add them to your `.bashrc`. If these commands give you an error, type:

```
echo $SHELL
```

And see what it says (`/bin/bash` if it's bash, may be `/bin/zsh` or `/bin/tcsh`). If it is not bash, then you need to google information for the line editor of whichever shell you are using.

Managing Jobs Sometimes it will be convenient to spawn a process and continue working in the current shell. Usually this is accomplished by redirecting the stdout and stderr to a file:

```
myLongRunningCommand foo bar baz 42 &> theProcess.log &
```

When you launch the command, you'll see something like:

```
[1] 19509
```

The number `19509` is the PID of the process. If you have multiple jobs going they can be summarized by typing `jobs`

```
[1]   Running                 myLongRunningCommand foo bar baz 42 &> theProcess.log &
[2]-  Running                 myLongRunningCommand foo bar baz 41 &> theProcess.log &
[3]+  Running                 myLongRunningCommand foo bar baz 40 &> theProcess.log &
```

Occasionally you'll realize that you don't want the jobs to run anymore, so to kill them:

```
kill %2
```

where `%1` is the job number you are referencing. You'll see something like:

```
[2]-  Terminated              myLongRunningCommand foo bar baz 41 &> theProcess.log &
```

Configuring SSH

Many of these tips are lifted from here. Put this in your `~/.ssh/config` file:

```
ControlMaster auto
ControlPath /tmp/ssh_mux_%h_%p_%r
ControlPersist yes
ServerAliveInterval 30
ServerAliveCountMax 1
```

It is possible to setup ssh shorthand to route you to remote machines. The syntax (in `~/.ssh/config`) is:

```
Host shortname
  #expands to shortname.remote.location.edu
  HostName %h.remote.location.edu
  User username
  ForwardX11 yes #this is equivalient to ssh -Y
  IdentityFile ~/.ssh/id_rsa #path to your pubkey
```

SSH Keys Follow this guide, stop at step 3. Now, when you need to start using a new machine:

```
ssh-copy-id user@remote.machine.name
```

Then enter your password. Now, when you type `ssh user@remote.machine.name` you will authenticate yourself with your newly minted RSA key, and you won't have to enter your password. The downside is that you'll have to enter your key's passphrase to unlock it. See below for a way to unlock it once per session.

NOTE While it is cryptographically more secure to authenticate yourself with ssh keys, if your machine is compromised (ie stolen or hacked) your ssh keys can provide the attacker with easier access to all the machines you had access to. This means you should:

1. Use a strong pass **phrase**, not password. You need to maximize the number of bits of entropy in your key in order to make it difficult to crack should the keys fall into enemy hands.

2. Inform the admins of any machines you had access to if your machine is compromised

3. Encrypt your ssh keys (and other sensitive information) in a private directory that only you can access

4. **NEVER EVER** store your ssh keys on a third party site (like Dropbox or similar services)

SSH Agent If you have ssh-agent running (through the `gnome-keyring` service on Ubuntu, or directly in your .xinitrc through `ssh-agent blah`) you can type `ssh-add` when you log in and it will add your ssh key to the keyring, then you can ssh to any machine that you have copied your key to without entering the password!

NOTE Once you've added your key to the ssh-agent, anyone can sit down at your keyboard and log into a remote machine as you! This means if you step away from your computer (even for a moment) you should lock the screen or log out.

Version Control Systems

The two major version control systems in HEP are Git and Subversion (svn). These are tools and utilities to allow collaboration on large pieces of software.

They also provide programmers with a convenient "paper trail" through the course of developing a piece of software. It allows them to revert the source code they are working on to any state that they've previously checked in.

Subversion is a successor of CVS, everything is stored on a remote site, and your source code directory contains metadata about the source code with reference to the remote site.

Some subversion tutorials:

- Source Control in 10 Minutes
- Command Line Subversion Tutorial (part 1)
- Version Control with Subversion (A comprehensive free book about Subversion)

Git is a software that was written by Linus Torvalds, the hacker behind Linux. It was written to manage the Linux kernel, a massive piece of software. Git's model for managing source code is slightly different. In Git, you maintain the entire repository in your local copy. This makes committing, managing, and branching very fast. It also means you can work with all of the advantages of a version control system without internet access. Simultaneously there is a copy of the repository on a remote server. Git handles syncing these two repositories when instructed. This can lead to confusion

if you've used other versioning systems, but shouldn't be a problem if you have no expectations.

Some good Git tutorials:

- type "man gittutorial" in the command line
- Pro Git (an online book, modular and comprehensive in scope)
- Git Immersion
- Git Concepts Simplified (slide show, click to advance)

Intermediate or advanced topics:

- Undoing, fixing, or removing commits in Git
- Simple Git workflow is simple
- Git tips from the trenches

*rc Files

`*rc` files are special files that are executed every time a program starts. They almost exclusively live in the user's home directory, and can be shadowed by the system. Sometimes they have a special syntax for setting options, sometimes they are written in a scripting language. The most relevant rc files for a new hitchhiker are:

.rootrc the file that sets root options

.bashrc this is executed every time you open a terminal in bash

.tcshrc as above but for tcsh (hopefully you aren't using this!)

.zshrc as above for the zsh shell

.vimrc configuration options for the venerable vim editor

.screenrc options for gnu screen

.emacs written in emacs lisp, is executed on startup, breaks the rc naming scheme. Advanced emacs users have multi-thousand line rc files

There are other files, if you want to know about them you can do:

```
ls .*rc
```

And google the ones that look interesting. Alternatively you can look at the system defaults:

```
ls /etc/*rc
```

Sometimes its useful to copy the system file to your home directory and then edit it there in order to add your customizations. Some programs document their options that way.

For Mac OS X Hitchhikers

Everyone should read For Linux Hitchhikers to understand what functionality they'll need (especially when working with or on remote machines). As a Mac user, you should also read "It just works... or does it? The dark side of Macs in HEP" by Andy Buckley. It explains in detail issues with software development on a Mac. It is an opinion piece, so don't expect it to be balanced. Also, consider asking your supervisor for an account on a Linux box and **never look back.**

Software you will need

- XQuartz: Like XMing for Windows, XQuartz runs a local X11 server for tunneling X11 applications over SSH, unlike Windows, you don't need a separate SSH program, ssh is built in.
- Terminal.app: This is Mac OS's default terminal emulator. It comes with Mac OS, so you shouldn't need to install it. You should be aware of it though.
- `ROOT`: The industry standard for High Energy Physics analysis. Beware: this program uses an Infinite Improbability Drive to perform analysis.
- Aquamacs: A port of Emacs that uses Aqua as a standard OS X application. This integrates Emacs with the Mac OS UI. In the long history of corporate acquisitions a lot of Emacs hackers (from NeXTSTEP) ended up at apple, you will find that Mac OS integrates the Emacs experience much more fundamentally than any other OS in existance. (This doesn't mean you need to use Emacs if you use Mac OS, just that your muscle memory will thank you subconsciously.)
- MacPorts: A system for compiling and installing open source software on the Mac

- Home Brew: A package manager for Mac OS, allowing you to install various utilities that don't necessarily come pre-installed with Mac OS.

Editors

Like the major world religions, there are also major editors. In the *nix ecosystem there are two main editors: Emacs and Vim. There are others, but they are many, and beyond the scope of this guide.

The most important thing to do after choosing an editor is to work through its corresponding tutorial (more pragmatic advice here). An oft heard recommendation is that "Emacs is easier to learn than vi(m)". A more accurate statement may be that it is easier to make things happen in Emacs than Vim, but the two editors are in some sense the yin and yang of text. True enlightenment in either of these editors takes roughly the same amount of time after completing the corresponding tutorial.

Finding an editor Guru

After you have finished the tutorial for your editor of choice, then it's time to find a guru. Guru's are best located by asking around. If you are talking with someone and notice they use your editor, don't be afraid to ask them how they did something. Most of the time the Guru will be flattered and may even volunteer to help you with any other editor related questions.

Editor Guru etiquette While it is generally OK to ask your Guru any editor related question, it is best to keep questions restricted to the editor in question. Flame wars have been fought for decades over which is the "one true editor."

In order to prevent a *faux pas*, it is best to make sure you know which editor your guru uses. This is especially true in the case of a vi(m) or Emacs guru.

Another thing to be careful of is repeatedly asking basic questions. Again, some gurus will tolerate this at the beginning, but after a point the guru expects you to master the basics (on your own). The most valuable knowledge your guru can impart is not written in the tutorial that came with the editor.

Keeping your Guru happy Guru's subsist mainly on a liquid diet of caffeinated beverages during the day and beer (occasionally wine) at night. It is important that your Guru remain well lubricated. It is generally considered

a good gesture to offer your Guru his/her beverage of choice if you've found him/her to be especially helpful on your path to enlightenment.

