More C++ and Root

Today's Lecture

- Series of topics from C++
- Example of thinking though a problem
- Useful physics classes in ROOT

Functions

```
const double PI = 3.14159265;      // global constant
double ellipseArea(double, double);  // prototype
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}

double ellipseArea(double a, double b){
  return PI*a*b;
}
```

- by this point you are used to the syntax of a function

Pass by value

- Imagine that you wrote the following code

```
void tryToChangeArg(int x){
  x = 2*x;
}
```

```
int x = 1;
tryToChangeArg(x);
cout << "now x = " << x << endl;
```

- What would be the output ?

Pass by Value

- Imagine that you wrote the following code

```
void tryToChangeArg(int x){
  x = 2*x;
}
```

- What would be the output ?

```
int x = 1;
tryToChangeArg(x);
cout << "now x = " << x << endl;
```

- you might be tempted to say 2 but the answer is 1

Pass by value

```
void tryToChangeArg(int x){
  x = 2*x;
}
```

```
int x = 1;
tryToChangeArg(x);
cout << "now x = " << x << endl;
```

- passing by value takes the value of the variable and assigns it to a new local copy of the variable
- Inside the function the local copy gets changed to 2

Pass by Value

- passing by value takes the value of the variable and assigns it to a new local copy of the variable
- Inside the function the local copy gets changed to 2
- However, after the function ends nothing is returned and so outside the function the value of x is still 1

```
void tryToChangeArg(int x){
  x = 2*x;
}
```

```
int x = 1;
tryToChangeArg(x);
cout << "now x = " << x << endl;
```

Pass by Reference

- There is another way to do this
- This takes the address of the variable and passes a reference to the variable outside the function
- IN this case it DOES return 2

```
void tryToChangeArg(int&);

void tryToChangeArg(int& x){
  x = 2*x;
}

int main(){
  int x = 1;
  tryToChangeArg(x);
  cout << "now x = " << x << endl;
}
```

Scope

```
double pow(double x, int n){
  double y = static_cast<double>(n) * log(x);
  return exp(y);
}

...
double y = pow(3,2);  // this is a different y
```

- The variable y in the definition of pow is local.
- We can use the same variable name outside this function with no effect on or from the variable y inside pow.
- We say that the scope of y inside pow is inside

Default Arguments

```
double line(double x, double slope=1, double offset=0);
```

Define as usual

```
double line(double x, double slope, double offset){
  return x*slope  +  offset;
}
```

```
y = line (x, 3.7, 5.2);  // here slope=3.7, offset=5.2
y = line (x, 3.7);       // uses offset=0;
y = line (x);            // uses slope=1, offset=0
```

Function Overload

- Can have multiple functions
- which one gets called depends on type of argument passed

```
double cube(double);
double cube (double x){
  return x*x*x;
}

double cube(float);
double cube (float x){
  double xd = static_cast<double>(x);
  return xd*xd*xd;
}
```

Function Overload

```
float x;
double y;
double z = cube(x);  // calls cube(float) version
double z = cube(y);  // calls cube(double) version
```

- Can have multiple functions
- called "overloading"

Arrays

- The number in brackets [] gives the total number of elements,e.g. the array score above has 10 elements, numbered 0 through 9.
- The individual elements are referred to as score[0], score[1], score[2], ..., score[9]
- Declaring an array: data-type variableName [numElements];
- int score[10];
- If you try to access score[10] this is an error!

```
int score[10];
double energy[50], momentum[50];
const int MaxParticles = 100;
double ionizationRate[MaxParticles];
```

Arrays

- An array can also have two or more indices. A two-dimensional array is often used to store the values of a matrix:
- const int numRows = 2;
- const int numColumns = 3;
- double matrix[numRows][numColumns];
- Again, notice that the array size is 2 by 3, but the row index runs from 0 to 1 and the column index from 0 to 2.
- matrix[i][j], matrix[i][j+1], etc.
- The elements are stored in memory in the order:
- Usually we don't need to know how the data are stored internally

Initialization

One dimensional array:

```
int myArray[5] = {2, 4, 6, 8, 10};
```

Multidimension:

```
double matrix[numRows][numColumns] =
                    { {3, 7, 2}, {2, 5, 4} };
```

