Functor classes

- Use a Functor class to wrap any C++ callable object which has a right signature
- Can work with:
  - free C functions
  - C++ classes (or structs) implementing operator()
  - member function of a class
- The requirement is that they must have the right signature
  - Ex: a function of type :
    - `double f ( double )`

Examples of Using Functors

- free functions:

```
double freeFunction( double x) {.............}

Functor f1(freeFunction);
f1(x); // evaluate functor at value x
```

- classes implementing operator()

```
struct MyFunction {
  double operator() (double x) { .........}
};
Functor f2(MyFunction());
f2(x); // evaluate functor at value x
```

- member functions

```
class MyClass {
.........
double Eval(double x) {.......}
.........
};
MyClass * pMyClass = ......;
Functor f3( pMyClass, &MyClass::Eval);
f3(x); // evaluate functor at value x
```

Advantages of Functor classes

- They are very convenient for users
  - easier to defining a functor than implementing a class following an abstract interface
  - Have value semantics (like shared pointers)
- Very flexible
  - users can customize the callable objects using its state
  - this is not possible when using a free function
- Example of usage in C++:
  - STL algorithm are based on similar concept
    - very powerful and flexible
  - boost::function (will be in next C++ standard)
  - Functor class from Alexandrescu (Loki)

Proposed usage in ROOT

- Use a Functor class in TF1 instead of using the free function pointer

```
double ( * ) ( double *, double * )
```

- Define a Functor class which provides this signature

```
class ParamFunctor  {

public:
   // Default constructor
   ParamFunctor ()  : fImpl(0) {}

   //    construct from a pointer to a member function
   template <class PtrObj, typename MemFn>
   ParamFunctor(const PtrObj& p, MemFn memFn)
      : fImpl(new ParamMemFunHandler<ParamFunctor, PtrObj, MemFn>(p, memFn))  {}

   //   construct from another generic Functor of multi-dimension
   template <typename Func>
   ParamFunctor( Func f) :
     fImpl(new ParamFunHandler<ParamFunctor,Func>(f) )  {}

   inline double operator() (double * x, double * p) const {
      return (*fImpl)(x,p);
   }

private :
  // ParamFunHandlerBase is a base class of ParamMemFunHandler and ParamFunHandler
  std::auto_ptr<ParamFunHandlerBase> fImpl;
};
```

Modifications to TF1

- Use this class as a data member of TF1 to replace the current `fFunction` pointer

```
//Double_t (*fFunction) (Double_t *, Double_t *);   //!Pointer to function
ROOT::Math::ParamFunctor fFunctor;  //!Functor object for compiled functions
```

- Have template constructors for callable objects and for member functions

```
// template constructors for creating a TF1 from a generic callable object
template <typename Func>
TF1(const char *name, Func f, Double_t xmin, Double_t xmax, Int_t npar) :
   TFormula(), TAttLine(), TAttFill(), TAttMarker()
{
   CreateFromFunctor(name, ROOT::Math::ParamFunctor(f), xmin, xmax, npar);
}
// template constructors for creating from a member function
template <class PtrObj, typename MemFn>
TF1(const char *name, const  PtrObj& p, MemFn memFn, Double_t xmin, Double_t xmax, Int_t npar) :
   TFormula(), TAttLine(), TAttFill(), TAttMarker()
{
   CreateFromFunctor(name, ROOT::Math::ParamFunctor(p,memFn), xmin, xmax, npar);
}

void CreateFromFunctor(const char *name, ROOT::Math::ParamFunctor f,
                       Double_t xmin, Double_t xmax, Int_t npar);
```