Emacs

The end goal of any student of the Church of Emacs is to obtain proficiency reprogramming the editor to solve the task at hand. This is ultimately stems from the philosophy of lisp (this gift was given to us by St. IGNUcious an AI hacker from MIT where Emacs was born). In lisp, the flexibility of the language allows it to be re-written to solve the problem as clearly as possible. In Emacs, an enlightened user will write a substrate of elisp (Emacs' dialect of lisp) in order to solve the editing problem at hand.

While customizing and writing your .emacs (the initialization file loaded by Emacs in your home directory) is a spiritual journey, there are those who have done their best to illuminate the path. A brief guide to customization philosophies here.

The Editor finds the following packages essential:

- tramp: If your reading this in Emacs, you can follow the link with "C-c C-o". It is **the** most important aspect of Emacs for HEP users. It allows you to "visit" files on remote machines from the Emacs running on your desktop. It does this through ssh. To visit a remote file, type "C-x C-f" and then type '/ssh:user@remote.host:~/remote/path', note that tab completion works remotely just the same as visiting a file locally! Tramp is also aware of ssh aliases in `~/.ssh/config`, see Configuring SSH.

- Calc - ""Calc" is an advanced desk calculator and mathematical tool written by Dave Gillespie that runs as part of the GNU Emacs environment." It handles barns and electron volts out of the box!

- filladapt: a mode for more intelligently filling text in paragraphs

- flyspell: a spell checker that highlights mispelled words (will check in comments if in a programming mode)

- rect-mark: Adds facilities for marking yanking and otherwise editing columnar formatted text.

- dired (another info link): a directory editor for manipulating files in the Emacs way

- solarized-theme: A theme by Ethan Schoonover, comes in dark and light variants that actually complement each other well, another good one is zenburn
- ibuffer: changes the buffer interface and allows you to group buffers based on various buffer attributes
- paredit: Enhances Emacs's awareness of parenthetic structure
- smartparens: Electrically pairs and deletes delimeters when appropriate (never miss a closing brace again!)
- auto-complete: When setup properly, tab completes anything at any point depending on past input or names in other buffers.
- auctex: LaTeX editing facilities (for when org-mode doesn't quite cut it)
- org-mode: This guide is written in org-mode. Org-mode can manage todo lists, write websites, serve as a lab notebook, execute code for literate programming and many other things. More relevant for physicists is the org-mode cookbook. People switch to Emacs just to get org-mode!

Init files of famous Emacs hackers are (in no order of awesomeness) Magnar Sveen, Technomancy, John Wiegley. There are also software packages that intend to comprehensively change the Emacs out of the box to a better user experience. The two most famous are Prelude and Emacs Live. An example (slightly annotated) init file can be found here.

Finally, there are some Emacs gurus who post blogs on the internet. Some particularly useful ones are Emacs Redux, Mastering Emacs, and Emacs Fu.

Various religious texts granting Emacs users various powers (such as reading email, chatting, tweeting, playing games, listening to music) can be found at the Emacs Wiki.

Vim

If Emacs is like Catholicism, then Vim is like Buddhism. Vim is the modern incarnation of vi, a modal text editor that descended from ed. The modal way of editing is by expressing in a few keystrokes how the text should be manipulated. This is in contrast to Emacs, where text is manipulated directly. This fundamental difference is the source of much confusion for new users, and is also why many people recommend Emacs as "being easier to

learn." This should not deter new users from learning vi(m), as its editing facilities are substantial.

A functional `.vimrc` looks like:

```
syntax on
set cursorline
set hlsearch
set ic
set incsearch
set ruler
set shiftwidth=4
set tabstop=4
set wrap
```

To learn Vim, type `vimtutor` at the command lime and follow the instructions. Take your time, and repeat the tutorial once or twice over a few days. In the mean time editors such as `gedit` or `nano` offer a more traditional experience. As your Vim skills improve, you will feel more comfortable with Vim and can stop using the less powerful editors.

Some useful links include:

- Vim Videos Tutorial videos by Derek Wyatt, the novice videos are must see if you are new to vi(m)
- Vim Genius a drill website for learning Vim commands
- New user Vim Tutorial
- Vim Koans tidbits of wisdom to ponder
- A collection of extensions and plugins for Vim
- YouCompleteMe A Vim autocompletion engine for editing.

Others

Followers of the Unix way realize that there are situations where a using a set of shell commands piped together may fit the task at hand more efficiently than either of the other two editors. Tools you should be familiar with are:

- sed and one-liners
- awk and one-liners

- perl (and its poetry)
- grep
- Heretics exist which exhort the use of pico or even nano.

Always keep in mind

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. – Jaimie Zawinski

Software Design

Well designed software is a true marvel, in the same way architecture is a marvel. You are a stone mason, and you are building a cathedral. Repeat that last sentence every time you want to take a shortcut when coding. A cathedral can't stand on a flimsy foundation.

In order to help you on your way, you should read the following:

- Architecture of Open Source Applications
- How to Write Unmaintainable Code (**warning**, many physicists take this guide literally)

Good software design is **very** hard, but when you have the pleasure of using well designed software, it is a true joy. Some examples of good HEP software:

- Rivet: Robust Independent Validation of Experiment and Theory
- Fastjet: Software package for jet finding

A brief introduction to C++

C++ is the industry standard programming language for analysis in HEP. Even if you are fortunate enough to do most of your work in Python, you will eventually be calling C++ code, and should understand some core concepts in order to debug problems should they arise.

Things to keep in mind:

- This portion of the guide covers C++ at a high level. Very little specific syntax will be covered. When you have a C++ question, google is your friend.

- When writing in any language, prefer that languages idioms. Don't write python in C++, C in C++ or C++ in python.
- C++ is a vast language, however being familiar with its roots, C, is invaluable.
- If faced with a decision between learning C++ vs Python, prefer C++. C++'s syntax is more rigid and requires more overhead. Once you know C++, python is much easier to pick up.
- There's always an exception to the rule, just make sure it's the right exception!

C++ is an imperative, object oriented language. It started out as a "C with classes" but has since bolted on significant language features different from C. Proficiency with C++ should be aimed towards comfortable use of the template meta-programming features of the language, although it is entirely possible to spend an entire career writing C++ without exercising this feature (just read the ROOT source code).

Pointers

Required Reading: The Descent to C

As C++ has evolved from C, it retains parts of C's low level nature. Part of this is the need to be explicit about managing memory manually. This is in stark contrast to languages such as Java or Python where memory management is handled for the programmer.

A consequence of this is the ability to address specific cells of memory (the smallest accessible unit, typically a byte). An object (`int`, `double`, `float`, `char`, `string`, etc) may span several memory cells. A pointer is the computer's representation of a memory cell's location in memory, ie a memory address. Ultimately the programmer is interested in the data contained in the set of memory cells "pointed to" by the pointer. The act of retrieving this data is called "*dereferencing* a pointer".

As in physics, facility with manipulating pointers is best gained through experience, however many analogies have been developed to ease confusion. One analogy is street addresses, A street address is a sequence of numbers (the pointer) which instructs someone, a mailman say, (the computer), how to find a specific location. Once at that location, it is possible to manipulate objects located at that address (deliver mail if your the mailman, break the mailbox if your a bored teenager, knock on the door if you are a vacuum salesman etc).

Now some syntax:

```
Foo* bar = new Foo("Baz",42,"What is the question?");
std::cout << "object bar lives at memory address:"<<bar<<std::endl;
std::cout << "bar calculated a question to the answer to \"The Ultimate Question\" as
std::cout <<"Another way to get the answer is: "<<(*bar).TheAnswer()<<std::endl;
```

Lots of interesting things have been introduced here. Let's look at a possible output of this program:

```
object bar lives at memory address: 0xd29ad0
bar calculated a question to the answer to "The Ultimate Question" as "What is 6x9?"
Another way to get the answer is: "What is 6x9?"
```

What happened? Let's look at the first line

```
Foo* bar = new Foo("Baz",42,"What is the question?");
```

`Foo*` is a pointer of type `Foo`. It's an address to a chunk of memory that contains an instance of `Foo`.

> Question: Why does the compiler need to know that it's a Foo type object at that address?
>
> Answer: Foo might fall across several memory cells, in which case the compiler must know how many memory cells to move if you ask for the bar+1 spot. In fact, in C there is a concept called the void*, a type-less pointer that is an address to anything. It is the programmer's responsibility to cast the void* to the correct type.

OK, so we have a pointer to an object of type `Foo` called `bar`.

> Question: What happens on the right hand side of the assignment operator (=)?
>
> Answer: C++ reserves the keyword "new" for memory allocation. The "new" keyword takes a class constructor on the right hand side, and returns a memory address on the left hand side. This address gets stored in the variable `bar`.

Operationally, the "new" keyword allocates a chunk of memory to hold the object on the right hand side, and returns a pointer to the beginning of the chunk.