EXAMPLE

```
// Initialize vector x and matrix A
const int n = 5;
double x[n];
double A[n][n];
for (int i=0; i<n; i++){
  x[i] = someFunction(i);
  for (int j=0; j<n; j++){
    A[i][j] = anotherFunction(i, j);
  }
}

// Now find y = Ax
double y[n];
for (int i=0; i<n; i++){
  y[i] = 0.0;
  for (int j=0; j<n; j++){
    y[i] += A[i][j] * x[j];
  }
}
```

Passing to a function

- when we pass an array to a function in the actual call we don't need the brackets
- If we include a specific element of the array then we are actually evaluating that first - in this case a double witch happens to be the ith element of the myMatrix

```
double sumElements(double a[], int len){
  double sum = 0.0;
  for (int i=0; i<len; i++){
    sum += a[i];
  }
  return sum;
}
```

```
double s = sumElements(myMatrix, itsLength);
```

```
double x = sqrt(myMatrix[i]);
```

Pass by Reference

```
void changeArray (double a[], int len){
  for(int i=0; i<len; i++){
    a[i] *= 2.0;
  }
}

int main(){
  ...
  changeArray(a, len);    // elements of a doubled
```

- note - passing an array to a function works like pass by reference!

Pointers

- A pointer is the address of a variable in memory
- The notation is to use the * when declaring a pointer
- Note that as long as the star is between the type and the name it doesn't matter!

```
int* iPtr;
double * xPtr;
char *c;
float *x, *y;
```

Pointers

- Here & means "address of". Don't confuse it with the & used when passing arguments by reference.

```
int i = 3;
```

```
int* iPtr = &i;
```

Initializing Pointers

- A statement like
- int* iPtr;
- declares a pointer variable, but does not initialize it. It will be
- pointing to some “random” location in memory. We need
- to set its value so that it points to a location we’re interested in,
- e.g., where we have stored a variable:
- iPtr = &i;
- (just as ordinary variables must be initialized before use).

Different Pointers

```
int* iPtr;       //  type "pointer to int"
float* fPtr;     //  type "pointer to float"
double* dPtr;    //  type "pointer to double"
```

- We need different types of pointers because in general, the different data types (int, float, double) take up different amounts of memory.

- If declare another pointer and set then the +1 means "plus one unit of memory address for int", i.e., if we had int variables stored contiguously, jPtr would point to the one just after iPtr.

```
int* jPtr = iPtr + 1;
```

- But the types float, double, etc., take up different amounts of memory, so the actual memory address increment is different.

Pointers in Functions

```
void passPointer(int* iPtr){
  *iPtr += 2;          // note *iPtr on left!
}

...
int i = 3;
int* iPtr = &i;
passPointer(iPtr);
cout << "i = " << i << endl;    // prints i = 5
passPointer(&i);               // equivalent to above
cout << "i = " << i << endl;    // prints i = 7
```

When a pointer is passed as an argument, it divulges an address to the called function, so the function can change the value stored at that address - acts like pass by reference...

Test

What would this piece of code output?

```
int i = 3;
int& j = i;          // j is a
j = 7;
cout << "i = " << i << endl;
```

Test

What would this piece of code output?

```
int i = 3;
int& j = i;        // j is a
j = 7;
cout << "i = " << i << endl;
```

i is now 7! The second line assigns the address of the variable j to i. Change j and now you also change i

What to do with pointers

- You can do lots of things with pointers in C++, many of whichresult in confusing code and hard-to-find bugs.
- One of the main differences between Java and C++: Java doesn't have pointer variables (generally seen as a Good Thing).
- The main usefulness of pointers for us is that they will allow us to allocate memory (create variables) dynamically, i.e., at run time, rather than at compile time.
- Be careful - the misallocation of memory in C++ is a major source of memory leaks, bugs, and general obfuscation . General advice: keep it as simple as possible

Classes in C++

- A class is something like a user-defined data type.
- Typically this would be in a file called MyClassName.h and the definitions of the functions would be in MyClassName.C
- Note the semi-colon after the closing brace.

```
class MyClassName {
  public:
    public function prototypes and
    data declarations;
    ...
  private:
    private function prototypes and
    data declarations;
    ...
};
```

