Plan for the day

- **Function pointers.**
- **Build a simple fitting framework from the ground up.**
- **Understand how the core GooFit engine works.**
- **Do simple fits in GooFit.**

Function pointers

- **In fully object-oriented fitting code (eg RooFit!) we would use virtual functions:**

```
class FooAbsPdf {
public:
  virtual double evaluate (FooEvent x) const = 0;
};

class FooGaussianPdf : public FooAbsPdf {
public:
  virtual double evaluate (FooEvent evt) const {
    double ret = exp(-0.5*pow((mean - evt.xval) / sigma, 2));
    ret /= sigma;
    ret /= sqrt(2*M_PI);
    return ret;
  }
private:
  double mean;
  double sigma;
};
```

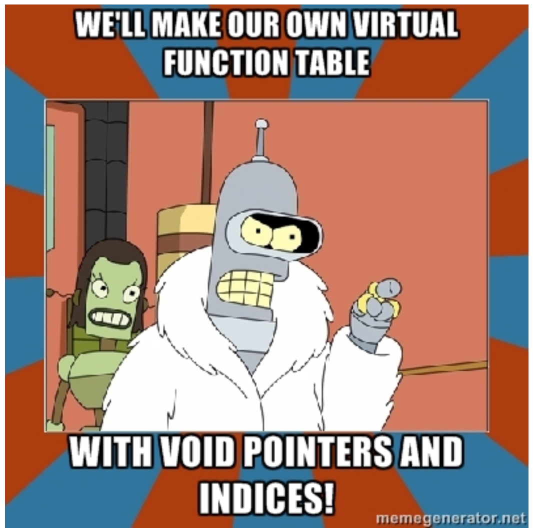
```
// This pointer is initialised somewhere in setup code.
FooAbsPdf* currentPdf;

void fcn_glue (int& npar,
               double* deriv,
               double& fVal,
               double param[],
               int flag) {

  currentPdf->setParams(param);
  double nll = 0;
  for (int i = 0; i < numEvents; ++i) {
    // Runtime figures out which implementation to call.
    nll -= 2*log(currentPdf->evaluate(eventArray[i]));
  }
  fVal = nll;
}
```

CUDA doesn't support virtual functions!

- **That's ok. Object orientation and other high-level language features are just syntactic sugar for the underlying ones and zeroes, anyway!**
- **We will implement our own virtual function table, like Real Programmers should!**
- **Simple code to be improved later (`example3a.cu`):**


WE'LL MAKE OUR OWN VIRTUAL FUNCTION TABLE
WITH VOID POINTERS AND INDICES!
memegenerator.net

```
// 512 functions should be enough for anyone.
__device__ void* dev_fcn_table[512];
typedef double (*dev_fcn_ptr) (double, double, double);

struct GeneralFcn : public thrust::unary_function<double, double> {
public:
  GeneralFcn (unsigned int fcn, double p1, double p2);
  __device__ double operator () (double xval);
private:
  unsigned int fcnIdx;
  double param1;
  double param2;
};
```

```
GeneralFcn::GeneralFcn (unsigned int fcn,
                        double p1,
                        double p2)
  : fcnIdx(fcn)
  , param1(p1)
  , param2(p2)
{}

__device__ double GeneralFcn::operator () (double xval) {
  dev_fcn_ptr theFunction;
  theFunction = reinterpret_cast<dev_fcn_ptr>(dev_fcn_table[fcnIdx]);
  double pdfVal = (*theFunction)(xval, param1, param2);
  return -2*log(pdfVal);
}
void fcn_glue (int& npar,
               double* deriv,
               double& fVal,
               double param[],
               int flag) {
  double initVal = 0;
  GeneralFcn functor(host_fcnIdx, param[0], param[1]),
  fVal = thrust::transform_reduce(dev_data->begin(),
                                  dev_data->end(),
                                  functor,
                                  initVal,
                                  thrust::plus<double>);
}
```

```
__device__ double dev_Gaussian (double xval, double mean, double sigma) {
  double ret = exp(-0.5*pow((xval - mean) / sigma, 2));
  ret /= sigma;
  ret /= sqrt(2*M_PI);
  return ret;
}

__device__ double dev_BreitWigner (double xval, double mean, double width) {
  double arg = xval - mean;
  double ret = width / (arg*arg + 0.25*width*width);
  ret /= (2*M_PI); // Normalise over -inf to inf
  return ret;
}

__device__ dev_fcn_ptr ptr_to_Gaussian = dev_Gaussian;
__device__ dev_fcn_ptr ptr_to_BreitWigner = dev_BreitWigner;
int host_fcnIdx = 0;
int main (int argc, char** argv) {
  // Generate random data (skipped)
  // Initialise function table
  void* host_fcn_ptrs[512];
  cudaMemcpyFromSymbol(host_fcn_ptrs+0, ptr_to_Gaussian, sizeof(void*));
  cudaMemcpyFromSymbol(host_fcn_ptrs+1, ptr_to_BreitWigner, sizeof(void*));
  cudaMemcpyToSymbol(dev_fcn_table, host_fcn_ptrs, 2*sizeof(void*));

  // Fit to Gaussian or Breit-Wigner
  host_fcnIdx = atoi(argv[1]);
  TMinuit minuit(2);
  minuit.DefineParameter(0, "mean",  0.04, 0.01, -1.0, 1.0);
  minuit.DefineParameter(1, "width", 0.50, 0.01, 0.0, 2.0);
  minuit.SetFCN(fcn_glue);
  minuit.Migrad();

  // Free the device memory.
  delete dev_data;
  return 0;
}
```