What happens when we want to access the memory that the pointer points to? There is another operation called "dereferencing" which goes to the address pointed to and returns the object contained at that point in memory. Consider the following snippet:

```
double* foo = new double(3.14159);
double pi = *foo;
std::cout <<"Pi is: "<<pi<<std::endl;
```

Here, a chunk of memory has the value 3.14159 written to it, then that value is retrieved and stored in another location of memory called `pi`. That data is the written out the terminal by `std::cout` and `std::endl`.

Now we can understand this line:

```
std::cout <<"Another way to get the answer is: "<<(*bar).TheAnswer()<<std::endl;
```

It means, retrieve the object pointed to by `bar`, and call the method `"TheAnswer()"` on it. Programmers abhor syntax that can easily get them into trouble, so the language designers (of c) added a shorthand for this kind of operation (the `->` operator):

```
Foo* bar=new Foo();
if(bar->Value()==(*bar).Value()){
  std::cout<<"They're the same!"<<std::endl;
}
```

Quiz: What will the output of this snippet be?

Why are pointer's useful at all? Clever hitchhikers will notice that this appears to be a bunch of bureaucratic mucking about with pointless details, most of the time it is. Since most of HEP deals with pointless details bureaucratically, a lot of HEP code uses pointers.

To understand the real purpose of pointers, we must examine dynamic allocation. Consider the following code:

```
double* foo(){
  double* bar = new double(0.0);

  {
    double baz=42;
    *bar=baz;//dereference bar, and store the value of baz
  }
```

```
  //baz is out of scope
  return bar;
}
int main(void){
  double* foobar=foo();
  std::cout <<"The Answer to the Ultimate question is :"<<*foobar<<std::endl;
  return 0;
}
```

Let's execute the code in our mind:

1. execute `main(void)`
2. a `double*` named `foobar` is allocated.
3. `foo()` is executed
   (a) a `double*` named `bar` is allocated
   (b) new initializes a `double` with value 0.0
   (c) new assigns the address containing that `double` to `bar`
   (d) enter the braces, intialize a `double` named `baz` with value 42
   (e) dereference `bar` and copy the value of `baz` into it
   (f) exit the braces and free the memory where `baz` was
   (g) return the address containing the value of `bar`;
4. Assign the value returned by `foo()` to `foobar`
5. stream the string "The Answer..." to stdout
6. dereference `*foobar` to obtain the value stored at `bar`, 42, stream that to stdout
7. add a newline to the output and flush the result to the terminal with `std::endl`
8. return 0
9. exit the program

References A similar concept present in C++ (but not c) are references. They can be thought of as aliases (the way Dave is an alias of David). Their syntax is:

```
int foo=42;
int& theAnswer=foo;
foo=0;
std::cout<<theAnswer<<std::endl;
```

Here foo is initialized to the value 42, then a reference named theAnswer is declared and assigned to foo. All this does is make a new name for the same object. What does the program output?

The answer is 0. References seem pointless (pun intended) until they're used in function definitions:

```
void bar(int& foo){
  //complicated calculation for foo
  foo++;
}
int main(void){
  int baz=41;
  bar(baz);
  std::cout<<baz<<std::endl;
}
```

Question: What is the output of this program?

Answer: 42

Methods

Methods, or functions are defined as:

```
return_type function_name(arg1_type arg1, arg2_type arg2, ...){
  //statements that define function_name
}
```

It is possible to "forward declare" functions, these are "promises to the compiler" that you have a function with a particular signature:

```
double foo(double,double);
// important other stuff
double foo(double theta, double phi){
  return sin(theta)*sin(theta) + cos(phi)*cos(phi);
}
```

Notice that the compiler doesn't need to know the names of the arguments in the forward declaration.

Before we move onto the next topic, a note on methods. Most of the time during development, you only have a few helper functions. This is fine! Just write your helper functions in a header file, and include them. Write the main function and move on with your life. There are many examples in HEP, where methods have been pigeon-holed into classes. The result is a cumbersome interface for the user (YOU!) or more importantly your supervisor. With that in mind, lets move on to classes.

Classes

STOP! Read the last paragraph of the previous section.

Now, ask yourself: Do I really need a class? No. Ok, great!

Yes. Are you sure? Maybe your needs are better served with a few functions and a well defined interface.

Do you have complicated data structures that need to be operated on by many methods? No? Maybe your needs are better served with a few functions and a well defined interface.

Yes? Maybe you should rethink your design.

You've rethought it and realized that you have to use a class because the person before you did, now there isn't a clean way to do it any other way. OK, classes.

> Editor's Note: Our dear reader may have been studying this guide and thought to themselves: "Gee wiz, there is a lot of great advice here, I should follow this exactly or dragons might eat my analysis program!" While this may be a possibility, the previous advice on classes may be a little, well, misguided.
>
> More sound advice may sound like this: When beginning development, make your first (and maybe second) drafts of the program without classes. This will force you to recognize which parts of your program fit together (data+methods) and which are separate. Then, revise your program to include classes with the appropriate encapsulation of data and methods. This will ensure that you write a clean interface, and that you don't end up with 20 methods that all take the same four arguments.

The basic syntax is:

```
class A{
public:
  A():a(0),b(0){};
  A(int _a):a(_a),b(0){};
  ~A(){};
  void SetA(int new_a){a=new_a;};
  void GetA(){return a;};
  void SetB(int new_b){b=new_b;};
  void GetB(){return b;};
private:
  int a;
protected:
  int b;
};
```

This is a trivial example, and it breaks many rules about naming conventions and clarity. It is not a good example. It should not be used as a good example to win arguments about concise code. In fact, you probably shouldn't have read it.

Important features of the code: Anything after the public keyword is accessible to the outside world, ie:

```
A myObj;
myObj.SetA(10);
myObj.GetA();
myObj.SetB(42);
myObj.GetB();
```

Is all valid code, anything that is written after private, is just that. You cannot access it outside of the class:

```
myObj.a; //Compiler error
myObj.b; //protected is a special form of private
```

The protected keyword is for class inheritance. It says that these variables and methods are private for users of the class, but if another class inherits from this one, they inherit these symbols.

Normally in class inheritance, you only inherit the public members of the class. The private members are not inherited. Protected offers a way to encapsulate data, but also share data among inheritance diagrams.

In case you haven't picked up on it, classes are one of the hairier aspects of C++. It's better if you refer to some other resource for a tutorial on classes, as their subtleties are beyond the scope of The Guide.

An even briefer introduction to Python

Python is a wonderful language. It is expressive and allows rapid prototyping with a shell type environment. Try learning it the hard way. Another approach is to google what you're trying to do, and make it run on a small test case.

When writing code, it is best to be idiomatic. This is especially true in python. Python's driving philosophy is "one right way" but since python is being developed by multiple hackers, there are "many right ways."

- Here's an older tutorial on idiomatic python.
- A slideshow on idiomatic python
- A non-programmer's tutorial on python 3 (also see the linked version for 2.7 as HEP is still using 2.7 or earlier in many cases)
- Finally, the official recommendations for best practices.

PyROOT: ROOT bindings in python

Mostly Harmless.

ROOT

For better or worse, for richer or poorer (always poorer), HEP Physicists are married to (read: stuck with) ROOT. It's the Unix of HEP. There is method in the madness, though it is not clear what the method is (just yet). The Editor is fairly certain that ROOT uses Bistromathics for many of its statistical operations.

ROOT is the industry standard tool for analyzing and manipulating gobs of data. Other statistical analysis tools crash and burn on the datasets that ROOT eats for breakfast.

Recently, ROOT has begun the transition from the 5 series to the 6 series. This transition has resulted in huge changes to the front and back-end of ROOT. Below there will be some comparisons to both versions. When something applies to the 5 series, it will be annotated ROOTv5. It should be assumed that ROOT or ROOTv6 refers to the newest 6 series. Without further ado, let's set it up and get to work!

Installing and setting up

On Windows These links are for 5.34.34. You are encouraged visit the downloading page of root.cern.ch to check for newer versions. Always prefer the "pro" version. It appears that in ROOT 6, the developers have stopped releasing windows builds. This means that to use ROOT 6 and above you'll have to either compile from source, or use a virtual machine.

- VC++11 (2012) EXE for 5.34.34 (install-able version, allows you to remove it using the Control Panel)

On MacOS If you are using `homebrew` you can simply do:

```
brew tap homebrew/science
brew install --with-cocoa root
```

(source)

On Ubuntu This will install all ROOT packages from the Ubuntu repositories:

```
sudo apt-get install root-system
```

Building From source

These instructions are for *nix based systems (ie it was written for Ubuntu, but MacOS shouldn't be much different and Windows is out of the question).

When choosing a version of ROOT, always pick the 'pro' (pro for production) version. It's the latest, stable, version recommended by the ROOT Devs.