Example Class

Say we wanted to represent vectors in 2-D we might make a class like:

```
class TwoVector {
  public:
    TwoVector();
    TwoVector(double x, double y);
    double x();
    double y();
    double r();
    double theta();
    void setX(double x);
    void setY(double y);
    void setR(double r);
    void setTheta(double theta);
  private:
    double m_x;
    double m_y;
};
```

Class Header

- The header file must be included (#include "MyClassName.h") in other files where the class will be used.
- To avoid multiple declarations we use a

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {
  public:
    ...
  private:
    ...
};

#endif
```

Objects

```
double a;     // a is a variable of type double
```

```
#include "TwoVector.h"
int main() {
  TwoVector v;  // v is an object of type TwoVector
```

- (Actually, variables are also objects in C++. Sometimes class instances are called "class objects" -- distinction is not important.)
- A class contains in general both:
  - variables, called "data members" and
  - functions, called "member functions" (or "methods

Data Members

```
private:
  double m_x;
  double m_y;
```

- Their values define the "state" of the object. Because here they are declared private, a TwoVector object's values of m_x and m_y cannot be accessed directly, but only from within the class's member functions (more later).
- The optional prefixes m_ indicate that these are data members.
- Some authors use e.g. a trailing underscore. (Any valid identifieris allowed.)

Constructors

- These are special functions called constructors.
- A constructor always has the same name as that of the class.
- It is a function that is called when an object is created.
- A constructor has no return type.
- There can be in general different constructors with different signatures (type and number of arguments).

```
public:
  TwoVector();
  TwoVector(double x, double y);
```

Constructors

When we declare an object, the constructor is called which has the matching signature, e.g.,

```
TwoVector u;    // calls TwoVector::TwoVector()
```

The constructor with no arguments is called the “default constructor”. If, however, we say

```
TwoVector v(1.5, 3.7);
```

then the version that takes two double arguments is called.
If we provide no constructors for our class, C++ automatically gives us a default constructor

IN ACTION

In the file that defines the member functions, e.g., TwoVector.cc, we precede each function name with the class name and :: (the scope resolution operator). For our two constructors we have:

```
TwoVector::TwoVector() {
  m_x = 0;
  m_y = 0;
}
TwoVector::TwoVector(double x, double y) {
  m_x = x;
  m_y = y;
}
```

The constructor serves to initialize the object.
If we already have a TwoVector v and we say
TwoVector w = v;
this calls a “copy constructor” (automatically provided).

FUNCTIONS

```
void TwoVector::setX(double x) { m_x = x; }
void TwoVector::setY(double y) { m_y = y; }
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

These are “setter” functions. As they belong to the class, they are allowed to manipulate the private data members m_x and m_y.
To use with an object, use the “dot” notation:

```
TwoVector v(1.5, 3.7);
v.setX(2.9);      // sets v's value of m_x to 2.9
```

Pointers to objects

```
TwoVector* vPtr;  // type "pointer to TwoVector"
```

This doesn't create an object yet!   This is done with, e.g.,

```
vPtr = new TwoVector(1.5, 3.7);
```

vPtr is now a pointer to our object. With an object pointer, we call member functions (and access data members) with **->** (not with "**.**"), e.g.,

```
double vX = vPtr->x();
cout << "vX = " << vX << endl;  // prints vX = 1.5
```

This Pointer

```
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

- C++ defines a pointer "this" which is a pointer to the object that is being called

Information Hiding

- Note that by declaring the m_x and m_y private this means that we can only access them via the 'accessor' functions
- Why do we do this?

C++ Paradigm

- The basic idea of the C++ paradigm is to 'hide' the implementation of private member data or function
- This way - if the implementation changes inside the class. None of the user code needs to know about it!
- This way we can encapsulate the behavior in a small piece of code that is used by a large number of clients

Example

- Let's write a program that checks to see if a number between 1 and 1000 is a prime number

Let's think

- How do we decide if a number is prime.

Let's Think

- How do we decide if a number is prime.
- If it is only divisible by itself and 1 then it is prime

so

- So a straightforward way to test this is to just try to divide the number by every number between 2 and N (where N is the test number) and see if there is any factors

V

- first we will get an input from the keyboard to see what number we have to test
- get_valid_number()

