

# Basic Data Analysis Using ROOT

## *Introduction*

This tutorial started as a one-day class I taught in 2001. Over the years, I've revised it as different versions of ROOT came out, and in response to feedback we've received from the students.

In 2009, I was asked to expand the class to two full days. In past years, many students hadn't gone through all the lessons that were intended to be done in a single day. So I added a set of advanced exercises for those students who knew enough C++ to get through the original material quickly, but allowed for the rest of the students to do in two days what earlier classes had been asked to do in one.

For 2010, in response to student feedback, what had been a single solid day of work was split into two half-day classes. Instead of eliminating the advanced exercises, I divided the two full days of 2009 into four parts, each part roughly corresponding to a half-day's work. This still allows each student to determine their own pace as they work through the material, but gives the more experienced students something to do.

If you're working with C++ for the first time, then it will probably take you at least a half-day per part. Even someone with years of prior experience in ROOT and C++ might barely cover two parts in half a day.

Most of the lessons have time estimates at the top. These are only rough estimates; some students take 45 minutes to go through a lesson labeled "15 minutes", others take only 5.

Don't be too concerned about time. The important thing is for you to learn something, not to punch a time clock. To put it another way: No one expects you to get through all 58 pages before you start your physics work for the summer.

You can find this tutorial in Postscript and PDF format (along with links to the sample files) at <http://www.nevis.columbia.edu/~seligman/root-class/>.

## *A guide to this tutorial*

If you see a command in this tutorial that's preceded by "[ ]", it means that it is a ROOT command. Type that command into the ROOT program as appropriate, *without* the "[ ]" symbols. For example, if you see

```
[ ] .x treeviewer.C
```

it means to type ".x treeviewer.C" at a ROOT command prompt.

If you see a command in this tutorial is preceded by ">", it means that it is a UNIX command. Type that command into UNIX, *not* into ROOT, and without the ">" symbol. For example, if you see

```
> man less
```

it means to type "man less" at a UNIX command prompt.

## Part One

### *Getting started on Linux at Nevis*

If you're sitting in front of a computer running Linux, just use your account name and password to login.

Click **once** on the Browser icon (at the top or bottom of the screen) to start a web browser. (This either looks like the standard Firefox icon, or like a sphere with a mouse around it.)

Type the following URL in the "Location" field of the web browser: <http://root.cern.ch/>. This is the ROOT web site. You'll be coming back here often, so be sure to bookmark this site.<sup>1</sup> You may also want to download the User's Guide for a handy reference: click on "Documents", then on the "User's Guide" link.

You'll need to open a terminal session to do your work. The menu path is:

Applications -> System Tools -> Terminal

You may find it convenient to add the Terminal application icon to the menu bar or to your desktop. You can do this by selecting Applications -> System Tools -> Terminal on the menu bar, but right-click on the word Terminal instead. That will give you options to create icons on the desktop or the "panel" (menu bar).

For those familiar with Linux-based window managers: No, I don't care whether you use GNOME or KDE. Feel free to switch.

Initially (for the ROOT class), you'll probably be content just to login to the student computers and start working. When you start your real work for this summer, I suggest that you ssh to the main server for your experiment; it's the one listed on "Summer Student Accounts" sheet that I handed out before the class.

---

<sup>1</sup> As of 2009, the ROOT web site link will tend to forward you to pages with the URL <http://root.cern.ch/drupal/>. This shouldn't be too much of a problem, but it's worth noting that they're using the Drupal content management system.

## ***A Brief Intro to Linux***

If you're already reasonably familiar with Linux or UNIX in general, skip this section.

You can spend a lifetime learning Linux; I've been working with UNIX since 1993 and I'm still learning something new every day. The commands below barely scratch the surface.

There are links at <http://www.nevis.columbia.edu/~seligman/root-class/links.html> to sites that can teach you more about Linux.

*To copy a file:* use the **cp** command.

For example, to copy the file "example.C" from the directory "~seligman/root-class" to your current working directory, type:

```
> cp ~seligman/root-class/example.C $PWD
```

In UNIX, the variable \$PWD means the results of the **pwd** ("print working directory") command. (I know that a period (.) is the more usual abbreviation, but many students kept missing the period the first time I taught this class.)

*To look at the contents of a text file:* Use the **less** command.

This command is handy if you want to quickly look at a file without editing it. If the name of the command seems puzzling, it may help to know the **more** command also displays the contents of a text file, and the **less** command was created as a more powerful version of **more**. So to quickly look at the contents of file example.C, type:

```
> less example.C
```

While **less** is running, type a space to go forward one screen, type "b" to go backward one screen, type "q" to quit, and type "h" for a complete list of commands you can use.

*To get help on any UNIX command:* type **man <command-name>**

While **man** is running, you can use the same navigation commands as **less**. For example, to learn about the **ls** command, type:

```
> man ls
```

## ***A Brief Intro to Linux (continued)***

*To edit a file:* I suggest you use the **emacs** command. (If you're already familiar with another editor, such as **xemacs**, **pico** or **vi**, you can use it instead.)

You will almost always want to add an ampersand (&) to the end of any "emacs" command; the ampersand means to run the command as a separate process. So to edit the file `example.C`, type:

```
> emacs example.C &
```

The "emacs" environment is complex, and you can spend a lifetime learning it.<sup>2</sup> For now, just use the mouse to move the cursor and look at the menus. As soon as you can (probably not during this class), I suggest that take the Emacs tutorial by selecting it under the "Help" menu.

Try to learn how to cut and paste in whatever editor you use. If you don't, you'll waste a lot of time today typing the same things over and over again.

Are you quitting the editor after you change a file, only to start up the editor again a moment later? It's time for you to look at the "File" menu. If you're editing many files, try opening them with "File->Open File..." and switch between them using the "Buffers" menu.

*Using the graphics user interface:*

This refers to the classic advantages of using a graphical environment over the command line; for example, opening a file with an appropriate application by double-clicking on its icon in a window. For activities like editing files or developing code, I'm going to recommend against this; almost all physics development work is done on the command line.

However, if all you're going to do is read a file, it's OK to double-click it and let UNIX pick an application for you.

---

<sup>2</sup> I've spent two of your lifetimes already, and the class has just started!

## ***Setting up ROOT (5 minutes)***

ROOT is a robust, complex environment for performing physics analysis, and you can spend a lifetime learning it.<sup>3</sup> Before you start using ROOT at Nevis, you have to type the following command:

```
> setup root
```

The command **setup root** sets some Unix environment variables and modifies your command and library paths. If you feel a need to remove these changes, use the command **unsetup root**.

One of the variables that is set is \$ROOTSYS. This will be helpful to you if you're following one of the examples in the ROOT Users Guide. For example, if you're told to find a file in \$ROOTSYS/tutorials, you'll be able to do this only after you've typed **setup root**.

You have to execute the **setup root** command only once, but you must do it each time you login to Linux. If you wish this command to be automatically executed when you login, you can add it to the .myprofile file in your home directory.

You are going to need to have at least two windows open during this class. One window I'll call your "ROOT command" window; this is where you'll run ROOT. The other is a separate "UNIX command" window. On Unix, you can create a second window with the following command; don't forget the ampersand (&):

```
> xterm &
```

You can also just run the Terminal application again, or select "Open Terminal..." from the File menu of a running Terminal application.

I like to use File->Open Tab... instead, but you can use whichever mode you prefer.

---

<sup>3</sup> Yes, that's three lifetimes so far.

## ***Starting ROOT (5 minutes)***

To actually run ROOT, just type:

```
> root
```

The window in which you type this command will become your ROOT command window.

First you'll see the white-and-blue ROOT window appear on your screen. It will then disappear, and a brief "Welcome to ROOT" display will be written on your command window.

If you grow tired of the introductory graphics window, type "root -l" instead of "root" to start the program.

Click on the ROOT window to select it, if necessary.

You can type "?" (or ".h") to see a list of ROOT commands... but you'll probably get more information than you can use right now. Try it and see.

The most important ROOT line command you need to know is how to quit ROOT. To exit ROOT, type ".q". Do this now, then start ROOT again, just to make sure you can do it.

Sometimes ROOT will crash. If it does, it can get into a state for which ".q" won't work. Try typing ".qqq" (three q) if ".q" doesn't work; if that still doesn't work, try five q, then seven q. Unfortunately, if you type ten q, ROOT won't respond, "You're welcome."

OK, dumb joke. But the tip about ".qqq", ".qqqqq", and ".qqqqqqq" is legitimate. Sometimes I find just typing "q" or using Ctrl-C also works.

## Plotting a function (15 minutes)

This example is based on the first example in Chapter 2 of the ROOT Users Guide (page 10). I emphasize different aspects of ROOT than the Users Guide, and it's a good idea to go over both the example in the Guide and the one below.

Let's plot a simple function. Start ROOT and type the following at the prompt:

```
[ ] TF1 f1("func1","sin(x)/x",0,10)
[ ] f1.Draw()
```

Note the use of C++ syntax to invoke ROOT commands. ROOT may help you out with context-based colors for the keywords it recognizes.

If you have a keen memory (or you type ".h" on the ROOT command line), you'll notice that neither TF1 nor any of its methods are listed as commands, nor will you find a detailed description of TF1 in the Users Guide. The only place that the complete ROOT functionality is documented is on the ROOT web site.

Go to the ROOT web site at <http://root.cern.ch/> (did you remember to bookmark this site?), click on "Reference Guide", then on "Pro Version...", then on "TF1"; you may want to use the browser menu Edit->Find to locate that link. Scroll down the page; you'll see some documentation and examples, the class methods, then method descriptions.

Get to know your way around this web site. You'll come back often.

Also note that when you executed "f1.Draw()", ROOT created a canvas for you named "c1". "Canvas" is ROOT's term for a window that contains ROOT graphics; everything ROOT draws must be inside a canvas.<sup>4</sup>

Bring the window named "c1" to the front by left-clicking on it. As you move the mouse over different parts of the drawing (the function, the axes, the graph label, the plot edges) note how the shape of the mouse changes. Right-click the mouse on different parts of the graph and see how the pop-up menu changes.

Position the mouse over the function itself (it will turn into a pointing finger or an arrow). Right-click the mouse and select "SetRange". Set the range to xmin=-10, xmax=10, and click "OK". Observe how the graph changes.

(continued on the next page)

---

<sup>4</sup> I'm simplifying things here. The actual rule is that everything that ROOT draws must be inside a "TPad." Unless you want to add graphics widgets to a window (e.g., buttons and menus), this distinction won't matter to you.

## ***Plotting a function (continued) (10 minutes)***

Let's start getting into a good habit by labeling our axes. Right-click on the x-axis of the plot, select "SetTitle", enter "x [radians]", and click "OK". Let's center that title: right-click on the x-axis again, select "CenterTitle", and click "OK".

Note that clicking on the title gives you a "TCanvas" pop-up, not a text pop-up; it's as if the title wasn't there. Only if you right-click on the axis can you affect the title. In object-oriented terms, the title and its centering are a property of the axis.

It's a good practice to always label the axes of your plots. Don't forget to include the units.

Do the same thing with the y-axis; call it "sin(x)/x". Select the "RotateTitle" property of the y-axis and see what happens.

You can zoom in on an axis interactively. Left-click on the number "2" on the x-axis, and drag to the number "4". The graph will expand its view. You can zoom in as much as you like. When you've finished, right-click on the axis and select "UnZoom."

You have a lot of control over how this plot is displayed. From the "View" menu, select "Editor". Play around with this a bit. Click on different parts of the graph; notice how the options automatically change. Select "View->Toolbar"; among other options, you can see how you can draw more objects on the plot. There's no simple "Undo" command, as there might be in a dedicated graphics program, but you can usually right-click on an object and select "Delete" from the pop-up menu.