Nota Bene If you're doing this on Mac OS, you'll need to use `brew install blah` instead of `apt-get install blah`, and the package names will probably be different.

If you haven't yet, read the `section of the guide` relevant to your OS.

For our install of ROOT, we'll be compiling and running it locally. This has a few advantages:

- "Uninstalling" is easy, either unset the environment variables pointing to ROOT, or completely delete the folder that root lives in (in this example `~/root`)
- Having multiple versions side-by-side is possible, you could have:

  - `~/root-clang` a version of root compiled with clang
  - `~/root-5.34` a stable version of ROOT
  - `~/root-5.99` the beta version of ROOT 6
  - To use any of them you would just have to source `~/root-ver/bin/thisroot.sh`

- You don't need root (administrative) access on the machine (as long as the pre-req's are installed). This is generally nice since it decouples ROOT from the hosting OS.

Getting the Pre-Requisites Now, all of the following information is documented at `root.cern.ch`, but it is even less organized than this guide. The following is a straight-shot from no source code to a fully working ROOT binary on a clean install of Ubuntu (currently 13.10, but the build process has been stable for the last ~3 years)

End to end, this takes ~40min on a machine circa 2011, so budget some coffee time.

```
sudo apt-get install git dpkg-dev make g++ gcc binutils libx11-dev libxpm-dev \
libxft-dev libxext-dev gfortran libssl-dev libpcre3-dev \
xlibmesa-glu-dev libglew1.5-dev libftgl-dev \
libmysqlclient-dev libfftw3-dev cfitsio-dev \
graphviz-dev libavahi-compat-libdnssd-dev \
libldap2-dev python-dev libxml2-dev libkrb5-dev \
libgsl0-dev libqt4-dev
```

You may have some of these packages already if you've installed `build-essential` or `git` before. In either case, `apt` is smart enough to see that and not re-install them.

The above list of packages are for a full-blown, all-features-enabled version of ROOT. If you want a stripped down version, you'll have to get the pre-reqs from `http://root.cern.ch`.

Alternatively you can type:

```
sudo apt-get build-dep root-system
```

And let `apt` install and manage any dependencies ROOT needs. This is overkill, even for a "bells and whistles" build of ROOT.

May the Source be with you Let's get a copy of the source:

```
git clone http://root.cern.ch/git/root.git
```

For future reference, if you want to update:

```
git pull
git tag -l
git checkout -b tag-name tag-name
```

`git tag -l` lists all the available tags, choose the one you want and substitute it for `tag-name`

For now, lets checkout the latest pro branch:

```
git checkout -b v6-04-14 v6-04-14
```

This will checkout the branch `v6-04-14` to a local branch `v6-04-14` and switch you to it. If you're new to "version control systems", or "source control management" then it's useful to do a tutorial to learn the basics. In HEP, the major system used is called SVN, in open source, Git has become the de facto standard almost overnight.

Building With our code checked out and ready, we need to configure it to match the computer we're compiling on. To do this:

```
./configure --all
```

To see all options run `./configure --help`, this command suggests piping the output to `more`, but most "modern" terminal emulators have a scroll-back buffer large enough that you can just scroll up and read the output. The `--all` option instructs configure to enable support for as many packages as your system supports. If you require a specific feature (say roofit) you would type:

```
./configure [other options] --enable-roofit
```

If you're interested in building ROOT with xrootd (network protocol which allows opening root files over a network connection) see Advanced Build Options.

After configuring you should see:

```
Enabled support for asimage, astiff, builtin_afterimage, builtin_llvm, explicitlink, f

To build ROOT type:

   make
```

Now type `make`:

```
make -j 4
```

The `-j` option tells make how many jobs it can run simultaneously. Without `-j`, only one job runs. A good rule of thumb is to choose the number of cores you have on your computer. If you are compiling on remote computer, it is probably shared by others, in which it is good etiquette to run your jobs single threaded. On your laptop, you should choose **one less** than the number of cores you have (so you don't notice a slow-down while its building in the background).

See `man make` and look under the option "-j [jobs]" for more detailed information about this switch.

This will take (depending on your hardware) between 20-45min, so now is a good time for a cup of tea, or coffee with your `editor guru`.

When it's finished, it will print out:

```
[lots of boring crap]

   ====================================================================
   ===                 ROOT BUILD SUCCESSFUL.                       ===
   === Run 'source bin/thisroot.[c]sh' before starting ROOT ===
   ====================================================================
```

If you do not get this message, but the build just ends with `[lots of boring crap]` find a senior grad student and have them look at the `[lots of boring crap]` (it won't be boring to them). They will be able to instruct you on what went wrong.

If you don't have a senior grad student handy, try googling some of the output and seeing if you can get anywhere. There is also the ROOT Talk Forums.

Using your new superpower Now, when you want to use `root`, you can run the command:

```
source ~/root/bin/thisroot.sh
```

If you are using tcsh (you shouldn't be) you would need to run:

```
source ~/root/bin/thisroot.csh
```

There are differing opinions about whether or not you should put something like this in your bashrc. One school of thought (especially applicable when you bounce between different versions) is that you should keep your environment as clean as possible and only setup what you need. In that case adding:

```
alias setupROOT='source ${HOME}/root/bin/thisroot.sh'
```

To your `~/.bashrc` file is enough.

Then, whenever you need root, you have to run `setupROOT`, before you can run `root`.

Another school of thought is that, you should always have some copy of root available if possible. In that case the following will always setup root when bash runs if the setup file exists:

```
[ -f ~/root/bin/thisroot.sh ] && source ~/root/bin/thisroot.sh
```

Now you can start root by typing `root` at the command line. You should see:

```
   ------------------------------------------------------------------
  | Welcome to ROOT 6.07/03                        http://root.cern.ch |
  |                                  (c) 1995-2016, The ROOT Team     |
  | Built for linuxx8664gcc                                           |
  | From heads/master@v6-07-02-292-g1c873d2, Feb 04 2016, 12:28:28    |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'        |
   ------------------------------------------------------------------

root [0]
```

In ROOTv5:

```
  ****************************************
  *                                      *
  *        W E L C O M E  to  R O O T     *
  *                                      *
  *   Version   5.34/15  11 February 2014  *
  *                                      *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch          *
  *                                      *
  ****************************************
```

```
ROOT 5.34/15 (v5-34-15@v5-34-15, Mar 21 2014, 14:04:01 on linuxx8664gcc)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
```

To start root without the splash screen type `root -l` in which case you see:

```
root [0]
```

Advanced Build Options

Full instructions here

If you want to use xrootd, you'll also need cmake:

```
apt-get install cmake
```

Then, (from the directory where you checked out `root`):

```
./build/unix/installXrootd.sh
```

This will install xrootd to whatever directory you're currently in, so if you want it installed somewhere else, cd to that directory first, just make sure that you update the location in the next command.

Now when you configure root you should use:

```
./configure --all --with-xrootd=${HOME}/root/xrootd-4.2.1/
```

Here `${HOME}/root/xrootd-4.2.1/` is the path where xrootd was installed (based on the `installXrootd.sh` script above). If you blindly follow the instructions the above command will "just work".

Nota Bene The path following `--with-xrootd=` must be a fully qualified path (ie `/home/username/root/xrootd-4.2.1/`), the configure script doesn't understand that `~/` is a shorthand for `${HOME}`.

if `xrootd` support has been enabled you should see:

```
Enabled support for asimage, astiff, builtin_afterimage, builtin_llvm, explicitlink, f

To build ROOT type:

   make
```

Make sure you see `xrootd` in the list! If not xrootd won't be installed and won't function properly. You can now continue with the `regular instructions`. **Note**: If you want to be able to access files hosted at CERN, you'll have to have a kerberos client installed (either `krb5-user` or `heimdal-clients` on debian or ubuntu systems). Then before connecting, run `kinit -5 ${CERN_USER}@CERN.CH` to generate a kerberos certificate.

A Path to ROOT enlightenment

There are three methods of running code through ROOT to produce results. These methods are listed below, each more sophisticated than the last. They also include example code intended as a starting point for hacking.

As a reminder, see Obtaining a copy and supporting material for instructions on obtaining and running the example code.

Level 1: Macros The first, and simplest way to execute ROOT related code is the humble macro. A macro is a set of ROOT commands enclosed by braces. For example:

```
{
  TFile* file= TFile::Open("MeaningOfLife.root");
  TH1F* hist = (TH1F*)file->Get("Hist1");
  cout << hist->GetNbinsX() <<endl;
}
```

For ROOT 6: ROOTv6 macros are compiled with the LLVM backend "just in time" (JIT). This means that you have to type valid C++ in order to make things work correctly. This alleviates a lot of the issues that make Macros fragile, and so it is easier to transition from Level 1 to Level 2, and you can achieve more with a macro than before.