```
int main()
{
  int number;

  cout << "This program tests to see if an integer\n";
  cout << "is a prime between 1 and 1000.\n\n\n";

  number = get_valid_number();
  while (number != 0) {
    cout << "The number " << number << " is ";
    if (!small_prime(number))
      cout << "not ";
    cout << "a prime between 1 and 1000.\n\n";
    number = get_valid_number();
  }

  return 0;
}
```

Retrieve the number

- Get a number, if it is out of range output a error message and stay in the loop
- if it is inside exit loop and return the number

```
int get_valid_number()
{
  int number;
  do {
    cout << "Enter an integer between 1 and 1000 (incl) (or 0 to end program): ";
    cin >> number;
    if (number < 0 || number > 1000)
      cout << "number out of range, try again!" << endl;
  } while (number < 0 || number > 1000);

  return number;
}
```

Next Step

- After a valid input check to see if it is prime

```
int main()
{
  int number;

  cout << "This program tests to see if an integer\n";
  cout << "is a prime between 1 and 1000.\n\n\n";

  number = get_valid_number();
  while (number != 0) {
    cout << "The number " << number << " is ";
    if (!small_prime(number))
      cout << "not ";
    cout << "a prime between 1 and 1000.\n\n";
    number = get_valid_number();
  }

  return 0;
}
```

Prime Checker

- Iterate through all numbers from 2 to N-1 and check to see if there is a remainder or not
- If you can't find one return true

```
Logical small_prime(int integer)
{
  for (int factor = 2; factor<integer; factor++) {
    if ((integer % factor) == 0)
      return False;
  }
  return True;
}
```

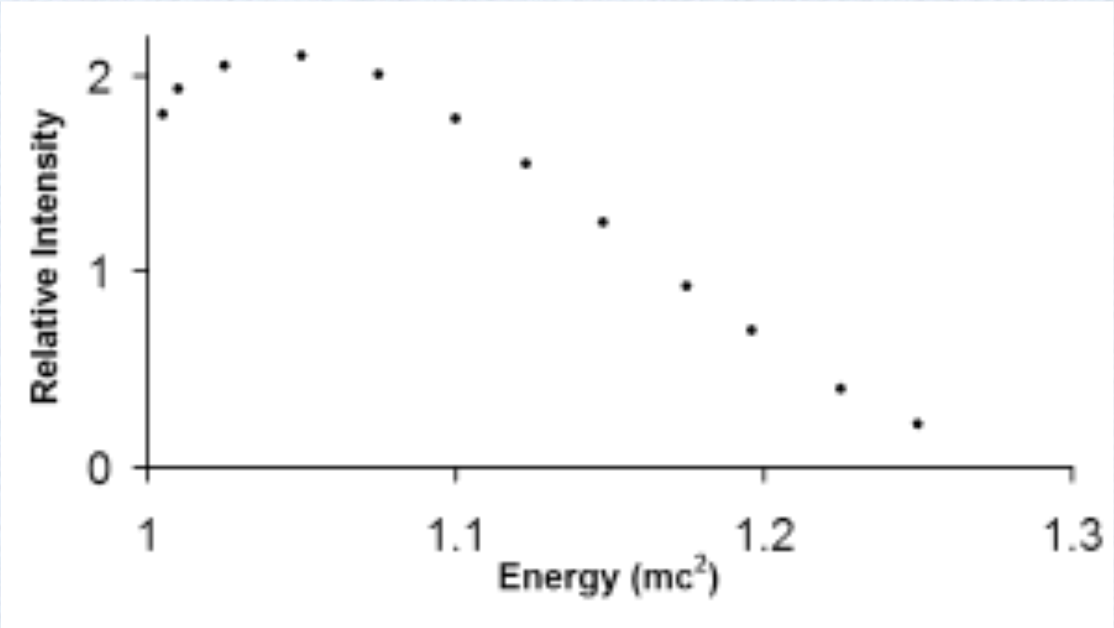
ROOT Physics Classes

- Example code on blackboard
- An example introducing some convenient physics classes

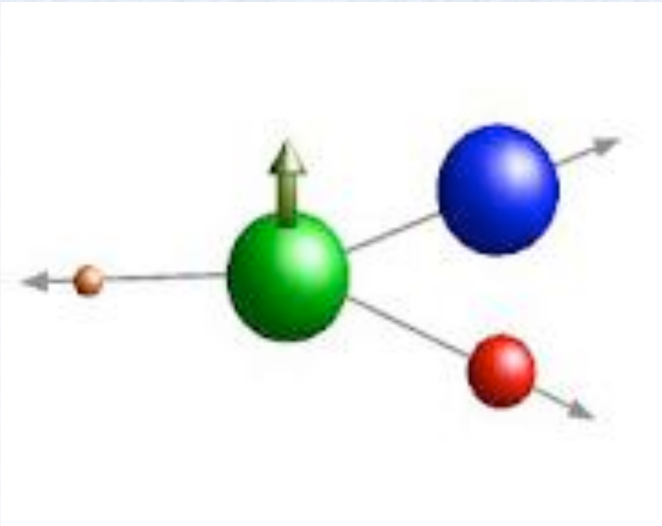
```
if (!gROOT->GetClass("TGenPhaseSpace")) gSystem.Load("libPhysics");