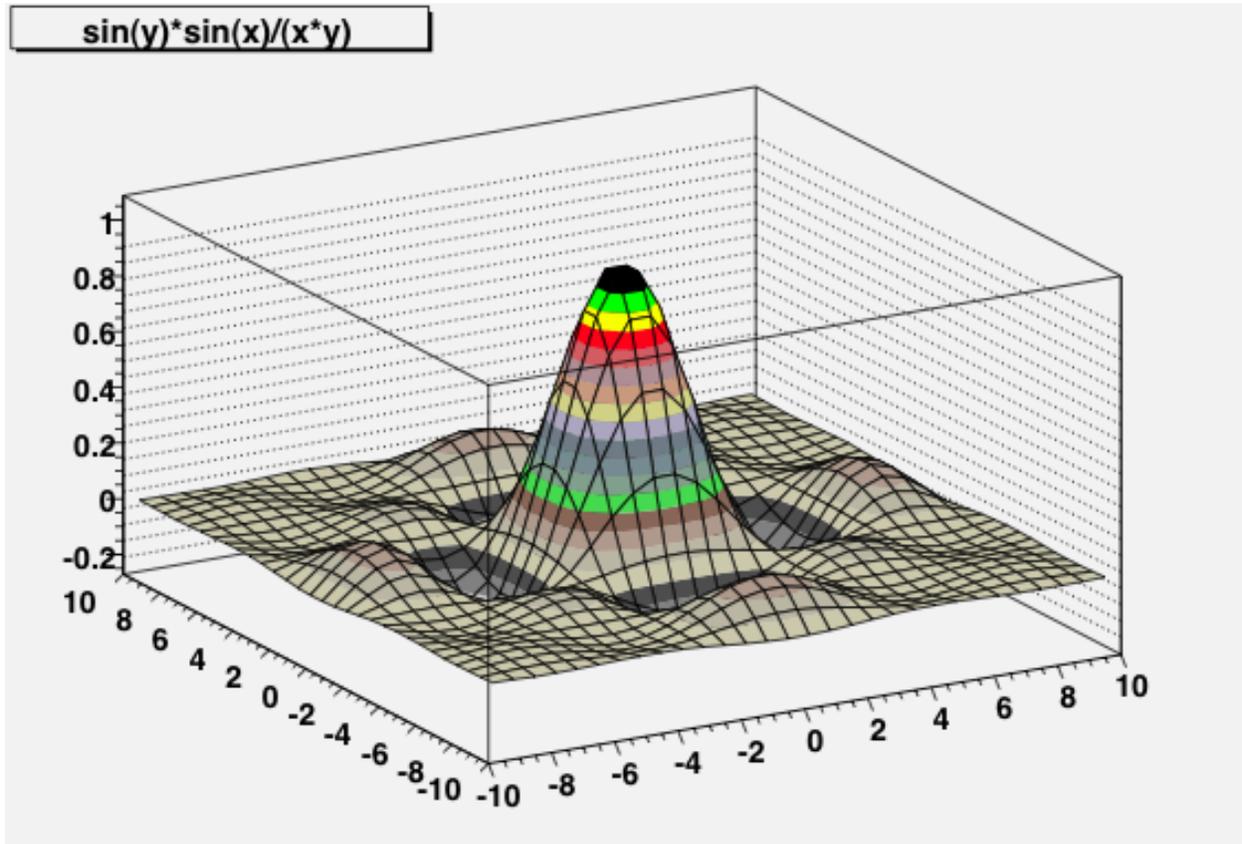
Select "Style" from the "Edit" menu. Select some different styles and hit "Apply"; when you choose a style, it might de-select the window, so you may have to hit "Apply" twice. This can be handy if you discover a "look and feel" that you like for your plots.

There's a Help menu on the upper-right hand corner of this window. Most ROOT windows have such a menu. Take look at its contents. I usually find that the information is enigmatic, but sometimes there's something useful.

If you "ruin" your plot, you can always quit ROOT and start it again. We're not going to work with this plot in the future anyway.

## Exercise 1 (10 minutes)

Try to duplicate the following plot:



Hints:

Look at the TF1 command on the last page. If class TF1 will generate a one-dimensional function, what class might generate a two-dimensional function?

If TF1 takes a function name, formula, and x-limits in its constructor, what arguments might a two-dimensional function class use?

You probably figured out how to draw something, but you got a contour plot, not a surface plot. Here's another hint: you want to give the option "surf1" to the Draw method.

If you're wondering how to figure out that "surf1" was a valid option to give to Draw(): Unfortunately, this is not obvious in the current ROOT web site or documentation. If you're clever and make a couple of guesses, you'll finally end up in the description of the THistPainter class; this describes all the available Draw() options.

This may seem unfair; why did I ask you to chase down an obscure option? Answer: to prepare you for the kind of detective work you often have to do to accomplish something in ROOT; for example, the exercises in Parts Three and Four of this tutorial.

## Working with Histograms (15 minutes)

Histograms are described in Chapter 3 of the ROOT Users Guide. You may want to look over this chapter later to get an idea of what else can be done with histograms other than what I cover in this class.

Let's create a simple histogram:

```
[ ] TH1D h1("hist1","Histogram from a gaussian",100,-3,3)
```

Let's think about what these arguments mean for a moment (and also look at the description of TH1D on the ROOT web site). The name of the histogram is "hist1". The title displayed when plotting the histogram is "Histogram from a gaussian". There are 100 bins in the histogram. The limits of the histogram are from -3 to 3.

Question: What is the width of one bin of this histogram? Type the following to see if your answer is the same as ROOT thinks it is:

```
[ ] h1.GetBinWidth(0)
```

Note that we have to indicate which bin's width we want (bin 0 in this case), because you can define histograms with varying bin widths.<sup>5</sup>

If you type

```
[ ] h1.Draw()
```

right now, you won't see much. That's because the histogram is empty. Let's randomly generate 10,000 values according to a distribution and fill the histogram with them:

```
[ ] h1.FillRandom("gaus",10000)
```

```
[ ] h1.Draw()
```

The "gaus" function is pre-defined by ROOT (see the TFormula class on the ROOT web site; there's also more on the next page of this tutorial). The default Gaussian distribution has a width of 1 and a mean of zero.

Note the histogram statistics in the top right-hand corner of the plot. Question (for those who've had statistics): Why isn't the mean exactly 0, or the width exactly 1?

Add another 10,000 events to histogram h1 with the FillRandom method (hit the up-arrows until you see "h1.FillRandom("gaus",10000)" again, and hit return). Click on the canvas. Does the histogram update immediately, or do you have to type another "Draw" command?

---

<sup>5</sup> For advanced users:

Why would you have varying bin widths? Recall the "too many bins" and "too few bins" examples that I showed in the introduction to the class. In physics, it's common to see event distributions with long "tails." There are times when it's a good idea to have small-width bins in regions with large numbers of events, and large bin widths in regions with only a few events. This can result in having roughly the same number of events per bin in the histogram, which helps with fitting to functions as discussed in the next few pages.

## Working with Histograms (continued) (10 minutes)

Let's put some error bars on the histogram. Select "View->Editor", then click on the histogram. From the "Error" pop-up menu, select "Simple". Try clicking on the "Simple Drawing" box and see how the plot changes.

The size of the error bars is equal to the square root of the number of events in that histogram bin. With the up-arrow key in the ROOT command window, execute the FillRandom method a few more times; draw the canvas again. Question: Why do the error bars get smaller? Hint: Look at how the y-axis changes.

You will often want to draw histograms with error bars. For future reference, you could have used the following command instead of the Editor:

```
[ ] h1.Draw("e")
```

Let's create a function of our own:

```
[ ] TF1 myfunc("myfunc","gaus",0,3)
```

The "gaus" (or gaussian) function is actually

$$P_0 e^{-\frac{(x-P_1)^2}{P_2}}$$

where  $P_0$ ,  $P_1$ , and  $P_2$  are "parameters" of the function.<sup>6</sup> Let's set these three parameters to values that we choose, draw the result, and then create a new histogram from our function:

```
[ ] myfunc.SetParameters(10.,1.0,0.5)
[ ] myfunc.Draw()
[ ] TH1D h2("hist2","Histogram from my function",100,-3,3)
[ ] h2.FillRandom("myfunc",10000)
[ ] h2.Draw()
```

Note that we could also set the function's parameters individually:

```
[ ] myfunc.SetParameter(1,-1.0)
[ ] h2.FillRandom("myfunc",10000)
```

What's the difference between 'SetParameters' and 'SetParameter'? If you have any doubts, check the description of class TF1 on the ROOT web site.

---

<sup>6</sup> For advanced users: In ROOT's TFormula notation, this would be "[0]\*exp(-0.5\*((x-[1])/[2])^2)" where "[n]" corresponds to  $P_n$ . I mention this so that when you become more experienced with defining your own parameterized functions, you can use a different formula:

```
[ ] TF1 myGaus("user","[0]*exp(-.5*((x-[1])/[2])^2)/([2]*sqrt(2.*pi))")
```

This may seem cryptic to you now. It's just a gaussian distribution with a different normalization so that  $P_0$  divided by the bin width becomes the number of events in the histogram:

```
[ ] myGaus.SetParameters(10.,0.,1.)
[ ] hist.Fit("user")
[ ] Double_t numberEquivalentEvents = myGaus.GetParameter(0) /
hist.GetBinWidth(0)
```

## ***Working with multiple plots (optional) (5 minutes)***

If you're running short on time, you can skip this page (or any of the other optional pages).

We have a lot of different histograms and functions now, but we're plotting them all on the same canvas, so we can't see more than one at a time. There are two ways to get around this.

First, we can create a new canvas by selecting "New Canvas" from the File menu of our existing canvas; this will create a new canvas with a name like "c1\_n2". Try this now.

Second, we can divide a canvas into "pads". On the new canvas, right-click in the middle and select "Divide". Enter  $nx=2$ ,  $ny=3$ , and click "OK".

Click on the different pads and canvases with the **middle** button (if you have a mouse with a scroll wheel, the wheel is "clickable" and serves as the middle button). Observe how the yellow highlight moves from box to box. The "target" of the Draw() method will be the highlighted box. Try it: select one pad with the middle button, then enter

```
[ ] h2.Draw()
```

Select another pad or canvas with the middle button, and type:

```
[ ] myfunc.Draw()
```

At this point you may wish that you had a bigger monitor!

## ***Saving and printing your work (15 minutes)***

By now you've probably noticed the "Save" sub-menu under the "File" menu on the canvas. There are many file formats listed here, but we're only going to use three of them for this tutorial.

Select "Save->canvas-name.C" from one of the canvases in your ROOT session. Let's assume for the moment that you're working with canvas c1, so the file "c1.C" is created. In your UNIX window, type

```
> less c1.C
```

(If you get complaints about a file not found, the name of the canvas is "cee-one", not "cee-ell.") As you can see, this can be an interesting way to learn more ROOT commands. However, it doesn't record the procedure you went through to create your plots, only the minimal commands necessary to display them.

Next, select "Save->c1.pdf" from the same canvas; we'll print it later.

Finally, select "Save->c1.root" from the same canvas. Let's assume the file is named "c1.root". Now quit ROOT with the ".q" command, and start it again.

To re-create your canvas from the ".C" file, use the command

```
[ ] .x c1.C
```

This is your first experience with a ROOT "macro", a stored sequence of ROOT commands that you can execute at a later time. One advantage of the ".C" method is that you can edit the macro file, or cut-and-paste useful command sequences into macro files of your own.

You can also start ROOT and have it execute the macro all in a single line:

```
> root c1.C
```

Quit ROOT and print out your Postscript file with the command

```
> lpr -Pbw-research c1.pdf
```

This may be point at which you'll notice that the default background color for ROOT plots is not pure white. You can change the background by right-clicking on a canvas and selecting "SetFillAttributes"; you'll have to do this in the regions both outside and inside the plot, and in the text boxes. If you don't want to keep doing this with all your plots, you may want to experiment more with "Edit->Style"; I find that the "Plain" style usually does what I want.

If you want to print directly from the canvas using "File->Print", then type

```
lpr -Pbw-research
```

in the first text box and leave the second one empty.

## ***The ROOT browser (5 minutes)***

The ROOT browser is a useful tool, and you may find yourself creating one at every ROOT session. See page 21 of the ROOT Users Guide to find out how to make ROOT start a new browser automatically each time you start ROOT.<sup>7</sup>

One way to retrieve the contents of file "c1.root" is to use the ROOT browser. Start up ROOT and create a browser with the command:

```
[ ] TBrowser tb
```

In the left-hand pane, scroll to the folder with the same name as your home directory.<sup>8</sup> Scroll through the list of files. You'll notice special icons for any files that end in ".C" or ".root". If you double-click on a file that ends in ".C": if the Editor tab is in front ROOT will display its contents in the editor window; if the Canvas tab is in front, ROOT will execute its contents. Click on the Canvas tab, then double-click on c1.C to see what happens.

Now double-click on "c1.root", then double-click on "c1;1".

What does "c1;1" mean? You're allowed to write more than one object with the same name to a ROOT file (this topic is part of an optional lesson later in this tutorial). The first object has ";1" put after its name, the second ";2", and so on. You can use this facility to keep many versions of a histogram in a file, and be able to refer back to any previous version.

At this point, saving a canvas as a ".C" file or as a ".root" file may look the same to you. But these files can do more than save and re-create canvases. In general, a ".C" file will contain ROOT commands and functions that you'll write yourself; ".root" files will contain complex objects such as n-tuples.

The ROOT browser has other "gee-whiz" features. For example, if you select Browser->New HTML, it will open a new tab and display the ROOT class index web page. Feel free to use this built-in web browser if you wish, though I find Firefox to be more convenient.