For ROOT 5: While not immediately obvious, ROOTv5 macros are not written in C or C++, but CINT. CINT covers "most of" ANSI C and ISO C++ 2003. There are some important differences:

- `foo.blah` and `foo->blah` are interchangeable
- a semicolon ';' at the end of a line is optional
- No need to "`#include`" headers

As you progress in writing more sophisticated C++, you will run into CINT's shortcomings as a C++ interpreter. It is recommended that you move to Level 2 or 3 before this happens.

While it is possible to write complicated CINT macros (files with multiple function definitions) it is not recommended. CINT has a habit of keeping up the appearance of doing one thing when in reality something entirely different is happening "behind the scenes".

CINT is best used for quick scripts to plot histograms already saved to a disk, or to inspect a few branches from a `TTree`. More sophisticated analyses are better served by Levels 2 and 3.

Level 2: Compiled Macros Compiled macros are full-blown C++ programs. Generally there is a "steering macro" that handles compiling and loading the required libraries. An example steering macro:

```
{
  //may need to load other libraries or files that depend on analysis.C
  gROOT->ProcessLine(".L analysis.C++");
  gROOT->ProcessLine("doAnalysis()");
}
```

The compiled macro itself looks more like a traditional C++ program:

```
#include <cstdlib>
#include "TFile.h"
#include "TH1F.h"

int doAnalysis(){

  return 42;
}
```

Since the ROOT binary already defines a "main" an error will occur if you redefine another function named "main", therefore we use the verb "doAnalysis".

The steering macro that compiles each source file can become arbitrarily complex. To some this may read "flexible", to others it may read "disorganized". If your analysis grows into a multi-file program, it's probably time to ascend to Level 3.

Level 3: Compiled Programs A compiled program is just that. Here ROOT takes the role of a rich set of libraries for composing a C++ based analysis.

An example program (and supporting Makefile) is included here.

Makefiles come with their own overhead, but the `make` system is very powerful. The make manual is very readable with many examples.

A note on Enlightenment Master Foo, of Rootless Root, gives the sage advice:

> "When you are hungry, eat; when you are thirsty, drink; when you are tired, sleep."

To spell it out (and to prevent the reader from enlightenment), it is wise to choose the use of ROOT which is most appropriate for a task at hand. The practicing HEP physicist is proficient with all three levels, and can pick and choose which approach is best for the task at hand.

PyROOT

PyROOT are a set of python bindings to ROOT. It works fairly well out of the box, but there are some things to keep in mind.

- Idiomatic python avoids "`from ROOT import *`", prefer "`from ROOT import blah`"
- the ROOT devs know you aren't going to be idiomatic, so instead they've implemented a lazy loading system (ROOT is huge, so "`from ROOT import *`" would take forever). This may be confusing if your a python expert and expect exploration commands like `dir()` to work with ROOT.
- If performance matters, try to stay in C++ land (ie call C++ functions from python) as much as possible
  - If performance really matters, write it in python and then port it to C++. This is fairly advanced, but not impossible. You'll have to generate CINT dictionaries for your source files.

If you're looking for a more "pythonic" (not my word) experience, maybe give rootpy a shot. See also An even briefer introduction to Python for resources to learn python itself.

Fitting Data with RooFit

RooFit is a shiny penny compared to the rest of the ROOT ecosystem. It has its own quirks and idioms, but the interfaces are fairly reasonable and

the manual is well written. The latest version of the manual and quickstart can always be found here:

- ROOT User's guide, Roofit Manual

Direct links are here, though they aren't guaranteed to be current:

- RooFit Manual (PDF) 2.91-33
- RooFit Quick Start Guide (PDF) 3.00

Styling Plots

A well designed graph is truly a work of art. The path from paltry graphics spit out by ROOT to something worthy of framing (and yes, truly amazing data visualizations are routinely framed) is long and fraught with naysayers who will insist that you are doing it wrong. Ignore them, and do what needs to be done to communicate your hard won data clearly and concisely. To get you started, give "Principles of Information Display for Visualization Practitioners" a read. It is an executive summary of Edward Tufte's works on visualizing information. If what you read resonates with you, then please read Tufte's books. They make for delightful coffee time distractions. After reading his books you will start to see his influence in particularly nice graphics. You will also see many sins committed by other practitioners in our field. Choose your role-models wisely.

Sage advice (passed down from The Editor's first mentor):

> When you make a plot, take the time to make it publication quality and reproducible.

This means two things:

1. Make it good enough to go into a paper
2. Prefer generating it with C++/Python over any other format (data inputs will frequently change at the last minute, and being able to "hit a button" and get the plot is very useful unless you have a room full of Mechanical Turks lying around)

This also means it's probably a good idea to keep `*.root` files containing your histograms for last minute style changes if you are writing a presentation.

Producing a publication quality plot can be challenging, however ROOT includes the concept of a "Style" which can be applied. These are global

rules for how plots should be printed. In previous versions of ROOT, the default style was notoriously bad. In The Editor's humble opinion, this was done to simultaneously encourage each physicist to set their own standard, and to immediately identify ROOT newbies from seasoned ROOT hackers.

Now, things are better, though the idea that "each physicist set their own standard" has stuck, and so there are many styles floating around.

Example Style An example style (probably from the CMS TDR, the details are lost to time):

```
{
  TStyle *tdrStyle = new TStyle("tdrStyle","Style for P-TDR");

  cout << "TDR Style initialized" << endl;

// For the canvas:
  tdrStyle->SetCanvasBorderMode(0);
  tdrStyle->SetCanvasColor(kWhite);
  tdrStyle->SetCanvasDefH(600); //Height of canvas
  tdrStyle->SetCanvasDefW(600); //Width of canvas
  tdrStyle->SetCanvasDefX(0);   //Position on screen
  tdrStyle->SetCanvasDefY(0);

// For the Pad:
  tdrStyle->SetPadBorderMode(0);
  // tdrStyle->SetPadBorderSize(Width_t size = 1);
  tdrStyle->SetPadColor(kWhite);
  tdrStyle->SetPadGridX(false);
  tdrStyle->SetPadGridY(false);
  tdrStyle->SetGridColor(0);
  tdrStyle->SetGridStyle(3);
  tdrStyle->SetGridWidth(1);

// For the frame:
  tdrStyle->SetFrameBorderMode(0);
  tdrStyle->SetFrameBorderSize(1);
  tdrStyle->SetFrameFillColor(0);
  tdrStyle->SetFrameFillStyle(0);
  tdrStyle->SetFrameLineColor(1);
  tdrStyle->SetFrameLineStyle(1);
```

```
  tdrStyle->SetFrameLineWidth(1);

// For the histo:
  // tdrStyle->SetHistFillColor(1);
  // tdrStyle->SetHistFillStyle(0);
  tdrStyle->SetHistLineColor(1);
  tdrStyle->SetHistLineStyle(0);
  tdrStyle->SetHistLineWidth(1);

  tdrStyle->SetEndErrorSize(2);
  //tdrStyle->SetErrorMarker(20);
  tdrStyle->SetErrorX(0.);

  tdrStyle->SetMarkerStyle(20);

//For the fit/function:
  tdrStyle->SetOptFit(1);
  tdrStyle->SetFitFormat("5.4g");
  tdrStyle->SetFuncColor(2);
  tdrStyle->SetFuncStyle(1);
  tdrStyle->SetFuncWidth(1);

//For the date:
  tdrStyle->SetOptDate(0);
  // tdrStyle->SetDateX(Float_t x = 0.01);
  // tdrStyle->SetDateY(Float_t y = 0.01);

// For the statistics box:
  tdrStyle->SetOptFile(0);
  tdrStyle->SetOptStat(0); // To display the mean and RMS:   SetOptStat("mr");
  tdrStyle->SetStatColor(kWhite);
  tdrStyle->SetStatFont(42);
  tdrStyle->SetStatFontSize(0.025);
  tdrStyle->SetStatTextColor(1);
  tdrStyle->SetStatFormat("6.4g");
  tdrStyle->SetStatBorderSize(1);
  tdrStyle->SetStatH(0.1);
  tdrStyle->SetStatW(0.15);
  // tdrStyle->SetStatStyle(Style_t style = 1001);
  // tdrStyle->SetStatX(Float_t x = 0);
```

```
  // tdrStyle->SetStatY(Float_t y = 0);

// Margins:
  tdrStyle->SetPadTopMargin(0.15);
  tdrStyle->SetPadBottomMargin(0.13);
  tdrStyle->SetPadLeftMargin(0.13);
  tdrStyle->SetPadRightMargin(0.15);

// For the Global title:

//  tdrStyle->SetOptTitle(0);
  tdrStyle->SetTitleFont(42);
  tdrStyle->SetTitleColor(1);
  tdrStyle->SetTitleTextColor(1);
  tdrStyle->SetTitleFillColor(10);
  tdrStyle->SetTitleFontSize(0.05);
  // tdrStyle->SetTitleH(0); // Set the height of the title box
  // tdrStyle->SetTitleW(0); // Set the width of the title box
  // tdrStyle->SetTitleX(0); // Set the position of the title box
  // tdrStyle->SetTitleY(0.985); // Set the position of the title box
  // tdrStyle->SetTitleStyle(Style_t style = 1001);
  // tdrStyle->SetTitleBorderSize(2);

// For the axis titles:

  tdrStyle->SetTitleColor(1, "XYZ");
  tdrStyle->SetTitleFont(42, "XYZ");
  tdrStyle->SetTitleSize(0.06, "XYZ");
// The inconsistency is great!
  tdrStyle->SetTitleXOffset(1.0);
  tdrStyle->SetTitleOffset(1.5, "Y");

// For the axis labels:

  tdrStyle->SetLabelColor(1, "XYZ");
  tdrStyle->SetLabelFont(42, "XYZ");
  tdrStyle->SetLabelOffset(0.007, "XYZ");
  tdrStyle->SetLabelSize(0.05, "XYZ");

// For the axis:
```

```
  tdrStyle->SetAxisColor(1, "XYZ");
  tdrStyle->SetStripDecimals(kTRUE);
  tdrStyle->SetTickLength(0.03, "XYZ");
  tdrStyle->SetNdivisions(510, "XYZ");
  tdrStyle->SetPadTickX(1);  // To get tick marks on the opposite side of the frame
  tdrStyle->SetPadTickY(1);

// Change for log plots:
  tdrStyle->SetOptLogx(0);
  tdrStyle->SetOptLogy(0);
  tdrStyle->SetOptLogz(0);

  tdrStyle->SetPalette(1,0);
  tdrStyle->cd();
}
```