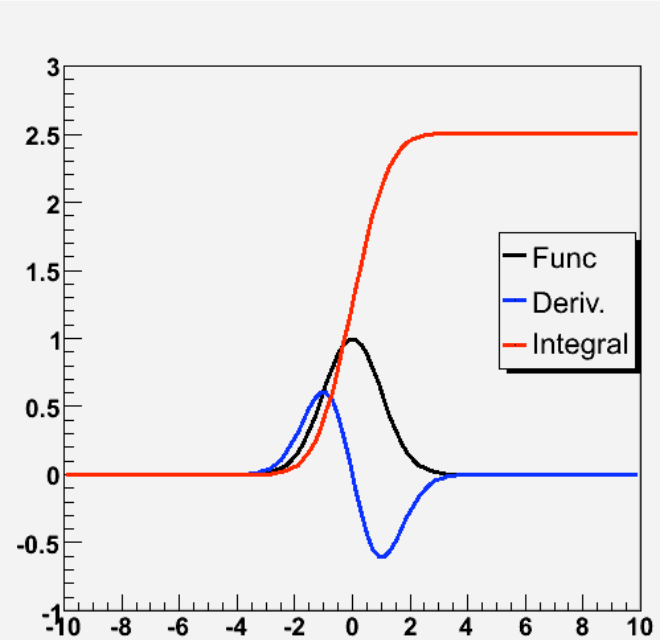
Example of Usage (1)

- Working example of how it works:

```
#include "TF1.h"
#include "TMath.h"


double MyFunc (double *x, double *p ) {
   return TMath::Gaus(x[0],p[0],p[1] );
}

struct MyDerivFunc {
   MyDerivFunc(TF1 * f): fFunc(f) {}
   double operator() (double *x, double * ) const {
      return fFunc->Derivative(*x);
   }
   TF1 * fFunc;
};
struct MyIntegFunc {
   MyIntegFunc(TF1 * f): fFunc(f) {}
   double operator() (double *x, double * ) const {
      double a = fFunc->GetXmin();
      return fFunc->Integral(a, *x);
   }
   TF1 * fFunc;
};
```

Example of Usage (2)

```
void testTF1Functor() {

   double xmin = -10;  double xmax = 10;
   TF1 * f1 = new TF1("f1",MyFunc,xmin,xmax,2);
   f1->SetParameters(0.,1.);
   f1->SetMaximum(3);   f1->SetMinimum(-1);
   f1->Draw();

   // derivatives function: no new parameters, they are stored in the parent function
   TF1 * f2 = new TF1("f2",MyDerivFunc(f1), xmin, xmax, 0);

   f2->SetLineColor(kBlue);
   f2->Draw("same");

   // integral function
   TF1 * f3 = new TF1("f3",MyIntegFunc(f1), xmin, xmax, 0);

   f3->SetLineColor(kRed);
   f3->Draw("same");
}
```



- this would be possible with the current TF1 only by using global objects

Further Remarks

- Different signatures can be easily matched using some Functor adapters
  - using something similar to std::bind2nd functions taking two parameters can be adapted to function with one parameter
  - one could project a multidimensional function in a 1D function
- Can also make it work from interpreter:
  - could use a virtual class implementing a virtual operator()
  - user defines the interpreted function object as a derived class of a base Functor

```
class MyDerivFunc : public ParamFunHandlerBase {
public:
   MyDerivFunc(TF1 * f): fFunc(f) {}
   double operator() (double *x, double * ) const {
      return fFunc->Derivative(*x);
   }
   // need a Clone to be able to copy through the base class
   MyDerivFunc * Clone() const { return new MyDerivFunc(fFunc); }
private:
   TF1 * fFunc;
};
```

Further Remarks (2)

- Would also like to have in TF1 defined the operator()
  - make it work easier with template methods requiring this signature
    - no need of an additional adapter
- Numerical Algorithm (Integration, Root Finder classes, etc..) will work also accepting a Functor
  - easier integration without need to have a common interface
  - better decoupling between the Function class (TF1) and the algorithm
  - can use a template method relying on the defined functor signature

Conclusions

- Propose to use a Functor class (a function wrapper) in TF1 to describe a callable object instead of a simple function pointer
- A prototype already exists working with a TF1 in compiled mode
- Investigate in more details all the consequences of these proposed changes
  - negligible changes in performances
    - ~ 10% for the simplest function like y = x1 + x2;
- Need to prototype also the solution for interpreted function and I/O
  - ideally is generation of the dictionary on the fly for the functor object passed by the user