TLorentzVector target(0.0, 0.0, 0.0, 0.938);
TLorentzVector beam(0.0, 0.0, .65, .65);
TLorentzVector W = beam + target;

//(Momentum, Energy units are Gev/C, GeV)
Double_t masses[3] = { 0.938, 0.139, 0.139} ;

TGenPhaseSpace event;
event.SetDecay(W, 3, masses);

TH2F *h2 = new TH2F("h2","h2", 50,1.1,1.8, 50,1.1,1.8);

for (Int_t n=0;n<100000;n++) {
   Double_t weight = event.Generate();

   TLorentzVector *pProton = event.GetDecay(0);

   TLorentzVector *pPip    = event.GetDecay(1);
   TLorentzVector *pPim    = event.GetDecay(2);

   TLorentzVector pPPip = *pProton + *pPip;
   TLorentzVector pPPim = *pProton + *pPim;

   h2->Fill(pPPip.M2() ,pPPim.M2() ,weight);
}
h2->Draw();
```

Aside : 2 versus 3 body decay

- For a 2 body decay as you showed in your homework you must have a fixed energy in the rest frame of the decaying particle. Conservation of energy and momentum
- For a 3 body decay there is enough phase space that you expect a continuous rather than discrete spectrum of energy



β decay spectrum



TLorentzVector

- http://root.cern.ch/root/html/TLorentzVector.html
- Constructors, and many member functions which both access the internal data and know how to rotate , boost, and add Lorentzvectors

public:

	TLorentzVector (const Double_t* carray)
	TLorentzVector (const Float_t* carray)
	TLorentzVector (const TLorentzVector& lorentzvector)
	TLorentzVector (const TVector3& vector3, Double_t t)
	TLorentzVector (Double_t x = 0.0, Double_t y = 0.0, Double_t z = 0.0, Double_t t = 0.0)
virtual	**~TLorentzVector** ()
Double_t	**Angle** (const TVector3& v) const
Double_t	**Beta** () const
void	**Boost** (const TVector3& b)
void	**Boost** (Double_t, Double_t, Double_t)
TVector3	**BoostVector** () const
static TClass*	**Class** ()
Double_t	**CosTheta** () const
Double_t	**DeltaPhi** (const TLorentzVector& v) const
Double_t	**DeltaR** (const TLorentzVector& v) const
Double_t	**Dot** (const TLorentzVector& q) const
Double_t	**DrEtaPhi** (const TLorentzVector& v) const
Double_t	**E** () const
Double_t	**Energy** () const
Double_t	**Et** () const
Double_t	**Et** (const TVector3& v) const
Double_t	**Et2** () const
Double_t	**Et2** (const TVector3& v) const
Double_t	**Eta** () const
TVector2	**EtaPhiVector** ()
Double_t	**Gamma** () const
void	**GetXYZT** (Double_t* carray) const
void	**GetXYZT** (Float_t* carray) const
virtual TClass*	**IsA** () const
Double_t	**M** () const

- Load library
- Construct two LorentzVectors