If your working with one of the major experiments, they'll most likely have a style for you to use (It will invariably be 95% the same as above, but the 5% will make **all** the difference).

Transparent Plots Add this to your `~/.rootrc` (or create it if it doesn't exist):

```
# GUI specific settings
Gui.Backend: qt
Gui.Factory: qt
```

This sets the graphics backend to qt (you have to have built root with qt support to use this feature).

Now, in your plotting code:

```
TColor* color = gROOT->GetColor(TColor::GetColor(red,green,blue));//Use ints from 0 to
color->SetAlpha(0.5);//0 is fully transparent, 1 fully opaque
hist->SetFillColor(color->GetNumber());
```

Is this a clean interface? No, but it can be just what your graphic needs to remain clear without cluttering the canvas with hatching.

Two warnings:

1. As of this writing, this is only supported for "popular" output formats (pdf, svg, gif, jpg, and png), though notably **not** postscript (ie ps).

2. It's very easy to create a shade that cannot be properly rendered on a projector, making the transparent component of your plots invisible.
3. This only works with the latest version of ROOT 5.34

More information:

- here

Extending ROOT with custom classes

Sometimes it is useful to add your own developed classes to ROOT (so they can be used in ROOT macros/PyROOT scripts or so they can be stored in a ROOT file). For example you may have a class of the form (in a file called CustomEvent.h, for example):

```
#ifndef CustomEvent_h
#define CustomEvent_h

#include "TObject.h"

class CustomEvent : public TObject {
  ClassDef(CustomEvent,1);
private:
  int    _eventID;
  double _eventEnergy;
public:
  CustomEvent() {}
  virtual ~CustomEvent() {}

  void set_eventID(const int eid);
  void set_eventEnergy(const double e);

  int eventID() const { return _eventID; }
  double eventEnergy() const { return _eventEnergy; }
};

#endif
```

It is a good idea to inherit from TObject if you require reading or writing objects to disk. Official documentation of the ins and outs of adding classes can be found in Chapter 15 of the User's Guide, as well as information here: cint and ClassDef. We also need CustomEvent.cxx:

```
#include "CustomEvent.h"

CustomEvent::CustomEvent() {}
CustomEvent::~CustomEvent() {}

void CustomEvent::set_eventID(const int eid) { _eventID = eid; }
void CustomEvent::set_eventEnergy(const double e) { _eventEnergy = e; }
```

To make this class accessible within (Py)ROOT you must create a LinkDef.h file of the form:

```
#ifdef __CINT__
#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class CustomEvent+;

#endif
```

Now you must generate a dictionary for the class:

```
rootcint -f CustomEventDictionary.cxx -c CustomEvent.h LinkDef.h
```

This generates `CustomEventDictionary.cxx` and `CustomEventDictionary.h`. Now you can compile the source for your class and the new dictionary source. Then finally link them together in a library:

```
g++ -fPIC -c CustomEvent.cxx `root-config --cflags`
g++ -fPIC -c CustomEventDictionary.cxx `root-config --cflags`
g++ -shared -o libCustomEvent.so CustomEvent.o CustomEventDictionary.o `root-config --
```

Now by loading the library libCustomEvent.so you can use your class within ROOT. For example, the macro:

```
{
  gSystem->Load("libCustomEvent");
  CustomEvent a;
  a.set_eventID(42);
  std::cout << a.eventID() << std::endl;
}
```

Important Gotcha's

At some point you'll get stuck. Hopefully you'll be stuck on a good problem, but more often than not you'll be stuck on some quirk that ROOT has. Remember ROOT's mantra: "It's not a bug, it's a feature!"

ROOT's object protocol is very strange. The naming schema is based on an industry standard for C programs where it's not possible to use namespaces. The result is very confusing for new users (very good for HEP). Every object in root that can be written to disk (ie saved in a ROOT file) derives from a `TObject` base class. This base class defines a protocol for objects. (All objects can print themselves, have a name, have a title, have a class name, etc). This makes it possible to have a list of disparate objects (as long as it's a list of `TObject*`). As you gain more experience with ROOT, this becomes a power tool. Like any power tool (as Tim Taylor can attest), this can be abused to no end.

TTrees

Drawing trees When you call `TTree::Draw` to draw multi-dimensional histograms, the order of the axes is "z:y:x" rather than the expected "x:y:z"

Caching trees Use `TTreeCache` to loop over trees rather than the standard `TTree::GetEntry(i)` idiom. A `TTreeCache` learns which branches you access most often and caches them, speeding up your processing time significantly. Documentation here. Since you won't read to the end (no one does...) here is the docs for when **not** to use a `TTreeCache`:

```
  SPECIAL CASES WHERE TreeCache should not be activated


 When reading only a small fraction of all entries such that not all branch
 buffers are read, it might be faster to run without a cache.


 HOW TO VERIFY That the TreeCache has been used and check its performance


Once your analysis loop has terminated, you can access/print the number
of effective system reads for a given file with a code like
(where TFile* f is a pointer to your file)
```

```
printf("Reading %lld bytes in %d transactions\n",f->GetBytesRead(),  f->GetReadCalls(
```

Splitting Trees If you want to split a TTree into n statistically independent parts use something like:

```
TTree* outTree=tree->CopyTree("Entry$%n==i");
```

Here, `n` is the number of parts requested, `i` is the i'th part. If you're just splitting it in half, a full blown macro (from the trenches) would look like:

```
// A macro to split a tree
{
  TFile* file=TFile::Open("./merged_dijets.root");
  TTree* tree=(TTree*)file->Get("micro");
  TFile* fileA=new TFile("UnfoldingStudy.dijets-pt1.root","RECREATE");
  fileA->cd();
  TTree* treeA=tree->CopyTree("Entry$%2==0");
  treeA->Write();
  fileA->Write();
  fileA->Close();
  TFile* fileB=new TFile("UnfoldingStudy.dijets-pt2.root","RECREATE");
  fileB->cd();
  TTree* treeB=tree->CopyTree("Entry$%2==1");
  treeB->Write();
  fileB->Write();
  fileB->Close();

}
```

TH1 Despite the name, TH1 is the base class for all histograms. This can lead to much !FUN!. Be extra wary of null pointers when handling TH1's of unknown origin.

TH2

Splitting a 2D One would expect an interface method like `TH2D::Split()`, but instead you need to use the appropriate THStack constructor: `THStack(const TH1* hist, Option_t* axis = "x", ...)`

Then call `THStack::GetHists` to get a TList of the histograms. Of course, you'll have to use ROOT's idioms for iterating over lists,

Another, slightly more direct option is `TH2::ProjectionX()` and `TH2::ProjectionY()`, used in the following fashion:

```
std::vector<TH1D*> split_list;
for(size_t i = 1; i < Hist2D->GetNbinsX()+1; i++) {
  split_list.push_back(Hist2D->ProjectionY("_py",i,i+1,"e"));
}
```

Depending on your ROOT version, you may have to change the ="$_{\text{py}}$"= string to be something unique. This can be done as follows:

```
std::vector<TH1D*> split_list;
char buff[256];
for(size_t i = 1; i < Hist2D->GetNbinsX()+1; i++) {
  snprintf(buff,sizeof(buff)/sizeof(*buff),"%s_%u_py",Hist2D->GetName(),i);
  split_list.push_back(Hist2D->ProjectionY(buff,i,i+1,"e"));
}
```

User Beware: not using a unique name can cause unexpected behavior when drawing the list of slices.