Some obvious problems

- **We can only deal with functions taking two parameters.**
- **What if we need to normalise something numerically?**
- **Dependence on more than one variable?**
- **Goodness-of-fit metrics that are not NLL?**
- **Binned fits?**
- **Plotting the PDF!**
- **All shall be revealed.**

Arbitrary number of parameters

- **We'll need some changes to the function signature (`example3b.cu`):**

```
__constant__ double dev_params[512];
__constant__ unsigned int dev_indices[512];
typedef double (*dev_fcn_ptr) (double);

__device__ double dev_Gaussian (double xval) {
  double mean  = dev_params[dev_indices[1]]; // Not a typo
  double sigma = dev_params[dev_indices[2]];

  double ret = exp(-0.5*pow((xval - mean) / sigma, 2));
  ret /= sigma;
  ret /= sqrt(2*M_PI);
  return ret;
}

// Similar changes to dev_BreitWigner
```

- **What is in `dev_indices[0]`?**
- **The total number of parameters of the function. Which is not needed for the Gaussian, but allows, for example, polynomials of arbitrary degree.**

```
__device__ double dev_Polynomial (double xval) {
  // nP  c1  c2  c3  ...
  int numParams = dev_indices[0];
  double ret = 0;
  for (int i = 0; i < numParams; ++i) {
    double coef  = dev_params[dev_indices[i + 1]];
    double power = pow(xval, i);
    ret         += coef*power;
  }

  // Not good to normalise from -inf to inf. Avoid the problem by hardcoding
  // integration limits. Note use of numerical integration to avoid places where
  // polynomial goes negative - PDF is always positive!

  double integral = 0;
  for (double xint = -5.0; xint < 5.0; xint += 0.01) {
    double curr = 0;
    for (int i = 0; i < numParams; ++i) {
      double coef  = dev_params[dev_indices[i + 1]];
      double power = pow(xint, i);
      curr        += coef*power;
    }
    if (curr < 0) continue;
    integral += 0.01*curr;
  }
  return max(ret / integral, 1e-6);
}
```

Other changes

- **Since `GeneralFcn` no longer stores the parameters, we need to explicitly copy them:**

```
void fcn_glue (int& npar, double* deriv, double& fVal,
               double param[], int flag) {
  GeneralFcn functor(host_fcnIdx);
  double initVal = 0;
  cudaMemcpyToSymbol(dev_params, param, npar*sizeof(double));
  fVal = transform_reduce(dev_data->begin(),
                          dev_data->end(),
                          functor,
                          initVal,
                          thrust::plus<double>());
}
```

- **We also need a more complicated setup structure:**

```
  unsigned int host_indices[10];
  host_indices[0] = npars;
  for (unsigned int i = 0; i < npars; ++i) host_indices[i+1] = i;
  cudaMemcpyToSymbol(dev_indices, host_indices,
                     (1+npars)*sizeof(unsigned int));
```

- **Entropy cannot be reduced, only moved around. But you can get more functionality per unit entropy.**

Still more general!

- We are now able to specify any function by an index into `dev_fcn_table`. But all our functions assume that they are the only game in town: They each interpret `dev_indices` as having their own number of parameters in the 0th entry, their first parameter in the 1st entry, and so on.
- What if we wanted to call two functions? For example, if we want to make a PDF that is the sum of two base PDFs.
- We could make a special `fcn_glue_sum` that does two `transform` calls to evaluate separate PDFs, then a `transform_reduce` to add them together with some weight and take the logarithm - but this is getting complicated, and what if you had three PDFs?
- Let us instead make our functions more general by adding another layer of abstraction. Specifically, a function will now be defined by two integers: Its index in `dev_fcn_table`, and its index in `dev_indices`, denoting the place where its parameter information starts.
- Full code in `example3c.cu`.

```
typedef double (*dev_fcn_ptr) (double, unsigned int);

__device__ double dev_Gaussian (double xval, unsigned int pIdx) {
  double mean  = dev_params[dev_indices[pIdx + 1]];
  double sigma = dev_params[dev_indices[pIdx + 2]];
  double ret = exp(-0.5*pow((xval - mean) / sigma, 2));
  ret /= sigma;
  ret /= sqrt(2*M_PI);
  return ret;
}

// Similar changes to dev_BreitWigner and dev_Polynomial.
// GeneralFcn gains a ’parIdx’ member variable.
```

Why did we do that, again?

- **Arbitrarily deep function nesting, that's why.**
- **Because functions are now fully specified by the `(fcnIdx, parIdx)` pair, we can write functions that call other functions.**
- **For example we can write a `dev_SumOfFunctions` whose index structure looks like this:**

  ```
  number of parameters = 5
  index of weight of first function = 0
  fcnIdx of first function = 0
  parIdx of first function = 6
  fcnIdx of second function = 0
  parIdx of second function = 9
  ```

  to calculate the sum of two Gaussians.
- **A certain amount of keeping the tongue straight in the mouth is necessary.**
- **If you have to set up the index array by hand, and painstakingly recalculate the indices every time your PDF changes, it's reasonable to ask if you're really saving yourself any effort!**
- **We will deal with these issues later.**