# ADMB Foundation

http://admb-project.org/

# Coping with C++

## data types, function prototypes, overloading, new functions

ADMB Foundation

sibert@hawaii.edu

# Data types and operator overloading

Pedantic example:
1/3 using three different data types
(and three different operators)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void)
{
  int ix = 1;
  int iy = 3;
  int idiv = ix/iy;

  float fx = 1.0;
  float fy = 3.0;
  float fdiv = fx/fy;


  double dx = 1.0;
  double dy = 3.0;
  double ddiv = dx/dy;

  cout << "int:    " << setprecision(20) << (double)idiv << endl;
  cout << "float:  " << setprecision(20) << (double)fdiv << endl;
  cout << "double: " << setprecision(20) << (double)ddiv << endl;
}
```

Output[a]:

```
int:    0
float:  0.33333334326744079590
double: 0.33333333333333314830
```

[a] gnu 4.4.3; 64-bit; ubuntu 10.04

# Data types and operator overloading

An even more pendantic example:

```
#include <iostream>
#include <iomanip>
using namespace std;
class myint
{
  public:
    int i;
  myint(const int a) {i = a;}
};
int operator / (const myint x, const myint y)
{
  int res = x.i/y.i;
  cerr << "** Warning: integer division can be missleading" << endl;
  cerr << "   You are dividing " << x.i << " by " << y.i
       << " = " << res << endl;
  cerr << "   A better result may be "
       << (double)(x.i)/(double)(y.i) << endl;
  return(res);
}
int main(void)
{
  myint ix=1;
  myint iy=3;
  int idiv = ix/iy;
  .
  .
  .
}
```

Output[a]:

```
** Warning: integer division can be missleading
   You are dividing 1 by 3 = 0
   A better result may be 0.333333
int:    0
float:  0.3333333432674407959
double: 0.33333333333333331483
```

[a] gnu 4.4.3; 64-bit; ubuntu 10.04

# Data exchange with functions

## Passing data to functions

| | **Prototype** | **Call** |
|---|---|---|
| By value | `void foo(double a1, const prevariable a2);` | `foo(x, y);` |
| As pointer | `void foo(double * a1, const prevariable * a2);` | `foo(&x, &y);` |
| By reference | `void foo(double & a1, const prevariable & a2);` | `foo(x, y);` |

## Returning data from functions

| | **Prototype** | **Call** |
|---|---|---|
| As return value | `dvariable foo(double a1);` | `dvariable y = foo(x);` |
| As argument | `void foo(double & a1, prevariable & a2);` | `foo(x, y);` |

# General steps to adding functions

1. Preliminary testing
2. Add function prototypes to header file, e.g. `fvar.hpp`.
3. Write code for function body in source (`.cpp`) file.
4. Add source file name to `objects.lst`.
5. Compile and build libraries, i.e., `make`.

# The function foo(...) for scalar objects

## Function prototypes in `fvar.hpp`

```
.
.
.
double foo(const double x, const double a);
// could be inserted in the header file in place of prototype
// inline double foo(const double x, const double a)
//   { return(pow((x-a),2)); }

dvariable foo(const prevariable & x, const double & a);
dvariable foo(const prevariable & x, const prevariable & a);
.
.
.
```

## Function code in `foo.cpp`

```
#include <fvar.hpp>

double foo(const double x, const double a)
{
   double y;
   y = square(x-a);
   return (y);
}

dvariable foo(const prevariable & x, const prevariable & a)
{
   RETURN_ARRAYS_INCREMENT();
   dvariable y;
   y = square(x-a);
   RETURN_ARRAYS_DECREMENT();
   return (y);
}

dvariable foo(const prevariable & x, const double & a)
{
   RETURN_ARRAYS_INCREMENT();
   dvariable y;
   y = square(x-a);
   RETURN_ARRAYS_DECREMENT();
   return (y);
}
```

# The function foo(...) for vector and matrix objects

## Function prototypes in `fvar.hpp`

```
.
.
.
double foo(const double x, const double a);
dvariable foo(const prevariable & x, const double & a);
dvariable foo(const prevariable & x, const prevariable & a);

dvar_vector foo(const dvar_vector & x, const double & a);
dvar_vector foo(const dvar_vector & x, const prevariable & a);
dvar_vector foo(const dvar_vector & x, const dvar_vector & a);

dvar_matrix foo(const dvar_matrix & x, const prevariable & a);
.
.
.
```

## Function code in `foo.cpp`

```
dvar_vector foo(const dvar_vector & x, const prevariable & a)
{
   RETURN_ARRAYS_INCREMENT();
   const int j1 = x.indexmin();
   const int j2 = x.indexmax();
   dvar_vector y(j1,j2);
   for (int j = j1; j <= j2; j++)
   {
      y(j) = foo(x(j),a);
   }
   RETURN_ARRAYS_DECREMENT();
   return(y);
}

dvar_matrix foo(const dvar_matrix & x, const prevariable & a)
{
   RETURN_ARRAYS_INCREMENT();
   const int i1 = x.rowmin();
   const int i2 = x.rowmax();
   dvar_matrix y;
   y.allocate(x);
   for (int i = i1; i <= i2; i++)
   {
      y(i) = foo(x(i),a);
   }
   RETURN_ARRAYS_DECREMENT();
   return(y);
}
```

# Adding API documentation with doxygen

```
/**
\file foo.cpp
Overloads of the function foo(x,a).
*/
.
.
.
/** Simple parabola; variable objects.
\param x dvar_vector of variable objects containing independant variables.
\param a dvariable for the offset of the parabola.
\return dvar_vector the elements of which are \f$(x_i-a)^2\f$
*/
dvar_vector foo(const dvar_vector & x, const prevariable & a)
{
   RETURN_ARRAYS_INCREMENT();
   const int j1 = x.indexmin();
   const int j2 = x.indexmax();
   dvar_vector y(j1,j2);
   for (int j = j1; j <= j2; j++)
   {
      y(j) = foo(x(j),a);
   }
   RETURN_ARRAYS_DECREMENT();
   return(y);
}
```

# Adding API documentation with doxygen

```
dvar_vector foo ( const dvar_vector &  x,
                  const dvariable &    a
                )
```

Simple parabola; variable objects.

**Parameters:**

- **x** dvar_vector of variable objects containing independant variables.
- **a** dvariable for the offset of the parabola.

**Returns:**

dvar_vector the elements of which are $(x_i - a)^2$

# objects.lst ... really?

```
OBJ1= \

OBJ2= \

OBJ3= \
mfexp.obj \
expm.obj \
orthpoly.obj \
makesub.obj \
fvar_a49.obj \
adpvm2.obj \
foo.obj
```

# Preliminary Testing

foo.tpl

```
GLOBALS_SECTION
  #include "foo.cpp"
```