THStack `THStack` is not mostly harmless. In fact, if you want to do reasonable things with `THStack`, it probably won't work, and if it does, it may work once and not the second time. If you can avoid `THStack`, then do it.

Get Sum of Stack To get a histogram representing the sum of a stack use `THStack::GetStack()->Last()`

A pointer here, a pointer there, who am I? If you want to iterate over the stack, there are two ways to get the underlying objects. They are not equal. Option 1:

```
THStack stack = new THStack("Stack","A stack of histograms");
// later on ...
TIter next(stack->GetHists());
TH1* hist = NULL;
while((hist=dynamic_cast<TH1*>(next()))){
  // Do something with *original* histograms added to stack
}
```

Option 2:

```
TH1* hist = NULL;
for(int i = 0; i < stack->GetStack()->GetEntries(); i++){
  hist = dynamic_cast<TH1*>(stack->GetStack()->At(i));
  // Do something with histograms to be painted on the canvas
}
```

These two methods appear equal but are not. The `TList` method from `THStack::GetHists` give you the original pointers (ie those that you added with `THStack::Add`). The `TObjArray` method from `THStack::GetStack` gives you the internal histograms which will be used to draw the histograms on the canvas.

TFile TFile's are greedy about object ownership. In fact, object ownership in ROOT is a very common ~~bug~~ feature. Many times you'll own objects you thought you didn't (memory leak) or, you'll delete objects you thought you did (double free core-dump).

The rule of thumb to keep in mind is "TObjects declared after a file is opened are owned by previously opened file"

Contrast:

```
TH1F hist("hist","Higgs Discovery Plot", 50,0,200);
TFile output("discovery.root","RECREATE");
output.Close();
```

with:

```
TFile output("discovery.root","RECREATE");
TH1F hist("hist","Higgs Discovery Plot", 50,0,200);
output.Close();
```

In the former, the file `discovery.root` will be empty. In the latter, it will contain a copy of `hist`.

This can get really hairy when you're dealing with pointers. Therefore (instead of being a responsible programmer), the best approach to managing memory in ROOT is to not manage it until you have to.

Recreate, create, new, update From the ROOT docs:

```
If option = NEW or CREATE   create a new file and open it for writing,
                            if the file already exists the file is
                            not opened.
          = RECREATE        create a new file, if the file already
```

```
                    exists it will be overwritten.
          = UPDATE  open an existing file for writing.
                    if no file exists, it is created.
          = READ    open an existing file for reading (default).
```

Important: Recreating will destroy the file if it exists. BE CAREFUL when you use this option!

Extracting ACLiCs compilation steps Here's a tip you hope you never need. The use case is when you are writing a standalone program and you know you need to invoke rootcint to generate dictionaries. When you follow the correct prescription, the produced dictionary fails even when linked to your code. What you can do is follow the "loader.C prescription" and then dump what ACLiC is doing under the hood with:

```
root [7] gDebug=7
(const int)7
root [8] .L libLoader.C++
```

This will dump all of the g++ and rootcint calls that are required to generate a working dictionary.

Debugging with ROOT

Eventually you'll encounter a segmentation fault or segfault in ROOT. (They can also go under the name core dump). This happens when you try to read, write, or otherwise abuse a part of memory that doesn't belong to you. The result is that the program (ROOT usually) crashes. ROOT has gotten pretty good at realizing that this has happened, and printing information about what was going on when the crash happened.

A "crash" course on reading a stack trace Here's a stack trace from a real-live analysis program (SFrame in this case)

```
==========================================================
There was a crash.
This is the entire stack trace of all threads:
==========================================================
#0  0x0000003ba7e9a075 in waitpid () from /lib64/libc.so.6
#1  0x0000003ba7e3c741 in do_system () from /lib64/libc.so.6
#2  0x00002b4f86156256 in TUnixSystem::StackTrace() ()
```

```
   from /cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase/x86_64/root/5.34.07-x86_64-slc5-g
#3  0x00002b4f86155b2c in TUnixSystem::DispatchSignals(ESignals) ()
   from /cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase/x86_64/root/5.34.07-x86_64-slc5-g
#4  <signal handler called>
#5  0x00002b4f93f9a47d in UnfoldingStudy::ExecuteEvent(SInputData const&, double) () f
#6  0x00002b4f85c33616 in SCycleBaseExec::Process(long long) ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#7  0x00002b4f8a62e1e0 in TTreePlayer::Process(TSelector*, char const*, long long, lon
   from /cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase/x86_64/root/5.34.07-x86_64-slc5-g
#8  0x00002b4f85c47ce8 in SCycleController::ExecuteNextCycle() ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#9  0x00002b4f85c43872 in SCycleController::ExecuteAllCycles() ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#10 0x000000000040226c in main ()
==========================================================


The lines below might hint at the cause of the crash.
If they do not help you then please submit a bug report at
http://root.cern.ch/bugs. Please post the ENTIRE stack trace
from above as an attachment in addition to anything else
that might help us fixing this issue.
==========================================================
#5  0x00002b4f93f9a47d in UnfoldingStudy::ExecuteEvent(SInputData const&, double) () f
#6  0x00002b4f85c33616 in SCycleBaseExec::Process(long long) ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#7  0x00002b4f8a62e1e0 in TTreePlayer::Process(TSelector*, char const*, long long, lon
   from /cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase/x86_64/root/5.34.07-x86_64-slc5-g
#8  0x00002b4f85c47ce8 in SCycleController::ExecuteNextCycle() ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#9  0x00002b4f85c43872 in SCycleController::ExecuteAllCycles() ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#10 0x000000000040226c in main ()
==========================================================
```

The numbered lines followed by the memory address (the 64bit hex numbers) represent the order in which each function was called. The most recent call is at the top of the list. Since this code was running a single thread, there is a only one stack trace. If there were multiple threads, there would be a trace for each thread. Typically the fastest route to a user called function is

to look at the portion:

```
The lines below might hint at the cause of the crash.
If they do not help you then please submit a bug report at
http://root.cern.ch/bugs. Please post the ENTIRE stack trace
from above as an attachment in addition to anything else
that might help us fixing this issue.
==========================================================
#5  0x00002b4f93f9a47d in UnfoldingStudy::ExecuteEvent(SInputData const&, double) () f
#6  0x00002b4f85c33616 in SCycleBaseExec::Process(long long) ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#7  0x00002b4f8a62e1e0 in TTreePlayer::Process(TSelector*, char const*, long long, lon
   from /cvmfs/atlas.cern.ch/repo/ATLASLocalRootBase/x86_64/root/5.34.07-x86_64-slc5-g
#8  0x00002b4f85c47ce8 in SCycleController::ExecuteNextCycle() ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#9  0x00002b4f85c43872 in SCycleController::ExecuteAllCycles() ()
   from /home/dmb60/bFrame/SFrame/lib/libSFrameCore.so
#10 0x000000000040226c in main ()
==========================================================
```

This strips out the system calls that clutter the full trace, and the top frame (#5 in this case) is the last function called that was defined (`UnfoldingStudy::ExecuteEvent`). This means that somewhere in that function, someone tried to access memory they shouldn't have.

You can trace the whole program from the `main()` invocation. To gain more insight into all the information contained in the stack trace, it is very useful to go through a `gdb` tutorial.

Also, see Getting Help for more problem solving strategies before filing a bug report. Remember: you probably just found a feature, not a bug.

Using gdb Here are some good gdb (GNU DeBugger) tutorials

- Debugging under Unix: gdb Tutorial
- RMS's gdb Debugger Tutorial (not the same RMS)
- GDB manual from GNU.org
- Debugging ROOT with GDB, somewhat dated but the information is relevant. Written by Axel Naumann, a real-life ROOT hacker!

Valgrind If you have memory related problems, you should be aware of valgrind. The quick-start is here.

Physics

At some point (not necessarily right away) a hitchhiker will want to better understand the physics underlying the research he/she is doing. There are numerous textbooks on the subject, but a few stand out as particularly good.