As nifty as the ROOT browser is, I'm not going to encourage you to use it. In the work that you'll do this summer, you'll probably reach the limits of what it can do for you, especially if you have to work with large numbers of files, histograms, n-tuples, or plots; I'd rather you understood how to do things using C++ commands.

Still, it's nice to know that it's there, in case (as the name suggests) you just have to browse quickly through just a couple of ROOT files.

---

<sup>7</sup> Hmm. There's a file called `rootlogon.C` in `~seligman/root-class`. I wonder what it does?

<sup>8</sup> The folder hierarchy may be puzzling to you; your home directory will be in `/a/home/<server>/<account>`. For now, don't worry about this. If you'd like to know more, there's a page on automount at <http://www.nevis.columbia.edu/twiki/bin/view/Nevis/Automount>.

## ***Fitting a histogram (15 minutes)***

I created a file with a couple of histograms in it for you to play with. Switch to your UNIX window and copy this file into your directory:<sup>9</sup>

```
> cp ~seligman/root-class/histogram.root $PWD
```

Go back to your browser window. (If you've quit ROOT, just start it again and start a new browser.) Click on the folder in the left-hand pane with the same name as your home directory.

Double-click on "histogram.root". You can see that I've created two histograms with the names "hist1" and "hist2". Double-click on "hist1"; you may have to move or switch windows around to see the "c1" canvas displayed.

You can guess from the x-axis label that I created this histogram from a gaussian distribution, but what were the parameters? In physics, to answer this question we typically perform a "fit" on the histogram: you assume a functional form that depends on one or more parameters, and then try to find the value of those parameters that make the function best fit the histogram.

Right-click on the histogram and select "FitPanel". Under "Fit Function", make sure that "Predef-1D" is selected. Then make sure "gaus" is selected in the pop-up menu next to it, and "Chi-square" is selected in the Fit Settings->Method pop-up menu. Click on "Fit" at the bottom of the panel. You'll see two changes: A function is drawn on top of the histogram, and the fit results are printed on the ROOT command window.

Interpreting fit results takes a bit of practice. Recall that a gaussian has 3 parameters ( $P_0$ ,  $P_1$ , and  $P_2$ ); these are labeled "Constant", "Mean", and "Sigma" on the fit output. ROOT determined that the best value for the "Mean" was  $5.96 \pm 0.03$ , and the best value for the "Sigma" was  $2.47 \pm 0.02$ . Compare this with the Mean and RMS printed in the box on the upper right-hand corner of the histogram. Statistics questions: Why are these values almost the same as the results from the fit? Why aren't they identical?

On the canvas, select "Fit Parameters" from the "Options" menu; you'll see the fit parameters displayed on the plot.

As a general rule, whenever you do a fit, you want to show the fit parameters on the plot. They give you some idea if your "theory" (which is often some function) agrees with the "data" (the points on the plot).

(continued on the next page)

---

<sup>9</sup> If you're going through this class and you're not logged onto a system on the Nevis Linux cluster, you'll have to get all the files from my web site: <http://www.nevis.columbia.edu/~seligman/root-class/files/>.

## ***Fitting a histogram (continued)***

Just as a check, click on "landau" on the FitPanel's "Fit Function" pop-up menu and click on "Fit" again; then try "expo" and fit again.

It looks like of these three choices (gaussian, landau, exponential), the gaussian is the best functional form for this histogram. Take a look at the "Chi2 / ndf" value in the statistics box on the histogram ("Chi2 / ndf" is pronounced "kie-squared per [number of] degrees of freedom"). Do the fits again, and observe how this number changes. Typically, you know you have a good fit if this ratio is about 1.

The FitPanel is good for gaussian distributions and other simple fits. But for fitting large numbers of histograms (as you'd do in Parts Three and Four) or more complex functions, you want to learn the C++ commands.

To fit hist1 to a gaussian, type the following command:

```
[ ] hist1.Fit("gaus")
```

This does the same thing as using the FitPanel. You can close the FitPanel; we won't be using it anymore.

Go back to the browser window and double-click on "hist2".

You've probably already guessed by reading the x-axis label that I created this histogram from the sum of two gaussian distributions. We're going to fit this histogram by defining a custom function of our own.

Define a user function with the following command:

```
[ ] TF1 func("mydoublegaus", "gaus(0)+gaus(3)")
```

Note that the internal ROOT name of the function is "mydoublegaus", but the C++ name is "func".

What does "gaus(0)+gaus(3)" mean? You already know that the "gaus" function uses three parameters. "gaus(0)" means to use the gaussian distribution starting with parameter 0; "gaus(3)" means to use the gaussian distribution starting with parameter 3. This means our user function has six parameters:  $P_0$ ,  $P_1$ , and  $P_2$  are the "constant", "mean", and "sigma" of the first gaussian, and  $P_3$ ,  $P_4$ , and  $P_5$  are the "constant", "mean", and "sigma" of the second gaussian.

(continued on the next page)

## ***Fitting a histogram (continued)***

Let's set the values of  $P_0$ ,  $P_1$ ,  $P_2$ ,  $P_3$ ,  $P_4$ , and  $P_5$ , and fit the histogram.

```
[ ] func.SetParameters(5.,5.,1.,1.,10.,1.)  
[ ] hist2.Fit("mydoublegaus")
```

It's not a very good fit, is it? This is because I deliberately picked a poor set of starting values. Let's try a better set:

```
[ ] func.SetParameters(5.,2.,1.,1.,10.,1.)  
[ ] hist2.Fit("mydoublegaus")
```

These simple fit examples may leave you with the impression that all histograms in physics are fit with gaussian distributions. Nothing could be further from the truth. I'm using gaussians in this class because they have properties (mean and width) that you can determine by eye.

Chapter 5 of the ROOT Users Guide has a lot more information on fitting histograms, and a much more realistic example.

If you want to see how I created the file histogram.root, go to the UNIX window and type:

```
> less ~/seligman/root-class/CreateHist.C
```

In general, for fitting histograms in a real analysis, you'll have to define your own functions and fit to them directly, with commands like:

```
[ ] TF1 func("myFunction","<...some parameterized TFormula...>")  
[ ] func.SetParameters(...some values...)  
[ ] myHistogram.Fit("myFunction")
```

For a simple gaussian fit to a single histogram, you can always go back to using the FitPanel.

## ***Saving your work, part 2 (15 minutes)***

So now you've got a histogram fitted to a complicated function. If you were to use "Save as c1.root", quit ROOT, restart it, then load canvas "c1;1" from the file, you'd get your histogram back with the function superimposed... but it's not obvious where the function is or how to access it now.

What if you want to save your work in the same file as the histograms you just read in? You can do it, but not by using the ROOT browser. The browser will only open .root files in read-only mode. To be able to modify a file, you have to open it with ROOT commands.

Try the following: Quit ROOT (note that you can select "Quit ROOT" from the "Browser" menu of the browser or the "File" menu of the canvas). Start ROOT again, then modify "histogram.root" with the following commands:

```
[ ] TFile file1("histogram.root","UPDATE")
```

It is the "UPDATE" option that will allow you to write new objects to "histogram.root".

```
[ ] hist2.Draw()
```

For the following two commands, try hitting the up-arrow key until you see them again. ROOT stores the last 80 or so ROOT commands you've typed in the file ".root-hist" in your home directory, and let's you re-use them with the arrow keys.

```
[ ] TF1 func("user","gaus(0)+gaus(3)")
```

```
[ ] func.SetParameters(5.,2.,1.,1.,10.,1.)
```

```
[ ] hist2.Fit("user")
```

Now you can do what you couldn't before: save objects into the ROOT file:

```
[ ] hist2.Write()
```

```
[ ] func.Write()
```

Close the file to make sure you save your changes (optional; ROOT usually closes the file for you when you quit the program):

```
[ ] file1.Close()
```

Quit ROOT, start it again, and use the ROOT browser to open "histogram.root". You'll see a couple of new objects: "hist2;2" and "user;1". Double-click on each of them to see what you've saved.

You wrote the function with `func.Write()`, but you saw `user;1` in the file. Do you see why? It has to do with the name you give to C++ objects, versus the internal name that you give to ROOT. I won't dwell on this issue, but I wanted to point it out so that you were aware that, though they seem closely connected at times, C++ and ROOT are two *different* entities.

Chapter 11 of the ROOT Users Guide has more information on using ROOT files.

## ***Dealing with PAW files (optional; for reference only) (5 minutes)***

Before ROOT, physicists used a package called CERNLIB to analyze data. You won't be asked to work with CERNLIB while you work at Nevis (at least, I hope not), but it may be that you'll be asked to read a file created by this old program library.

Suppose someone gives you a file that contains n-tuples or histograms, and tells you that the file was created with PAW, HBOOK, or CERNLIB (actually, to first order these are three different names for the same thing). How do you read these files using ROOT?

The answer is that you can't, at least not directly. You must convert these files into ROOT format using the command "h2root".

For example, if someone gives you a file called "testbeam.hbook", you can convert it with the command

```
> h2root testbeam.hbook
```

This creates a file "testbeam.root" that you can open in the ROOT browser.

There is no simple way of converting a ROOT file back into PAW/HBOOK/CERNLIB format. You generally have to write a custom program with both FORTRAN and C++ subroutines to accomplish this task.

Note that the "h2root" command is set up (along with ROOT) with the command

```
> setup root
```

that you type when you log in. If you accidentally type "h2root" (or "root") before you set up ROOT, you'll get the error message:

```
h2root: Command not found
```

You can get more information about "h2root" by using a special form of the "man" command:

```
> man $ROOTSYS/man/man1/h2root.1
```

There's also information on page 22 of the ROOT Users Guide.

## Accessing variables in ROOT NTuples/Trees (10 minutes)

I've created a sample ROOT n-tuple in `~/seligman/root-class/experiment.root`.

Start fresh by quitting ROOT. Copy my 2.1 MB example file with the command

```
> cp ~/seligman/root-class/experiment.root $PWD
```

Start ROOT again. Start a new browser with the command

```
[ ] TBrowser b
```

Click on the folder in the left-hand pane with the same name as your home directory. Double-click on "experiment.root". There's just one object inside: "tree1", a ROOT Tree (or n-tuple) with 100,000 simulated physics events.

Actually, there's little or no real physics associated with the contents of this tree. I created it solely to illustrate ROOT concepts, not to demonstrate real physics with a real detector.

Right-click on the "tree1" icon, and select "Scan". You'll be presented with a dialog box; just hit "OK" for now. Select your ROOT window, even though the dialog box didn't go away. At first you'll probably just notice that it's a lot of numbers. Take a look at near the top of the screen; you should see the names of the variables in this ROOT Tree.

In this overly-simple example, an imaginary particle is traveling in a positive direction along the z-axis with energy "ebeam". It hits a target at  $z=0$ , and travels a distance "zv" before it is deflected by the material of the target. The particle's new trajectory is represented by "px", "py", and "pz", the final momenta in the x-, y-, and z-directions respectively. The variable "chi2" represents a confidence level in the measurement of the particle's momentum.

Did you notice what's missing from the above description? One important omission is the units; for example, I didn't tell you whether "zv" is in millimeters, centimeters, inches, yards, etc. Such information is not usually stored inside an n-tuple; you have to find out what it is and include the units in the labels of the plots you create.<sup>10</sup> For this example, assume that "zv" is in centimeters (cm), and all energies and momenta are in GeV.

You can hit "return" to see more numbers, but you probably won't learn much. Hit "q" to finish the scan. You may have to hit "return" a couple of times to see the ROOT prompt again.

---

<sup>10</sup> Advanced note: There is a way of storing comments about the contents of a ROOT tree, which can include information such as units. However, you can't do this with n-tuples; you have to create a C++ class that contains your information in the form of comments, and use a ROOT "dictionary" to include the additional information. This is outside the scope of what you'll be asked to do this summer, but if you're interested in the concept, it's described in Chapter 15 of the Root User's Guide. You'll also have a chance to look at an example in Part Four of this class.

## ***Simple analysis using the Draw command (10 minutes)***

It may be that all the analysis tasks that your supervisor will ask you to do can be performed using the Draw command, the TreeViewer (see page 41), the FitPanel and other simple techniques discussed in the ROOT Users Guide.

However, it's more likely that these simple commands will only be useful when you get started; for example, you can draw a histogram of just one variable to see what the histogram limits might be in C++. Let's start with the simple tricks, then move on to more realistic techniques.

If you don't already have the sample ROOT TTree file open, open it with the following command:

```
[ ] TFile myFile("experiment.root")
```

You can use the Scan command to look at the contents of the Tree, instead of using the TBrowser as described on the previous page:

```
[ ] tree1->Scan()
```

If you take a moment to think about it (a habit I strongly encourage), you may ask how ROOT knows that there's a variable named "tree1", when you didn't type in a command to create it.