```
if (!gROOT->GetClass("TGenPhaseSpace")) gSystem.Load("libPhysics");

TLorentzVector target(0.0, 0.0, 0.0, 0.938);
TLorentzVector beam(0.0, 0.0, .65, .65);
TLorentzVector W = beam + target;

//(Momentum, Energy units are Gev/C, GeV)
Double_t masses[3] = { 0.938, 0.139, 0.139} ;

TGenPhaseSpace event;
event.SetDecay(W, 3, masses);

TH2F *h2 = new TH2F("h2","h2", 50,1.1,1.8, 50,1.1,1.8);

for (Int_t n=0;n<100000;n++) {
   Double_t weight = event.Generate();

   TLorentzVector *pProton = event.GetDecay(0);

   TLorentzVector *pPip    = event.GetDecay(1);
   TLorentzVector *pPim    = event.GetDecay(2);

   TLorentzVector pPPip = *pProton + *pPip;
   TLorentzVector pPPim = *pProton + *pPim;

   h2->Fill(pPPip.M2() ,pPPim.M2() ,weight);
}
h2->Draw();
```

- Load library
- Construct two LorentzVectors
- TGenPhaseSpace

```
if (!gROOT->GetClass("TGenPhaseSpace")) gSystem.Load("libPhysics");

TLorentzVector target(0.0, 0.0, 0.0, 0.938);
TLorentzVector beam(0.0, 0.0, .65, .65);
TLorentzVector W = beam + target;

//(Momentum, Energy units are Gev/C, GeV)
Double_t masses[3] = { 0.938, 0.139, 0.139} ;

TGenPhaseSpace event;
event.SetDecay(W, 3, masses);

TH2F *h2 = new TH2F("h2","h2", 50,1.1,1.8, 50,1.1,1.8);

for (Int_t n=0;n<100000;n++) {
   Double_t weight = event.Generate();

   TLorentzVector *pProton = event.GetDecay(0);

   TLorentzVector *pPip    = event.GetDecay(1);
   TLorentzVector *pPim    = event.GetDecay(2);

   TLorentzVector pPPip = *pProton + *pPip;
   TLorentzVector pPPim = *pProton + *pPim;