- Introduction to Elementary Particles 2nd Ed. - David Griffiths
- Quarks and Leptons: An Introductory Course in Modern Particle Physics - Francis Halzen and Alan D. Martin

Study Guide For Griffiths

If you've never touched HEP or heard of Particle Physics, read chapter 1 and 2. Otherwise here's a rough path through the book (with suggested exercises to work):

- Chapter 3 (Problems 3.4, 3.14 (for fun), 3.15, 3.16, 3.25, 3.26 (last two cover Mandelstam variables, which are very useful tools))
- Chapter 6 (Problems 6.8 or 6.9, 6.12, 6.13, 6.14, 6.15)
- Chapter 7 (Problems 7.6, 7.8 (optional), 7.14, 7.23, 7.30, 7.36, 7.37, 7.39, 7.51)
- Chapter 8 (Problems 8.14, 8.15, 8.16, 8.19, 8.23, 8.28)
- Chapter 9 (Problems 9.2, 9.3, 9.6, 9.14, 9.17, 9.20, 9.23, 9.25, 9.31, 9.32)
- Chapter 10 (Problems 10.4 10.5, 10.6, 10.15, 10.16, 10.21, 10.23)

Halzen and Martin has a better treatment of Quantum Chromodynamics, but that just adds another book to your library.

This assumes the reader has covered the material in chapter 4 in a undergraduate quantum course (at the level of Griffiths). Chapter 4 is too much of a review to be useful as a primary source. Griffith's quantum book covers it, but Townsend's "A Modern Approach to Quantum Mechanics" is a better text to have on your bookshelf.

Coordinate Systems used in HEP

HEP uses cylindrical coordinates (well, a combination of cylindrical and spherical really) with the z axis oriented along the beam line, R radially "up" and the ϕ curling right-handed around the beam axis.

In addition, HEP physicists think in terms of a variable called "pseudo-rapidity" denoted η. This is defined as

$$\eta = -\log\left(\tan\frac{\theta}{2}\right)$$

Where θ is the polar angle from the beam azis. Why use this crazy coordinate system? Well, it's related to rapidity, the relativistic counterpart to speed. More important to HEP experiments, *differences* in pseudorapidity are Lorentz invariant *along the beam axis*.

If you want to keep a mental map of angles (from wikipedia):

η	Location
0	"Straight up" (θ=$\pi/2$)
0.88	"Forty Five Degrees" (θ=$\pi/4$)
4.5	"Along the Beam Pipe" (θ=2°)
∞	"Beam Axis" (θ=0)

Monte Carlo Event Weights

This is taken from The Editor's Lab Notebook (which is locked by a password, and hence will only be read by one person):

Someone generates gobs of MC for use in any analysis for some process ($W \to \mu\nu + p$ where p is a parton). I want to study what a variable will look like in data, so I run my analysis over the MC and get a plot out. Then I run the same code over data and plot the two on top of each other. The trouble is that the number of MC events I ran over is not the same as the amount of data I ran over. Now the problem becomes how to appropriately scale the Monte Carlo prediction to match the data (the result of the experiment).

As experimental high energy physicists, we define variables so that we can relate the cross sections predicted by theory to what we measure out of the beam. To this end, we can get the number of expected events from

$$N_D = \sigma\mathcal{L} \tag{1}$$

Here σ is the cross section in barns (typically pico-barns) and $\mathcal{L}$ is the integrated luminosity collected by the experiment. How do we compare this to

the Monte Carlo prediction? There, the computer counts up the number of events that were generated for a specific process, what we want is

$$N_{exp} = WN_{MC} \tag{2}$$

Here N_{exp} is the expected number of events we get from Data. In order to compare data to MC we need to scale N_{exp} to the same order as N_D. Remember, there is a cross section calculated from theory that the MC prediction used, therefore we can write

$$N_{exp} = \sigma_{MC}\mathcal{L} \tag{3}$$

It is possible that σ_{MC} is corrected to the next order. In order to avoid recalculating everything, a k factor is reported as the ratio of σ_{NLO}/σ_{MC}. Then all you have to do is multiply σ_{MC} by the k factor, and the calculation is updated to the newest NLO prediction.

We're interested in the weight, so

$$W = \frac{\sigma_{MC}k\mathcal{L}}{N_{MC}} \tag{4}$$

Now, when each bin in the Monte Carlo histogram is scaled by W, it will be equal to the theoretically expected yield calculated by $\sigma_{MC}k$.

Drawing Feynman Diagrams Digitally

There are many possibilities. The most "user friendly" is probably JaxoDraw. It's even got a name reminiscent of Douglas Adams!

JaxoDraw comes as a jar file, you may want to make it more command line friendly by making a script to invoke it (put this in `${HOME}/local/bin/jaxodraw`):

```
#!/bin/bash
java -jar ${HOME}/local/lib/java/jaxodraw-2.1-0.jar $@
```

Now make it executable:

```
chmod +x ${HOME}/local/bin/jaxodraw
```

And add it to your `PATH`. I have my path set up to search this path from my `.bashrc`:

```
export PATH=${HOME}/local/bin:$PATH
```

Now to test it out by typing `jaxodraw` at the command prompt, it should just launch. If it doesn't try again.

Someday you may be working on a presentation and want the figures exported by jaxodraw to have a transparent background. Some kind soul has hacked the conversion steps to do this for you. You can grab the scripts here: `http://personal.psu.edu/jcc8//jd-conversions/`

There are good instructions for using these scripts in this thread. If you have jaxodraw setup as above you can just dump the scripts in `$HOME/local/bin` and run them on the xml file produced by jaxodraw.

What to do if you've lost a 2π

Calm down, take a deep breath and read the first line of The Guide. Then come back here. Somewhere a fellow grad student has a copy of "Introduction to Elementary Particles (2nd Edition)" by David Griffiths. Read Chapter 6 in entirety paying special note to the footnote on page 205.

Responsible Research

This is a topic better covered else where. The part that overlaps with this guide is in documenting the work you do. It is important that you keep a traceable paper-trail of everything you do.

Lab Notebooks

When keeping a lab notebook, it is important to make the barrier for writing something down as small as possible. If it's difficult or inconvenient, you will not be inclined to document as well as you should.

Pen/Paper Notebook This is the Science standard, and it is perfectly applicable to HEP research. Keep it open and write in it as you work. Make sure there is a 1-to-1 mapping from what you are writing in your notebook and what can be found on a hard drive somewhere (plots, text, code etc).

Org-mode Notebook The Editor keeps an org-mode notebook. There is a thorough write-up here.

Flat text file One option is to keep a (well organized) text file for recording what you're doing. Paste links and your thoughts here. Choose a markdown language and write your posts in that. This way you are forced to keep more

structure in the file, and you can export to whatever output formats are supported by your markdown language.

Wiki/ELog Some research groups reserve webspace for hosting private e-logs. These are typically wiki syntax, but they can also be bulletin board style formatting. In any case, it's possible to use a browser extension like "It's All Text!" to use your favorite editor. One downside to these style notebooks is that it is typically awkward to post many plots at once.

Getting Help

For better or worse, eventually you will get stuck. This is Research! If you're not stuck half the time you're not doing it right!

Here's a rough strategy for tackling problems:

1. Google it, don't just Google specific errors, search related terms. Eventually you will be able to make the search more and more general until you get the answer you want, try reading the Google guide.

2. Read documentation. Many times the answer is buried deep inside the program. Don't be afraid to crack open the source code and actually try to understand whats going on (after exhausting any manuals or user guides available).

3. Just take a deep breath, go get a cup of coffee and come back to the problem.

4. If the coffee didn't help, table the problem for the moment and work on something else. Most likely when you come back to it (today or tomorrow) the solution will be obvious.

    - Now may be an appropriate time for an email to the person you are working with directly. Chances are they've already encountered and solved your problem.

5. If you're still stuck, it may be time to post to a forum or mailing list. In general, when soliciting help from people you don't know, it is polite to avoid contacting them directly.

    - Above all else, **do not** be a help vampire. (They do exist!)

FAQ

- I tried: `#include <TheAnswerIsFortyTwo.h>` in C++ but it doesn't work, what gives?

In short: "`#include <...>`" instructs the compiler to look in specific system include directories. In contrast '`#include "..."`' instructs the compiler to look in all directories that it can find (usually specified with the `-I` flag). (Further information)

- I was typing along and all of the sudden my terminal froze!? The only way to continue was to exit the program!

Most likely you accidentally typed "C-s" (the control key followed by the s key). This sends the XOFF command to your terminal emulator. To fix it (and "unfreeze" your terminal) type "C-q". More Info Here.

- Dear Mr. The Editor, my <insert problem here>, can you fix it?

No. Please refer to Getting Help.

- I have some information that would be useful for your guide, can you use it?

Yes! Please see Contributing.

- I'm lost and confused, your guide is overwhelming and overbearing, but I really like HEP, what can I do?

Please don't despair, with time things will come into focus. In the mean time, it will probably be useful to seek out other sources of information. If you're having trouble finding material, you might try the Google guide. No, this is not intended to be an insult, a surprising fraction of people don't know how to use Google, even though it's a verb.

- I have a question not covered by this FAQ that doesn't involve a research problem, whats the best way to communicate it to you?

You can email me at david.b@duke.edu. If it is related to some technical issue and you already have a GitHub account, please open an issue on GitHub's issue tracker for this text.