The answer is that when you read a file containing ROOT objects (see "Saving your work, part 2" on page 18) in an interactive ROOT session, ROOT automatically looks at the objects in the file and creates variables with the same name as the objects.

This is *not* standard behavior in C++; it isn't even standard behavior when you're working with ROOT macros. Don't become too used to it!

You can also display the TTree in a different way that doesn't show the data, but displays the names of the variables and the size of the TTree:

```
[ ] tree1->Print()
```

Either way, you can see that the variables stored in the TTree are "event", "ebeam", "px", "py", "pz", "zv", and "chi2".

Create a histogram of one of the variables. For example:

```
[ ] tree1->Draw("ebeam")
```

Using the Draw command, make histograms of the other variables.

By the way, the variable "event" is just the event number (it's 0 for the first event, 1 for the second event, 2 for the third event... 99999 for the 100,000th event).

## ***Pointers: An all-too-brief explanation (for those who don't know C++ or C) (5 minutes)***

Notice that on the previous page we used the pointer symbols "->" (a dash followed by a greater-than sign) instead of the period "." to issue the commands to the TTree. This is because the variable "tree1" isn't really the TTree itself; it's a 'pointer' to the TTree.

The difference between an object and a pointer in C++ (and ROOT) is a key concept in programming. Unfortunately, a detailed explanation is beyond the scope of this tutorial, although I may say something about this in class. I strongly suggest that you look up the description of pointers that you can find in every introductory text on C++ or C.

For now, I hope it's enough to just show a couple of examples:

```
[ ] TH1D hist1("h1", "a histogram", 100, -3, 3)
```

This creates a new histogram in ROOT, and the name of the 'histogram object' is "hist1". I must use a period to issue commands to the histogram:

```
[ ] hist1.Draw()
```

Here's the same thing, but using a pointer instead:

```
[ ] TH1D *hist1 = new TH1D("h1", "a histogram", 100, -3, 3)
```

Note the use of the asterisk "\*" when I define the variable, and the use of the C++ keyword "new".

In this example, "hist1" is not a 'histogram object,' it's a 'histogram pointer.' I must use the pointer symbols to issue commands:

```
[ ] hist1->Draw()
```

Take another look at the file c1.C that you created in a previous example. Note that ROOT uses pointers for almost all the code it creates. On the previous page, I mentioned that ROOT automatically creates variables when it opens files in interactive mode; these variables are always pointers.

It's a little harder to think in terms of pointers than in terms of objects. However, you have to use pointers if you want to take advantage of the C++ code that ROOT can generate for you automatically.

You also have to use pointers to take advantage of object inheritance and polymorphism in C++. ROOT relies heavily on object inheritance (some would say too heavily); we'll get more into inheritance in Part Three of this tutorial.

## ***Simple analysis using the Draw command, part 2 (10 minutes)***

Instead of just plotting a single variable, let's try plotting two variables at once:

```
[ ] tree1->Draw("ebeam:px")
```

This is a scatterplot, a handy way of observing the correlations between two variables. The Draw command interprets the variables as ("x:y") to decide which axes to use.

Be careful: it's easy to fall into the trap of thinking that each (x,y) point on a scatterplot represents two values in your n-tuple. In fact, the scatterplot is a grid and each square in the grid is randomly populated with a density of dots that's proportional to the number of values in that grid.

Try making scatterplots of different pairs of variables. Do you see any correlations between the variables?

If you just see a shapeless blob on the scatterplot, the variables are likely to be uncorrelated; for example, plot "px" versus "py". If you see a pattern, there may be a correlation; for example, plot "pz" versus "zv". It appears that the higher "pz" is, the lower "zv" is, and vice versa. Perhaps the particle loses energy before it is deflected in the target.

By the way, you can also make three-dimensional plots this way:

```
[ ] tree1->Draw("px:py:pz")
```

After looking at these plots, you can see why it's important to always label your axes!

Let's create a "cut" (a limit on the range of a variable to be plotted):

```
[ ] tree1->Draw("zv", "zv<20")
```

Look at the x-axis of the histogram. Compare this with:

```
[ ] tree1->Draw("zv")
```

Note that ROOT determines an appropriate range for the x-axis of your histogram. Enjoy this while you can; this feature is lost when you start using analysis macros.<sup>11</sup>

---

<sup>11</sup> Another advanced note: If you know what you're doing, you can use the same trick that ROOT uses when it creates the histogram you create with commands like `tree1->Draw("zv")`. The trick is:

```
TH1* hist = new TH1D(...); // define your histogram
hist->SetBit(TH1::kCanRebin); // allow the histogram to re-bin itself
hist->Sumw2(); // so the error bars are correct after re-binning
```

“Re-binning” means that if a value is supplied to the histogram that's outside its limits, it will adjust those limits automatically. It does this by summing existing bins then doubling the bin width; the bin limits change, while the number of histogram bins remains constant.

### ***Simple analysis using the Draw command, part 3 (5 minutes)***

Note that a variable in a cut does not have to be one of the variables you're plotting:

```
[ ] tree1->Draw("ebeam", "zv<20")
```

Try this with some of the other variables in the tree.

The symbol for logical AND in C++ is "&&". Try using this in a cut, e.g.:

```
[ ] tree1->Draw("ebeam", "px>10 && zv<20")
```

For those who know what a “weighted histogram” means:

A "cut" is actually a weight that ROOT applies when filling a histogram; a logical expression has the value 1 if true and the value 0 if false. If you want to fill a histogram with weighted values, use an expression for the cut that corresponds to the weight.

For example: a cut of "1/e" will fill a histogram with each event weighted by 1/e; a cut of "(1/e)\*(sqrt(z)>3.2)" will fill a histogram with events weighted by 1/e, for those events with sqrt(z) greater than 3.2.

## Part Two

### *Using C++ to analyze a Tree (10 minutes)*

You can spend a lifetime learning all the in-and-outs of object-oriented programming in C++.<sup>12</sup> Fortunately, you only need a small subset of this to perform analysis tasks with ROOT. The first step is to have ROOT write the skeleton of an analysis class for your n-tuple. This is done with the MakeClass command.

Let's start with a clean slate: quit ROOT if it's running and start it up again. Open the ROOT tree again:

```
[ ] TFile myFile("experiment.root")
```

Now create an analysis macro for "tree1" with MakeClass. I'm going to use the name 'Analyze' for this macro, but you can use any name you want; just remember to use your name instead of 'Analyze' in all the examples below.

```
[ ] tree1->MakeClass("Analyze")
```

Switch to the UNIX window and examine the files that were created:

```
> less Analyze.h  
> less Analyze.C
```

Remember this from my introductory talk? Unless you're familiar with C++, this probably looks like gobbledy-gook to you. (I know C++, and it looked like gobbledy-gook to *me*... at first.) We can simplify this by understanding the approach of most analysis tasks:

- **Set-up** (open files, define variables, create histograms, etc.).
- **Loop** (for each event in the n-tuple or Tree, perform some tasks: calculate values, apply cuts, fill histograms, etc.).
- **Wrap-up** (display results, save histograms, etc.)

(continued on the next page)

---

<sup>12</sup> That's four lifetimes. And you thought you only signed up for a ten-week project! Gosh, I wonder if it takes a lifetime to understand high-energy physics.

## ***Using C++ to analyze a Tree (continued)***

The C++ code from Analyze.C is on the next page. I've marked the places in the code where you'd place your own commands for Set-up, Loop, and Wrap-up. Compare the code you see in Analyze.C with what I've put on the next page.

You've probably already guessed that lines beginning with `///  
//` are comments. Your next observation may be that the comments put there by ROOT aren't helpful to you. These are the comments that ROOT automatically generates with the `MakeClass` command; you can edit or delete them after they're created, but you can't easily prevent them from being created in the first place.

Note that Loop and Wrap-up are separated by a single right curly bracket `}`. Make sure your commands go in the right place! Also, be careful not to accidentally delete the final `}` in the file when you edit your Wrap-up commands.

Finally, I'm sure you've noticed the comments I put in the code are in a different font on the next page. That's where you're going to put your own analysis code. You can edit `Analyze.C` and put those comments in there to act as placeholders for your code; I suggest you give the file a different name as you edit it, such as `AnalyzeComments.C`. I've already done this for you, and you can copy this code if you wish:

```
> cp ~seligman/root-class/AnalyzeComments.C $PWD
```

```

#define Analyze_cxx
#include "Analyze.h"
#include <TH2.h>
#include <TStyle.h>
#include <TCanvas.h>

void Analyze::Loop()
{
//   In a ROOT session, you can do:
//       Root > .L Analyze.C
//       Root > Analyze t
//       Root > t.GetEntry(12); // Fill t data members with entry number 12
//       Root > t.Show();      // Show values of entry 12
//       Root > t.Show(16);    // Read and show values of entry 16
//       Root > t.Loop();      // Loop on all entries
//
//
//   This is the loop skeleton where:
//   jentry is the global entry number in the chain
//   ientry is the entry number in the current Tree
//   Note that the argument to GetEntry must be:
//   jentry for TChain::GetEntry
//   ientry for TTree::GetEntry and TBranch::GetEntry
//
//       To read only selected branches, Insert statements like:
// METHOD1:
//   fChain->SetBranchStatus("*",0);  // disable all branches
//   fChain->SetBranchStatus("branchname",1); // activate branchname
// METHOD2: replace line
//   fChain->GetEntry(jentry);       //read all branches
//by  b_branchname->GetEntry(ientry); //read only this branch
   if (fChain == 0) return;

      // The Set-up code goes here.
      Long64_t nentries = fChain->GetEntriesFast();

      Long64_t nbytes = 0, nb = 0;
      for (Long64_t jentry=0; jentry<nentries;jentry++) {
         Long64_t ientry = LoadTree(jentry);
         if (ientry < 0) break;
         nb = fChain->GetEntry(jentry);   nbytes += nb;
         // if (Cut(ientry) < 0) continue;

            // The Loop code goes here.

      }

      // The Wrap-up code goes here.
}

```

## ***Running the Analyze macro (10 minutes)***

As it stands, the Analyze macro does nothing, but let's learn how to run it anyway. Quit ROOT, start it again, and enter the following lines:

```
[ ] .L AnalyzeComments.C
[ ] Analyze a
[ ] a.Loop()
```

Have you figured out tab-completion on your own yet? If not, try this when you type the first command above: type ".L An", then hit the tab key, then "C", then hit the tab key again. If ROOT can figure out that you're trying to type in a file name, it will try to complete that name as best it can when you hit the tab key.

By the way, it's not just ROOT that can do this. When you're in the UNIX window and you have a long file name to work with, try typing the first couple of letters and hit tab.

After the last command, ROOT will pause as it reads through all the events in the Tree. Since we haven't included any analysis code yet, you won't see anything else happen.

If you're not familiar with C++, you may be very confused at this point. What do any of the above commands have to do with the file "experiment.root" or the TTree inside it? And what do these commands mean?

Take another look at Analyze.h. If you scan through it, you'll see C++ commands that explicitly refer to the name of the file, the name of the Tree, and its variables. Now go back and look at the top of AnalyzeComments.C. You'll see the line

```
#include Analyze.h
```

This means that ROOT will include the contents of the file Analyze.h when it loads AnalyzeComments.C.

.L AnalyzeComments.C tells ROOT to load the computer code inside the file AnalyzeComments.C, and to interpret the code to create a C++ class. The name of this class will be "Analyze"; look near the top of Analyze.h, and you'll see the C++ keywords "class Analyze".

Analyze a creates an object whose name is "a" (I'll explain this on the next page).

a.Loop( ) tells ROOT to execute the Loop command of object "a". Look at AnalyzeComments.C again. Near the beginning, you'll see the line "void Analyze::Loop". The code in this file, and therefore the code that you'll be working with for the rest of Part Two, defines the Loop command.

## ***Classes and objects: An all-too-brief explanation (for those who don't know C++) (10 minutes)***

So what is a class and what is an object? The way I usually think of it is that a class defines an *abstract* view of a concept, but an object is a *concrete* thing.

Here's an analogy. Suppose I define the concept of a circle. What are the properties of a circle? Well, it has a radius, and the location of its center. Since we're talking about computers, we might think of giving the circle some commands to obey: tell me your circumference, tell me your area. The commands (C++ code) are instructions on what to do with the properties of the circle (e.g.,  $A=4\pi r^2$ ).

If I ask you "Where is the center of a circle?" you'd have trouble answering the question numerically before I've given you anything concrete. I've defined the concepts of a circle, but I haven't drawn an actual circle to work with. To put it in C++ terms, I've created a circle *class*, but not a circle *object*.