   h2->Fill(pPPip.M2() ,pPPim.M2() ,weight);
}
h2->Draw();
```

- http://root.cern.ch/root/html/TGenPhaseSpace.html
- Generates n-body phase space events (by default assumes constant cross-section) ie. only phase space comes in

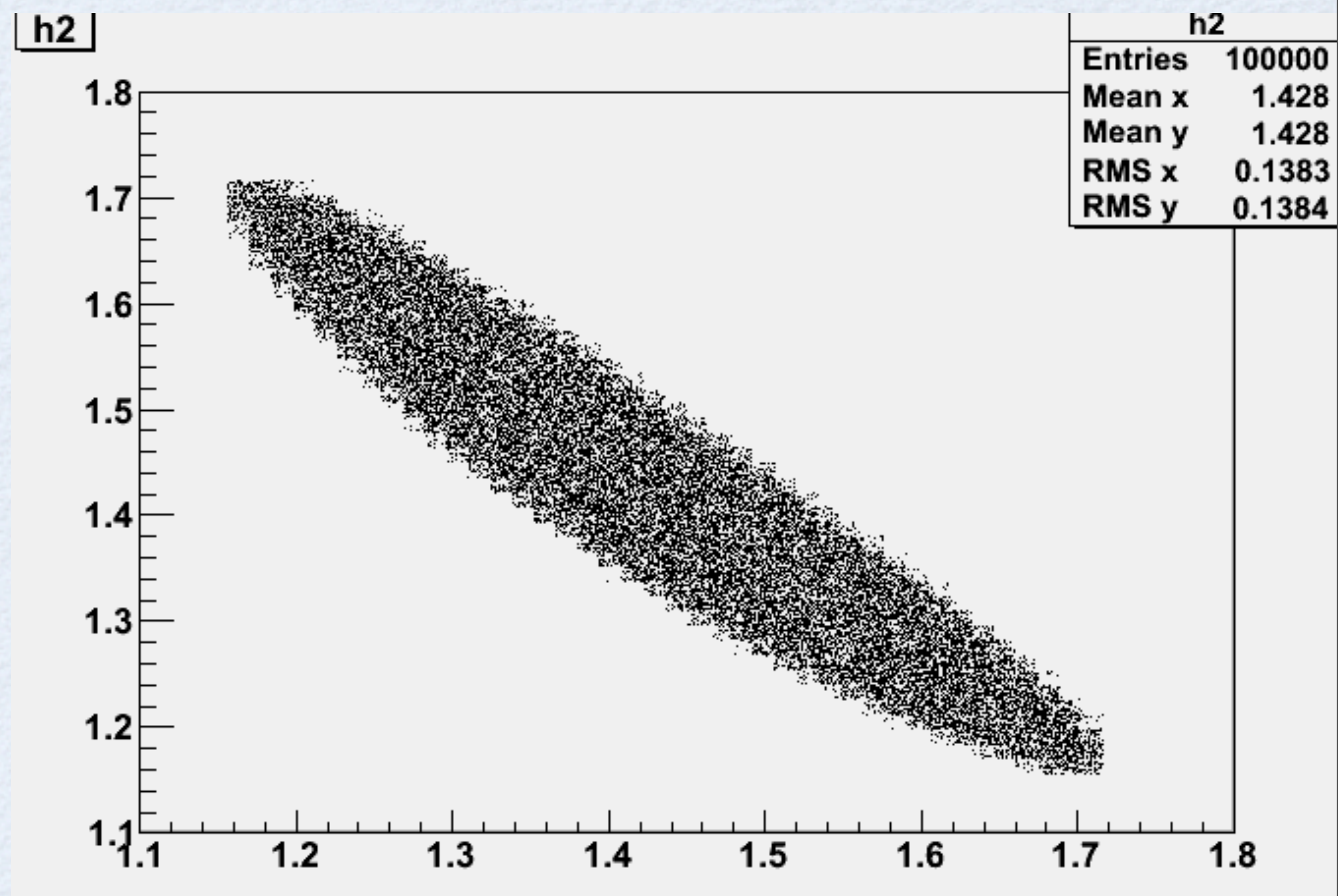
- Set properties of phase space generator
- Create histogram
- Generate events
- Calculate the invariant mass of the proton and pion both positive and negative from the decay products
- Draw the histogram

```
if (!gROOT->GetClass("TGenPhaseSpace")) gSystem.Load("libPhysics");

TLorentzVector target(0.0, 0.0, 0.0, 0.938);
TLorentzVector beam(0.0, 0.0, .65, .65);
TLorentzVector W = beam + target;

//(Momentum, Energy units are Gev/C, GeV)
Double_t masses[3] = { 0.938, 0.139, 0.139} ;

TGenPhaseSpace event;
event.SetDecay(W, 3, masses);

TH2F *h2 = new TH2F("h2","h2", 50,1.1,1.8, 50,1.1,1.8);

for (Int_t n=0;n<100000;n++) {
   Double_t weight = event.Generate();

   TLorentzVector *pProton = event.GetDecay(0);

   TLorentzVector *pPip    = event.GetDecay(1);
   TLorentzVector *pPim    = event.GetDecay(2);