Suppose I draw an actual circle. Here it is:



Now you can tell me the center of the drawn circle in some co-ordinate system; for example, you could take a ruler and measure the center from the edge of the page. The object has the hard numbers that allow the circle's commands to calculate numerical values for the circumference, area, and so on.

To put it another way: the class represents the rules for accessing the information; the object holds the specific information.

Assume that I write C++ code to define a circle class. I'm going to put this code in a file whose name is CircleClass.C. I'm going to give the class a name: Circle. That class is going to contain a command: Area.

These are the ROOT commands that might be used to find the area of c, a particular circle:

```
[ ] .L CircleClass.C
[ ] Circle c
[ ] c.Area()
```

(continued on next page)

## ***Classes and objects: (too briefly continued)***

There's something I've avoided so far: I didn't discuss how I might tell ROOT the radius or the actual (x,y) co-ordinates of the center of the circle; in C++ terms, I haven't discussed passing values to the class constructor. The reason why I've skipped over this is that it's not relevant to the Analyze example: the name of the ROOT file, TTree, and variables are specified explicitly in Analyze.h.

However, you have passed values to a constructor before; you did it when you typed this:

```
TH1D myHist("hist","example histogram",100,-3.,3.);
```

This creates an object myHist of class TH1D. After you did that, you could send commands to myHist; e.g.,

```
myHist.Draw();
```

You didn't have to type in something like ".L TH1D.C" because that class is built into ROOT. "Analyze" is *not* built-in, so we have to tell ROOT to load it.

So to do our analysis, we need a file for our C++ code. The file will define a class to perform the analysis. We need to create a concrete object of the class, and we need to send that object the Loop command.

From a practical standpoint, this means that every time you edit the file AnalyzeComments.C, you must re-load it into ROOT, you must create a brand-new object, and you must execute the Loop command again.

Take another look at the three commands at the top of page 28. Get used to them. You'll be typing them (or variants of them) over and over again as you do your analysis. Remember, the up-arrow key is your friend!

## ***Making a histogram with Analyze (15 minutes)***

Enough with the abstract concepts. Let's do some work.

Make a copy of the Analyze.C or AnalyzeComments.C file:

```
> cp AnalyzeComments.C AnalyzeHistogram.C
```

Edit the file AnalyzeHistogram.C. In the Set-up section, put the following code:

```
TH1* chi2Hist = new TH1D("chi2","Histogram of Chi2",100,0,20);
```

In the Loop section, put this in:

```
chi2Hist->Fill(chi2);
```

This goes in the Wrap-up section:

```
chi2Hist->Draw();
```

Don't forget the semi-colons ";" at the ends of the lines! You can omit them in interactive commands, but not in macros.

Save the file, then enter the following commands in ROOT:

```
[ ] .L AnalyzeHistograms.C  
[ ] Analyze a  
[ ] a.Loop()
```

Finally, we've made our first histogram with a C++ analysis macro. In the Set-up section, we defined a histogram; in the loop section, we filled the histogram with values; in the Wrap-up section, we drew the histogram.

How did I know which bin limits to use on "chi2Hist"? Before I wrote the code, I drew a test histogram with the command:

```
[ ] tree1->Draw("chi2")
```

Hmm, the histogram's axes aren't labeled. How do I put the labels in the macro? Here's how I figured it out: I labeled the axes on the test histogram by right-clicking on them and selecting "SetTitle". Then I saved the canvas by selecting "Save->c1.C" from the File menu. Then I looked at c1.C and saw these commands in the file:

```
chi2->GetXaxis()->SetTitle("chi2");  
chi2->GetYaxis()->SetTitle("number of events");
```

I scrolled up and saw that ROOT had used the variable "chi2" for the name of the histogram pointer. I copied the lines into AnalyzeHistogram.C, but used the name of my histogram instead:

```
chi2Hist->GetXaxis()->SetTitle("chi2");  
chi2Hist->GetYaxis()->SetTitle("number of events");
```

Try this yourself: add the two lines above to the Set-up section, right after the line that defines the histogram. Test the revised Analyze class.

The labels overlap the axis numbers. This is good enough for now, but you'll have to figure out how to move the labels if you were ever to publish this plot. You'll deal with this issue in Exercise 11.

## **Exercise 2 (5 minutes)**

We're still plotting the chi2 histogram as a solid curve. Most of the time, your supervisor will want to see histograms with errors. Revise the `Analyze::Loop` method in `AnalyzeHistogram.C` to draw the histograms with error bars.

Hint: Look back at "Working with Histograms" earlier in this tutorial.

Warning: The histogram may not be immediately visible, because all the points are squeezed into the left-hand side of the plot. We'll investigate the reason why in a subsequent exercise.

## **Exercise 3 (15 minutes)**

Revise `AnalyzeHistogram.C` to create, fill, and display an additional histogram of the variable "ebeam" (with error bars and axis labels, of course).

First, some hints for those new to C++:

Take care! On page 25 I broke up a typical physics analysis task into three pieces: the Set-up, the Loop, and the Wrap-up; I also marked the location in the analysis macro where you'd put these steps.

What may not be obvious is that *all* your commands that relate to setting things up must go in the Set-up section, *all* your commands that are repeated for each event must go in the Loop section, and so on. Don't try to create two histograms by copying the entire `Analyze::Loop` program and pasting it into the file more than once; it won't work.

Now, some warnings for everyone:

Prediction: You're going to run into trouble when you get to the Wrap-up section and draw the histograms. When you run your code, you'll probably only see one histogram plotted, and it will be the last one you plot.

The problem is that when you issue the Draw command for a histogram, by default it's drawn on the "current" canvas. If there is no canvas, a default one (our old friend "c1") is created. So both histograms are being drawn to the same canvas.

You can solve the problem in one of two ways: you can create a new canvas for each histogram, or you can create one large canvas and divide it into sub-pads (see the lesson "Working with multiple plots" on page 12). I'll let you pick which to use, but be forewarned: working with pads is more ambitious than creating one canvas for each plot.

More clues: Look at `c1.C` to see an example of how a canvas is created. Look up the `TCanvas` class on the ROOT web site to figure out what the commands do. To figure out how to switch between canvases, look at `TCanvas::cd()` (that is, the `cd()` method of the `TCanvas` class).

## Exercise 4 (10 minutes)

Fit the ebeam histogram to a gaussian distribution.

OK, that part was easy. It was particularly easy because the "gaus" function is built into ROOT, so you don't have to worry about a user-defined function.

Let's make it a bit harder: the parameters from the fit are displayed in the ROOT text window; your task is to put them on the histogram as well. You want to see the parameter names, the values of the parameters, and the errors on the parameters as part of the plot.

This is trickier, because you have to hunt for the answer on the ROOT web site... and when you see the answer, you may be tempted to change it instead of typing in exactly what's on the web site.

Take a look at the description of the TH1::Draw() method. In that description, it says "See THistPainter::Paint for a description of all the drawing options". Click on the word "THistPainter". There's lots of interesting stuff here, but for now focus on the section "Statistics Display". (By the way, now you know how I figured out the "surf1" option for Exercise 1 on page 9).

There was another way to figure this out, and maybe you tried it: Draw a histogram, select "Options->Fit Parameters", fit a function to the histogram, save it as c1.C, and look at the file. OK, the command is there... but would you have been able to guess which one it was if you hadn't looked it up on the web site?<sup>13</sup>

## Exercise 5 (10 minutes)

Now add another plot: a scatterplot of "chi2" versus "ebeam". Don't forget to label the axes!

Hint: Remember back in Exercise 1, I asked you to figure out the name "TF2" given that the name of the 1-dimensional function class was "TF1"? Well, the name of the one-dimensional histogram class is "TH1D", so what do you think the name of the two-dimensional histogram class is? Check your guess on the ROOT web site.

---

<sup>13</sup> A tangent: Suppose you're told to fill two histograms, then add them together. If you do this, you'll want to call the "Sumw2" method of both histograms before you fill them; e.g.,

```
TH1* hist1 = new TH1D(...);
TH1* hist2 = new TH1D(...);
hist1->Sumw2();
hist2->Sumw2();
// Fill your histograms, then to add hist2 to the contents of hist1:
hist1->Add(hist2);
```

If you forget the "Sumw2", then your error bars after the math operation won't be correct. General rule: If you're going to perform histogram arithmetic, use "Sumw2" (which means "sum the squares of the weights"). Some physicists use "Sumw2" all the time, just in case.

## Calculating our own variables (10 minutes)

There are other quantities that we may be interested in apart from the ones already present in the n-tuple. One such quantity is  $p_T$  which is defined by:

$$p_T = \sqrt{p_x^2 + p_y^2}$$

This is the transverse momentum of the particle, that is, the component of the particle's momentum that's perpendicular to the z-axis.

Let's calculate our own values in an analysis macro. Start fresh by copying our AnalyzeComments example again:

```
> cp AnalyzeComments.C AnalyzeVariables.C
```

In the Loop section, put in the following line:

```
Double_t pt = TMath::Sqrt(px*px + py*py);
```

What does this mean?

Whenever you create a new variable in C++, you must say what type of thing it is. Actually, we've already done this in statements like

```
TF1 func("user", "gaus(0)+gaus(3)")
```

This statement creates a brand-new variable named "func", with a type of "TF1". In the Loop section of AnalyzeVariables, we're creating a new variable named "pt", and its type is "Double\_t".

For the purpose of the analyses that you're likely to do, there are only a few types of numeric variables that you'll have to know: "Float\_t"; which is used for real numbers, "Double\_t" which is used for double-precision real numbers; "Int\_t", which is used for integers; "Bool\_t" for boolean (true/false) values. "Long64\_t" specifies 64-bit integers, which you probably won't need for your work. Most physicists use double precision, just in case.

If you already know C++: the reason why we don't just use the built-in types "float", "double", "int", and "bool" is discussed on pages 18-19 of the ROOT Users Guide.

ROOT comes with a very complete set of math functions. You can browse them all by looking at the TMath class on the ROOT web site, or Chapter 13 in the ROOT User's Guide. For now, it's enough to know that TMath::Sqrt() computes the square root of the expression within the parenthesis "()".

Test the macro in AnalyzeVariables to make sure it runs. You won't see any output, but we'll fix that in the next exercise.

### **Exercise 6 (10 minutes)**

Revise AnalyzeVariables.C to make a histogram of the variable "pt". Don't forget to label the axes; remember that the momenta are in  $GeV$ .

If you want to figure out what the bin limits of the histogram should be, I'll permit you to "cheat" and use the following command interactively:

```
tree1->Draw("sqrt(px*px + py*py)")
```

### **Exercise 7 (15 minutes)**

The quantity "theta", or the angle that the beam makes with the  $z$ -axis, is calculated by:

$$\theta = \arctan\left(\frac{p_T}{p_z}\right)$$

The units are radians. Revise AnalyzeVariables.C to include a histogram of theta.

I'll make your life a little easier: the math function you want is `TMath::ATan2(y,x)`, which computes the arctangent of  $y/x$ . It's better to use this function than `TMath::ATan(y/x)`, because the `ATan2` function correctly handles the case when  $x=0$ .

## Applying a cut (10 minutes)

The last "trick" you need to learn is how to apply a cut in an analysis macro. Once you've absorbed this, you'll know enough about ROOT to start using it for a real physics analysis.

The simplest way to apply a cut in C++ is to use the "if" statement. This is described in every introductory C and C++ text, and I won't go into detail here. Instead I'll provide an example to get you started.

Once again, let's start with a fresh Analyze macro:

```
> cp AnalyzeComments.C AnalyzeCuts.C
```

Our goal is to count the number of events for which  $p_z$  is less than 145 *GeV*. Since we're going to count the events, we're going to need a counter. Put the following in the Set-up section of AnalyzeCuts.C:

```
Int_t pzCount = 0;
```

Why "Int\_t" and not "Long64\_t"? Well, I find that "Int\_t" is easier to remember. I could even 'cheat' and just use "int", which will work for this example. You would only have to use the type Long64\_t if you were counting more than  $2^{31}$  entries. I promise you that there aren't that many entries in this file!

For every event that passes the cut, we want to add one to the count. Put the following in the Loop section:

```
if ( pz < 145 )
{
    pzCount = pzCount + 1; // you could use "pzCount++;" instead
}
```

Be careful: it's important that you surround the logical expression "pz < 145" with parentheses "()", but the "if-clause" must use curly brackets "{}".

Now we have to display the value. Again, I'm going to defer a complete description of formatting text output to a C++ textbook, and simply supply the following statement for your Wrap-up section:

```
std::cout << "The number of events with pz < 145 is "
<< pzCount << std::endl;
```

When I run this macro, I get the following output:

```
The number of events with pz < 145 is 14962
```

Hopefully you'll get the same answer.

## Exercise 8 (15 minutes)

Go back and run the macro you created in Exercise 5. If you've overwritten it, you can copy my version:

```
> cp ~seligman/root-class/AnalyzeExercise5.C $PWD
```

The chi2 distribution and the scatterplot hint that something interesting may be going on.

The histogram, whose limits I originally got from the command `tree1->Draw("chi2")`, looks unusual: there's a peak around 1, but the x-axis extends far beyond that, up to `chi2 > 18`. Evidently there are some events with a large chi2, but not enough of them to show up on the plot.

On the scatterplot, we can see a dark band that represents the main peak of the chi2 distribution, and a scattering of dots that represents a group of events with anomalously high chi2.

The chi2 represents a confidence level in reconstructing the particle's trajectory. If the chi2 is high, the trajectory reconstruction was poor. It would be acceptable to apply a cut of `"chi2 < 1.5"`, but let's see if we can correlate a large chi2 with anything else.

Make a scatterplot of "chi2" versus "theta". It's easiest if you just copy the relevant lines from your code in Exercise 7; again, there's a file `AnalyzeExercise7.C` in my area if that will help.

Take a careful look at the scatterplot. It looks like all the large-chi2 values are found in the region `theta > 0.15` radians. It may be that our trajectory-finding code has a problem with large angles. Let's put in both a theta cut and a chi2 cut to be certain we're looking at a sample of events with good reconstructed trajectories.

Use an "if" statement to only fill your histograms if `chi2 < 1.5` and `theta < 0.15`. Change the bin limits of your histograms to reflect these cuts; for example, there's no point to putting bins above 1.5 in your chi2 histograms since you know there won't be any events in those bins.

It may help to remember that the symbol for logical AND in C++ is `"&&"`.

A tip for the future: in a real analysis, you'd probably have to make plots of your results both before and after cuts. A physicist usually wants to see the effects of cuts on their data.

I must confess: I cheated when I pointed you directly to theta as the cause of the high-chi2 events. I knew this because I wrote the program that created the tree. If you want to look at this program yourself, go to the UNIX window and type:

```
> less ~seligman/root-class/CreateTree.C
```

### **Exercise 9 (15 minutes)**

Assuming a relativistic particle, the measured energy of the particle in our example n-tuple is given by

$$E_{meas}^2 = p_x^2 + p_y^2 + p_z^2$$

and the energy lost by the particle is given by

$$E_{loss} = E_{beam} - E_{meas}$$

Create a new analysis macro (or revise one of the ones you've got) to make a scatterplot of  $E_{loss}$  vs. "zv". Is there a relationship between the z-distance traveled in the target and the amount of energy lost?

### **Exercise 10 (10 minutes)**

In all the analysis macros we've worked with, we've drawn any plots in the Wrap-up section. Pick one of your analysis macros that creates histograms, and revise it so that it does not draw the histograms on the screen, but writes them to a file instead. Make sure that you don't try to write the histograms to "experiment.root"; write them to a different file named "analysis.root". When you're done, open "analysis.root" in ROOT and check that your plots are what you expect.

Hints:

In "Saving your work, part 2" on page 18, I described all the commands you're likely to need.

Don't forget to use the ROOT web site as a reference. Here's a question that's also a bit of a hint: What would be the difference between opening your new file with "UPDATE" access, "RECREATE" access, and "NEW" access? Why might it be a bad idea to open a file with "NEW" access? (A hint within a hint: what would happen if you ran your macro twice?)

## ***Exercise 11: Publishing your work (10 minutes)***

Let's create a sample histogram:

```
[ ] TH1D h1("hist", "My Final Results", 100, -3, 3)
[ ] h1.FillRandom("gaus", 100000)
[ ] h1.Draw("e1")
```

Your task is to make this plot neat enough for publication. Some things to note:

- Don't forget to label your axes. How about "x [arbitrary units]" for the x-axis, and "f(x) [arbitrary units]" for the y-axis.
- The x-axis label looks OK, but that y-axis label overlaps the numbers on the axis. You'll have to figure out how to move the axis, the label, or both. Hint: "SetTitleOffset".
- Print out the graph to check. Does anything look wrong?
- Do any of ROOT's pre-defined styles do the same thing?

I did this in front of you at the start of the class. You will have to do it on your own in nine weeks, as you prepare your final talk or paper. The point of this exercise, as you've probably guessed, is to have you figure out how to do this using the tools and techniques you've learned so far. Hopefully, you'll still remember how to do this at the end of the summer.

## ***Exercise 12: Stand-alone program (optional) (between 15–60 minutes depending on whether you've programmed in C++ before)***

Why would you ever want to write a stand-alone program instead of using ROOT interactively? First of all, compiled code executes faster; maybe you've already learned about the techniques described on page 93 of the ROOT User's Guide. Also, stand-alone programs are easier to submit to batch systems that run in the background while you do something else.

I'll be honest with you: I'm spending all this time to teach you about interactive ROOT, but I never use it. I can develop code faster in a stand-alone program, without restarting ROOT or dealing with a puzzling error message that refers to the wrong line in a macro.

If it's near the end of the first day, don't bother to start this exercise. But if you still have an hour or more before the end of the class -- well, then you're pretty good. This exercise is a bit of a challenge for you.

So far, you've used ROOT interactively to perform all the exercises. Your task now is to write a stand-alone program that uses ROOT. Start with the macro you created in Exercise 10: you have a ROOT script (a ".C" file) that reads an n-tuple, performs a calculation, and writes a plot to a file. Create, compile, run, and test a C++ program (a ".cc" file) that does the same thing.

You can't just take Analyze.C, copy it to Analyze.cc, and hope it will compile. For one thing, Analyze.C does not have a "main" routine; you will have to write one. Also, C++ doesn't know about the ROOT classes; you have to find a way to include the classes in your program. There are links on this page that may help you:

<<http://www.nevis.columbia.edu/~seligman/root-class/links.html>>

When you try to compile the program, the following simple attempt won't work:

```
> g++ Analyze.cc -o Analyze
```

You will have to add flags to the g++ command that will refer to the ROOT header files and the ROOT libraries. You can save yourself some time by using the root-config command. Take a look at the **man** page for this command:

```
> man $ROOTSYS/man/man1/root-config.1
```

Try it:

```
> root-config --cflags
```

```
> root-config --libs
```

If only there were a way of getting all that text into your compilation command without typing it all over again. This is where the UNIX "backtick" comes in handy. Try:

```
> g++ Analyze.cc -o Analyze `root-config --cflags`
```

Be careful as you type this; it's not the usual single quote (') but the backtick (`), which is typically located in the upper left-hand corner of a computer keyboard.

Are things still not working? Maybe I want you to think about adding more than one argument to a single command.

That's enough hints.

## ***Playing with TreeViewer (very definitely optional)***

Start up a fresh ROOT session, open experiment.root with the TBrowser, and navigate to tree1. Right-click on the tree1 icon, and select “StartViewer”.

You’re looking at the TreeViewer, a tool for making plots from n-tuples interactively. Everything that you’ve done with experiment.root (from page 20 on) can be done more quickly with the TreeViewer.

Then why did I make you do everything the hard way? Because all of the above exercises are much simpler than what you’ll do when you get to a real physics analysis. The TreeViewer is handy for quick, superficial studies of n-tuples, but it’s almost certainly not enough to get you through the work you’ll have to do this summer. Any serious analysis work will involve editing ROOT macros and writing C++ code.

Still, there are times when a simple tool can be useful. So I offer the following optional exercise... but it treat it as “playtime,” not serious work.

### ***Exercise 13: Very optionally playing with TreeViewer***

Using the TreeViewer, repeat all of exercises 2 through 9.

This may sound like a lot of work, but once you get the hang of it, it takes less than ten minutes. Use the Help menu on the TreeViewer to learn how to use it.

## References

You've learned about a few ROOT resources so far:

- the ROOT web site
- the ROOT manual
- the Help menu located in the upper right-hand corner of most ROOT windows
- create something “by hand,” save it as a .C file, then examine the file to see how ROOT does it

There's one other resource: the example ROOT programs that come with the package. You'll find them in \$ROOTSYS/tutorials. When I ask the question, “How do I do something complicated in ROOT?”, I often find the answer in one of the examples they provide.

I've found it handy to make my own copy:<sup>14</sup>

```
> cp -arv $ROOTSYS/tutorials $PWD
```

Then I go into the “tutorials” sub-directory, run their examples, and look at their code:

```
> cd tutorials
> root -l demos.C
> cd graphics
> root -l first.C
> less first.C
```

You're going to need these resources as you move into the following topics for Parts Three and Four of the tutorial. I'm going to do less “hand holding” in these notes from now on, because a part of these exercises to teach you how to use these references.<sup>15</sup>

If the distributed nature of the information is annoying to you, welcome to the club! I often have to go hunting to find the answers I want when using ROOT, even after years of working with the package. Occasionally I've had no other choice but to examine the C++ source code of the ROOT program itself to find out the answer to a question.

---

<sup>14</sup> If the command doesn't work: Did you remember to type “setup root” in your UNIX command window? That's what sets the value of \$ROOTSYS.

<sup>15</sup> You can still ask me questions; I mean that any remaining written hints in this tutorial will be less detailed or require more thought.

## Part Three

If you still haven't finished the exercises for Parts One and Two, then keep working on them. The following exercises are relevant to larger-scale analyses, but may not be relevant to the work that you'll be asked to do this summer.

Before we get to those exercises, let's consider some more advanced topics in ROOT and C++.

### ***Working with folders inside ROOT files***

As you worked with the TBrowser, you may have realized that ROOT organizes its internal resources in the form of "folders," which are conceptually similar to the hierarchy of directories on a disk. You can also have folders within a single ROOT file.

Folders are discussed in Chapter 10 in the ROOT Users Guide, but I have not seen the approach they describe (the TTask class) used in any experiment on which I've worked. Instead I'll focus on ROOT folders in the way they're more often used (if they're used at all): to organize objects within a file.

Copy the file `folders.root` from my `root-class` directory into your own, and use the ROOT TBrowser to examine its contents.

You'll see three folders within the file: `example1`, `example2`, and `example3`. Each of these folders will be the basis of the next three exercises.

All three exercises will require you to make a plot of data points with error bars. You'll want to use the TGraphErrors class for this.

## ***A little bit more about inheritance***

Look up the TGraphErrors class on the ROOT web site. After you've looked over what little help there is on this page, click on the Source tab to see an example of how the class is used. At the top of the page it says "TGraphErrors: public TGraph"; click on the TGraph link to find out where to look for options when drawing the graph.

The "public TGraph" part means that the class TGraphErrors is inherited from the TGraph class. If you don't know about class inheritance in C++ yet, don't worry. For now, it's enough to know that if class "Derived" inherits from class "Base", class Derived will have all the methods and properties of class Base, plus some additional features.<sup>16</sup>

Here's a simple example: If you saw "Circle: public Shape" in the ROOT reference guide, it would mean that the class Circle derives from the class Shape. If Shape had a Draw() method, then Circle would have a Draw() method as well.

A more relevant example: TGraphErrors inherits from TGraph. Since TGraph has a Draw() method, so does TGraphErrors; TGraph also has Fit() method, a GetMean() method, a Print() method, so therefore TGraphErrors has all those methods too.

However, TGraphErrors also has a SetPointError() method, which TGraph does not have. (To continue the above analogy, a Circle class might have a GetRadius() method that a more general Shape class would not have.)

You may have noticed that TGraph, in turn, inherits from Named and some other classes. Click on the Named link; you'll see that Named inherits from TObject. Does TObject inherit from any other class?

---

<sup>16</sup> For those familiar with the issues of public inheritance: yes, I'm skipping over a lot of details, such as the distinction between virtual versus non-virtual methods.

## Container classes

Go back to the description of the TGraphErrors class. Note that to create a TGraphErrors object, you need to supply some arguments:

These are all different ways to construct a plot with error bars:

- TGraphErrors() – This is used internally by ROOT when reading a TGraphErrors object from a file. You won't use this method directly.
- TGraphErrors(Int\_t n) – You use this when you just want to supply TGraphErrors with the number of points that will be in the graph, then use the SetPoint() and SetPointError() methods to assign values and errors to the points.
- TGraphErrors(const TGraphErrors& gr) – This is called a “copy constructor” in C++, and is used when you copy a TGraphErrors object. You can ignore this.
- TGraphErrors(const TH1\* h) – You use this to create a TGraphErrors plot based on values in a histogram.

Now that I've give you a guide to the first four ways to construct a TGraphErrors object, you can probably figure out what the next five are: to create one from the contents of a file, and to create plots from either float or double-precision... somethings.

Those somethings are containers. In ROOT and C++, there are three general categories of containers you have to know about.