   TLorentzVector pPPip = *pProton + *pPip;
   TLorentzVector pPPim = *pProton + *pPim;

   h2->Fill(pPPip.M2() ,pPPim.M2() ,weight);
}
h2->Draw();
```

Dalitz Plot

- What we have just generated is called a Dalitz plot
- Provides information on the phase space of three body decays
- Uniform if no angular correlations between decay products
- If resonant structures or correlations phase space is populated unequally

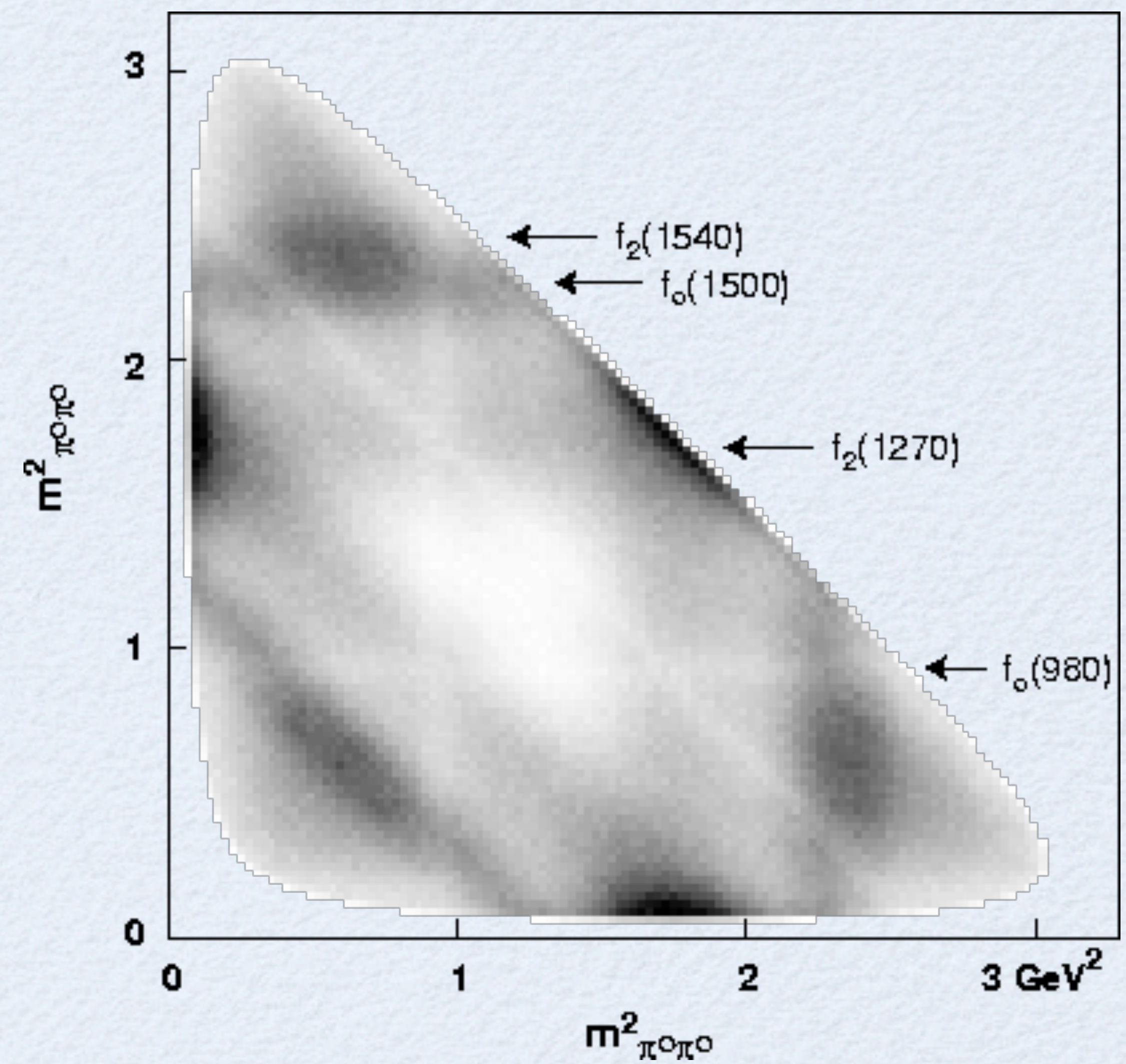
$M(\pi^+ p)$



$M(\pi^- p)$

Example

- proton+antiproton to three neutral pions
- clear structure and resonances reveals intermediate particles that were not seen by the detector itself but can be inferred from the measurements made!



Next Week

- Introduction to Limits (or how I learned to report just how much i didn't see new physics)
- Root statistics classes
- Presentations!!