## Arrays

Do a web search on “C++ arrays” to learn about these containers. Briefly, to create a double-precision array of eight elements, you could say:

```
Double_t myArray[8];
```

To refer to the 3<sup>rd</sup> element in the array, you might use (remember, in C++ the first element has an index of 0):

```
Int_t i = 2;  
myArray[i] = 0.05;
```

If you're new to C++, it won't be obvious that while myArray[2] is a Double\_t object, the type of the name myArray (without any index) is Double\_t\*, or a pointer to a Double\_t (see page 22).

Getting confused? Let's keep it simple. If you've created arrays with values and errors...

```
Double_t xValue[22];  
Double_t xError[22];  
Double_t yValue[22];  
Double_t yError[22];
```

...and you've put numbers into those arrays, then you can create a TGraphErrors with:

```
TGraphErrors* myPlot = new TGraphErrors(22, xValue, yValue, xError, yError);
```

## Arrays (continued)

Did you notice a problem with that example? I had to supply a fixed value for the number of points in each array to make the plot. In general, you won't be able to do that; in fact, in exercises 15 and 16 below you *can't* do that.

In C++, one way to get around this problem is to use "dynamic arrays." I'll let you read about those on the web (search on "C++ dynamic arrays"), but I'm not going to say more about them, because I rarely use them.

## ROOT's containers

Go to the Class Index page of the ROOT Reference Guide on the web. Near the top of the page there's a list of categories; click on CORE, then on CONT. You'll see a list of ROOT's container classes, along with links for information about collections and why they're used. Read the "Understanding Collections" page, and at least skim the chapter about collections in the ROOT Users Guide.

I'll be blunt here, and perhaps editorialize too much: I don't like ROOT's collection classes. The main reason is that most of them can only hold pointers to classes that inherit from TObject. For example, if you wanted to create a TList that held strings or double-precision numbers (TString and Double\_t in ROOT), you can't do it.

Go back to the TGraphErrors page. The seventh way to create a TGraphErrors object has a TVectorD link; click on that link to read the description. Learn much? Try clicking on TVectorT<double>.

This is ROOT's answer to the issue I raised above: they provide special containers for certain types.

You need to know a little about ROOT's collection classes to be able to understand how ROOT works with collections of objects; exercise 16 below is an example of this. For any other work, I'm going to suggest something else:

## Standard Template Library (STL)

Do a web search on "standard template library". This will probably take you to SGI's web site at first. Skim a few sites, especially those that contain the words "introduction" or "tutorial". You don't have to get too in-depth; for example, you probably don't have enough time today to fully understand the concept of iterators.

Did you guess that STL is my preferred method of using containers in C++?

## Vectors

The Standard Template Library is an important development in the C++ programming language. It ties into the concepts of design patterns and generic programming, and you can spend a lifetime learning them.<sup>17</sup>

For the work that you'll be asked to do in Parts Three and Four, and quite probably for the rest of this summer, there's only one STL class you'd have to understand: vectors. Here are the basics:

If you want to use vectors in a program, or even a ROOT macro, you have to put the following near the top of your code:

```
#include <vector>
```

To create a vector that will contain a certain type, e.g., double-precision values:

```
std::vector<Double_t> myVector;
```

If you want to create a vector with a fixed number of elements, e.g., 8:

```
std::vector<Double_t> myOtherVector(8);
```

To refer to a specific element of a vector, use the same notation that you use for C++ arrays:

```
myOtherVector[2] = 0.05;
```

To append a value to the end of the vector, which will make the vector one element longer, use the `push_back()` method:

```
myVector.push_back( 0.015 );
```

To find out the current length of a vector, use the `size()` method:

```
Int_t length = myVector.size();
```

Here's a simple code fragment that loops over the elements of a vector and prints them out.

```
for ( Int_t i = 0; i != someVector.size(); ++i )
{
    std::cout << "The value of element " << i
              << " is " << someVector[i] << std::endl;
}
```

You have a vector, but TGraphErrors wants a C++ array name. Here's the trick:

```
// Define four vectors.
std::vector<Double_t> x,y,ex,ey;
// Put values in the vectors (omitted so you can do it!)
Int_t n = x.size();
TGraphErrors* plot = new TGraphErrors(n, &x[0], &y[0], &ex[0], &ey[0]);
```

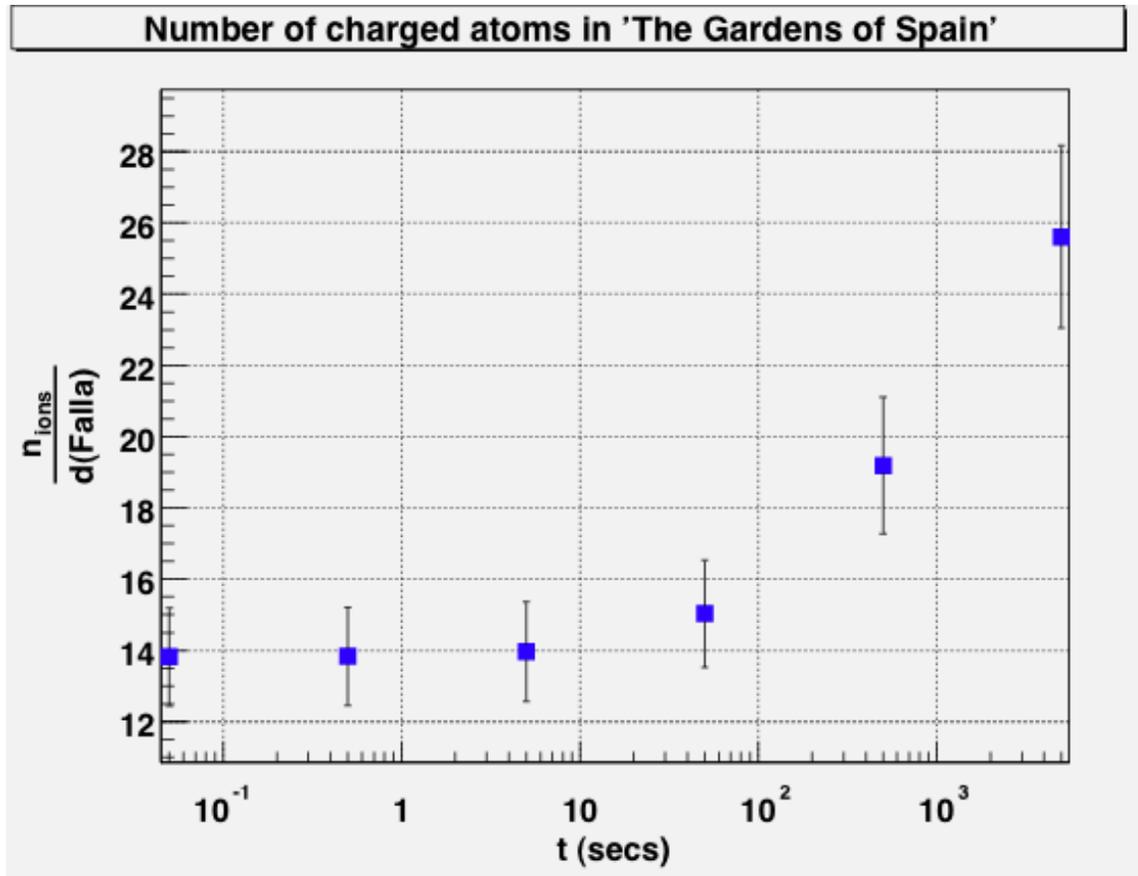
In other words, if `v` has the type `std::vector<Double_t>`, then `&v[0]` is a `Double_t*` (a pointer to `Double_t`) that's equivalent to an array name.

---

<sup>17</sup> I've lost track of the number of your lifetimes I've spent. You're probably tired of the joke anyway.

### Exercise 14: Create a basic x-y plot (1-2.5 hours)

You're going to re-create that "pun plot" that I showed during my initial talk:



Use the histograms in folder example1 file folders.root. The y-values and error bars will come from fitting each histogram to a gaussian distribution; the y-value is the mean of the gaussian, and the y-error is the width of the gaussian.

You've spent five pages reading about abstract concepts and are probably eager to do some work, but there's still a couple of things you'll have to figure out.

First of all, there's no n-tuple in this exercise. You'll have to create a ROOT macro to create the graph on your own.<sup>18</sup> You've seen some macros before (remember c1.C?), and you'll find many more in the ROOT tutorials.

(continued on next page)

<sup>18</sup> You could try typing in the ROOT commands on the ROOT command line one-by-one. But unless you have a shining grasp of ROOT concepts and perfect typing skills, you're going to make mistakes that will involve many quit-and-restarts of ROOT. It's much easier to write and edit a macro.

Want to see more examples of using TGraphErrors? Look at the ROOT tutorials directory. The problem is that there are lots of examples; how do you find those that use TGraphErrors? I copied the ROOT tutorials directory (see page 42), and then I used the UNIX grep command:

```
> cd tutorials
> grep -rl TGraphErrors *
```

This will list the names of the files that contain the text “TGraphErrors”. That’s how I found out how to draw a TGraphErrors plot inside a ROOT canvas.

The UNIX grep command is very useful; type “man grep” to learn about it.<sup>19</sup>

You need to figure out how to get the x-values. In this case, it’s relatively simple. There are only six histograms in the example1 folder. In TBrowser, double-click on the histograms and read the titles. The histograms are numbered from hist0 to hist5; so you can derive the formula to go from the histogram index to the value of x.

You already know how to open a ROOT file within a macro (it was part of exercise 10 on page 38), but it’s not obvious how to “navigate” to a particular folder within a file. Look at the description of the TFile class on the ROOT web site. Is there a method that looks like it might get a directory?

By now, you’ve probably gotten the idea that for ROOT to know where to look to plot, read, or write something, it has to know where to “focus.” If an object requires focus in some way, it will have a cd() method (short for “change directory”). Based on that hint, and what you can see on the TFile web page, something like this might work:

```
TDirectory* example1 = inputFile->GetDirectory("example1");
example1->cd();
```

The histograms are number 0 to 5 consecutively. It would be nice to write a loop to read in “hist0”, “hist1”, ... “hist5” and fit each one. But to do that, you have to somehow convert a numeric value to a text string.

If you know C or C++, you already know ways to do this. If all this is new to you, here’s one way to do it:

(continued on next page)

---

<sup>19</sup> Optional tangent:

“Grep” is a program that implements something called “regular expressions,” a powerful method for searching, replacing, and processing text. More sophisticated programs that use regular expressions include sed, awk, and perl. They are used in processing text, not numerical calculations, so the deep nitty-gritty of regular expressions is rarely relevant in physics.

Regular expressions are a complex topic, and it can take a lifetime to learn about them. (You may be tired of the joke, but I’m not!)

```

#include <sstream> // put this near the top of your macro
for ( Int_t i = 0; i != 6; ++i )
{
    std::ostringstream os;
    os << "hist" << i;
    TString histogramName = os.str();
    // ... do what you need to with histogramName
}

```

There are other problems you'll have to solve:

- How do you read a histogram from a file? Or the more general question is: How do you get a ROOT object from a file?  
Hint: How do you “get” an object from a TFile? (Once you’ve figured this out, you might want to use the tip about GetObject. Look through the tutorial files for more clues.)
- Once you fit a histogram to a gaussian distribution, how do you get the mean and width of the gaussian from the fit?  
Hint: Look through the histogram methods; click through the links to other classes. The TH1D page doesn’t list the method you’ll need, but the TH1 page does.
- In the above plot, the x-axis is logarithmic. How do you make that change?  
Hint: Remember how you found out how to label an axis?
- Speaking of axis labels, how do you put in  $\frac{n_{ions}}{d(Falla)}$  ?  
Hint: look up TLatex in the ROOT web site. You don’t have to declare a TLatex object; just put the text codes into the axis label and ROOT will interpret them.<sup>20</sup>
- How do you get the marker shapes and colors as shown in the plot?  
Some looking around the ROOT web site should give you the answer.

I’ll be frank: if this class is your first exposure to C++, then this exercise is *hard*. Don’t feel bound by the suggested time of 2.5 hours. Instead, use the references to learn enough about C++ to try to get this exercise done by the end of the workshop.

---

<sup>20</sup> Another tangent:

LaTeX is a document-preparation package that’s often used in research. If you write a paper for publication this summer, you are going to use LaTeX; physics publications don’t accept articles in MS-Office format. A real LaTeX document is much more complex than you’ll read about in the TLatex documentation, but don’t worry about that. No one writes a LaTeX document from scratch; they get one from someone and learn by example. It’s much easier than learning ROOT; it’s closer to another page mark-up language, HTML, which you’ve probably seen before.

You can spend a lifetime learning LaTeX, but no one ever has.

## ***Exercise 15: A more realistic example of an x-y plotting task (1-2 hours)***

Take a look at folder `example2` in `folders.root`. You'll see a bunch of histograms, and an n-tuple named `histogramList`. Right-click on `histogramList` and Scan the n-tuple. On the ROOT text window, you'll see that the n-tuple is a list of histogram ID numbers and an associated value.

Once again, you're going to fit all those histograms to a gaussian and make an x-y plot. The y values and error bars will come from the fits, as in the previous exercise. The x values will come from the n-tuple; for example, the value of x for histogram ID 14 is 1.0122363.

I'll let you pick the axis labels for this graph; don't make the x-axis logarithmic.

You've probably already figured out that you can use `MakeClass` on the `histogramList` n-tuple, just like you did on page 25. The challenge will be in putting together the code inside the `Loop` method of the class you made with bits and pieces from the previous exercise.

In the previous exercise, perhaps you hard-coded the number of histograms in the folder. Don't do that here. You could get the number of histograms from the number of entries in the n-tuple.

Or maybe that's not a good idea; what if there were an entry in the n-tuple but no corresponding histogram? Keep a separate count of the number of "valid" histograms you're able to read. This means you'll have to check if you've read each histogram correctly. Tip: If a ROOT operation to read a pointer fails, that pointer will be set to zero.

## Part Four

### ***Exercise 16: A brutally realistic example of a plotting task (1-2 hours)***

Now take a look at folder `example3`. You probably already looked in there, and were overwhelmed with the number of histograms.

Here's the task: it's another x-y plot, with the y values and error bars from fitting the histograms. You only want to include those histograms whose names begin with "plotAfterCuts"; the other histograms you can ignore.

The x values come from the histograms themselves. Double-click on a few histograms to plot them. You'll see that the x values are in the titles (not the names!) of the histograms.

You'll be able to re-use the code you developed for the previous two exercises. There are some new problems to solve: how to get the list of all the histograms in the `example3` folder, how to test if a histogram's name begins with "plotAfterCuts", and how to convert a histogram's title from string form to a number.

Let's think about the easier problems first.

If you're fairly familiar with C or C++, you probably already know how to convert strings into numbers. If you're not, then I suggest you take a look at the description of the `TString` class on the ROOT web site; the `Atof()` method may have some interesting comments.

The `TString` class is pretty good about converting string formats implicitly.<sup>21</sup> You probably already figured out how to look up getting the title from a histogram. The method returns "const char \*" but something like this will work:

```
TString title = histogram->GetTitle();
```

What about testing if the text in a `TString` begins with "plotAfterCuts"? Take another look at the `TString` web page. Is there a method that looks like it might help you with that test?

(continued on next page)

---

<sup>21</sup> Optional digression: There are three main ways of handling strings in ROOT/C++:

- The original way in the older language C, as an array of char: `char oldStyleString[256];`
- A newer way, added to the C++ language: `std::string newStyleString;`
- The ROOT way: `TString rootStyleString;`

Which is better? My attitude is that none of them is best. In a ROOT program, I tend to use `TString`; if my program doesn't use ROOT, I use `std::string` for string variables and arrays of char for constant strings.

The blunt reality is that C++ doesn't have the built-in text manipulation facilities of languages like perl or python. This can be important in a physics analysis procedure; while your calculations are based on numbers, manipulating files or program arguments can be based on strings.

## Exercise 16 (continued)

The next problem is trickier: How do you get a list of objects in a directory?

By now you've got the hang of the above hint: I want to "Get" a "List" of objects in a directory. When I worked on this problem, I went to the TFile web page and looked for methods with names that began with "GetList". I clicked on the name of the method... then pounded my head against the desk.<sup>22</sup>

I finally got the answer by using the UNIX grep command to search through the ROOT tutorials directory for the text "GetList". There are many files there with a "GetList..." call, but one file name stood out for me, and I looked at it first. Since I had read the TList web page first, I could see that the answer was there. But it's written in a sloppy fashion that you'd have to change.

To understand what you'd have to change, consider a chain of class inheritance that's similar to what you looked at on page 43:

Go to the web page for class TH1D. It inherits from two classes, TH1 and TArrayD; ignore the latter for now. Click on the TH1 link to see from which classes TH1 inherits. Eventually you'll work your way up to the class TObject.

Go back to the TH1D page, and search for the "Class Charts" section. You'll see a diagram of the inheritance; you'll even note that there's another class, TProfile, which inherits from TH1D.

In C++, the practical aspect of class inheritance is that you can use a pointer to a base class to refer to a derived class object; if class Derived inherits from class Base, you can do this:

```
Base* basePointer = new Derived();
```

If that's a little abstract for you, consider this in terms of the classes with which you've been working. Any of the following is correct in C++:

```
TH1D* doublePrecisionHistogram = new TH1D(...);
```

```
TH1* histogram = new TH1D(...);
```

```
TObject* genericRootObject = new TH1D(...);
```

Why does this matter? Because *ROOT does not read or write histograms, functions, n-tuples, nor any other specific object. ROOT reads and writes pointers to class TObject.* After you read in a TObject\*, you'll probably want to convert it to a pointer to something useful.

---

<sup>22</sup> Optional editorializing again: If you followed the steps I just described and saw the same thing I did, it's pretty clear what happened: the person who wrote the method intended to supply some comments later.

Here's a tip for writing code that will make you a hero: the word "later" does not exist. Treat the comments as part of the code-writing process. If you have to edit the code, edit the comments.

Yes, I know it's a pain. But pounding your head on a desk is a bigger pain. It's the biggest pain of all when you realize that you wrote the code six months ago, have completely forgotten what it means, and must now spend an hour figuring it out. It would have taken five seconds to write a comment.

## Exercise 16 (continued)

In C++, the simplest way to attempt to convert a base class pointer to a derived class pointer something like this (assuming `genericRootObject` is a `TObject*`):

```
TH1* histogram = (TH1*) genericRootObject;
If ( histogram == 0 )
{
    // The genericRootObject was not a TH1*
}
else
{
    // The genericRootObject was a TH1*; you can use it for things like:
    histogram->FillRandom("gaus",10000);
    histogram->Draw();
}
```

If I didn't put that test in there and just tried `histogram->FillRandom("gaus",10000)`, and `histogram==0`, then the program would crash with a segmentation fault.<sup>23</sup>

Why did I just take one-and-half pages to go over such a dry topic?

- Understanding object inheritance makes it clear why the macros that ROOT automatically creates for you use pointers, why those container classes you saw on page 46 only contain `TObject*`, and other aspects of ROOT.
- It's so when you see a line like this in the ROOT tutorials, you have some idea what it's doing: it's using a `TKey` to read in a `TObject*`, then converting it to a `TH1F*`:

```
h = (TH1F*)key->ReadObj();
```

Now you should have an idea of how to edit this line to do what you want to do... and how to check if what you've read is actually a histogram, or some other object that was placed inside that folder.

---

<sup>23</sup> If you haven't encountered a segmentation fault yet in this tutorial, you're either very lucky or very good at managing your pointers. Now you know why it happens: someone tried to call a method for an object that wasn't there.

## **Exercise 17: Data reduction (1–2 hours)**

Up until now, we've considered n-tuples and histograms that someone else has created for you. The process by which a file that contains complex data structures is converted into a relatively simple n-tuple is called "data reduction." It's usually a necessary step in the overall physics analysis chain.

As I implied in the first day of this tutorial, perhaps you'll be given an n-tuple and told to work with it. However, it's just as likely that you'll be given a file containing the next-to-last step in the analysis chain: a file with C++ object with data structures. You'd want to extract data from those structures to create your own n-tuples.

Copy the following files from my `root-class` directory:

```
ExampleEvent.h
exampleEvents.root
libExampleEvent.so
```

The file `exampleEvents.root` contains a ROOT tree of C++ objects. The task is to take the event information in those C++ objects and reduce it to a relatively simple n-tuple.

First, take a look at `ExampleEvent.h`. You're not going to edit this file. It's the file that someone else used to create the events in the ROOT tree. If you're given an `ExampleEvents` object, you can use any of the methods you see to access information in that object; for example:

```
ExampleEvent* exampleEvent;
// Assume we fill in exampleEvent somehow.
Int_t numberLeptons = exampleEvent->GetNumberLeptons();
```

For this hypothetical analysis, you've been told that the following information is to be put into the n-tuple you're going to create:

- the run number;
- the event number;
- the total energy of all the particles in the event;
- the total number of particles in the event.
- a boolean indicator: does the event have only one muon?
- the total energy of all the muons in the event;
- the number of muons in the event;

The task is to write the code to read the events in `exampleEvents.root` and write an n-tuple to a different file, `exampleNtuple.root`.

(continued on the next page)

## Exercise 17 (continued)

After what you've done before, your first inclination may be to open `exampleEvents.root` directly in ROOT and look at it with the TBrowser. Go ahead and try it.

It doesn't fail, but you'll get an error message about not being able to find a dictionary for some portions of the `ExampleEvent` class.<sup>24</sup> I mentioned this earlier in footnote 10 on page 20: it's possible to extend ROOT's list of classes with your own by creating a custom dictionary. Only classes that have a dictionary can be fully displayed using the ROOT browser.

Try to see how much of the `ExampleEvent` tree you can see without the dictionary. Then restart ROOT and type the following ROOT command:

```
[ ] gSystem->Load("libExampleEvent.so");
```

This causes ROOT to load in the code for a dictionary that I've pre-compiled for you.<sup>25</sup> Now you can open the `exampleEvents.root` using a `TFile` object and use the ROOT browser to navigate through the `ExampleEvent` objects stored in the tree.

As you look at the file, you'll see that there's a hierarchy of objects. There's only one object in the file, `exampleEventsTree`. Inside that tree, there is only one "branch", `exampleEventsBranch`.

That's a bit of a clue: a ROOT n-tuple is actually a `TTree` object with one `Branch` for every simple variable.

(continued on the next page)

---

<sup>24</sup> If you didn't get such a message, then you probably copied `rootlogon.C` from my `root-class` directory in your working directory. That's OK, but you might want to temporarily rename this file and restart ROOT just so you can see that error message. That way you'll know how it looks if you have a missing-dictionary problem.

<sup>25</sup> This library may not work if you're on a different kind of system than the one on which I created the library. If you get some kind of load error, here's what to do. Copy the following additional files from my `root-class` directory:

```
LinkDef.h
ExampleEvent.cxx
BuildExampleEvent.cxx
BuildExampleEvent.sh
```

Run the UNIX command script with:

```
> sh BuildExampleEvent.sh
```

This will (re-)create the `libExampleEvent` shared library. It will also create the program `BuildExampleEvent`, which I used to create the file `exampleEvent.root`.

If you're running this on a Macintosh, the name of the library will be `libExampleEvent.dylib`; that's the name to use in the `gSystem->Load()` command in the Mac version of ROOT.

## Exercise 17 (continued)

At this point, you could actually use `MakeClass()` to create a ROOT macro for you, but I suggest that you only do this to get some useful code fragments to copy into your own macro.<sup>26</sup>

I'll offer you the following additional hints:

- The first line of your ROOT macro for this exercise is likely to be the library load command on the previous page.
- If you're writing a stand-alone program, instead of loading the library you'll have the line:

```
#include "ExampleEvent.h"
```

and include `libExampleEvent.so` on the line you use to compile your code.

- Look at the examples in the `tutorials/tree` directory, on the TTree web page, and in the macro you created with `MakeClass` (if you chose to do so).
- Yes, the ampersands are important!

---

<sup>26</sup> Why don't I want you to use `MakeClass` here? The answer is that some physics experiments only use ROOT to make n-tuples, they don't use it for their more complex C++ classes. In that case, you won't be able to use `MakeClass` because you won't have a ROOT dictionary. It's likely that such a physics experiment would have its own I/O methods that you'd use to read its physics classes, but you'd still use a ROOT TTree and branches to write your n-tuple.

## ***Wrap-up***

The last four exercises are difficult. I chose those tasks because they represent the typical kind of work that I find myself doing whenever I use ROOT: pulling together documentation from different places, translating the examples into the work I'm actually doing... and pounding my head against the desk whenever there are no comments, or I get yet another segmentation fault.<sup>27</sup>

If you'd like to see how I solved those same exercises, you'll find my code in `PlotGraphs.C` (for exercises 14-16) and `MakeNtuple.C` (for exercise 17).<sup>28</sup>

Good luck!

---

<sup>27</sup> Now you know the reason for my bald patches!

<sup>28</sup> Maybe now you're thinking, "Wow! It's lucky I turned to the last page before I actually started doing any of the work!" Take my word for it: reading my solution is not a substitute for working through the problem yourself.