Overloading Operators, Special `this` Pointer, `friend` Methods

Operations between Objects

- Since `Datum` represents user data we could imagine having

```
Datum d1(-3.87,0.16);
Datum d2(6.55,2.1);

Datum d3 = d1.plus(d2);

Datum d4 = d1.minus(d2);

Datum d5 = d1.product(d2);
```

- These functions are easy to implement, providing behaviour similar to doubles, ints, floats
- However, they are functions not operators: they look different from what we use to handle numbers

Operators

- C++ has a variety of built-in operators for built-in types

```
int i = 8;
int j = 10;

int l = i + j;
int k = i * j;
```

- C++ allows you to implement such built-in operators also for user-defined types (classes!)

```
Datum d1(-3.87,0.16);
Datum d2(6.55,2.1);

Datum d3 = d1 + d2;
```

- This is called **overloading of operators**
  - We need to tell the compiler what to do when adding two `Datum` objects!

C++ Operators

BINARY OPERATORS

+	+=	<<=
-	-=	==
*	*=	!=
/	/=	<=
%	%=	>=
^	^=	&&
&	&=	||
|	|=	,
>	>>	()
<	<<=	[]
=	>>=	->*

UNARY OPERATORS

+
-
*
&
->
~
!
++
- -

- Require a left and a right operand

Example of Overloaded Operator

Based on `examples/06/Overload+.cpp` and `Datum.*`

```
class Datum {
  public:
    // interface same as before

    Datum operator+( const Datum& rhs ) const;

  private:
    // same data members
};
```

```
$ g++ -o Overload+ Overload+.cpp Datum.cc
$ ./Overload+
input data d1 and d2:
datum: 1.2 +/- 0.3
datum: -0.4 +/- 0.4
output d3 = d1+d2
datum: 0.8 +/- 0.5
datum: 0.8 +/- 0.5
```

```
#include <iostream>
using namespace std;

#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  cout << "input data d1 and d2: " << endl;
  d1.print();
  d2.print();

  Datum d3 = d1 + d2;

  cout << "output d3 = d1+d2 " << endl;
  d3.print();

  Datum d4 = d1.operator+( d2 );
  d4.print();

  return 0;
}
```

```
#include "Datum.h"
#include <cmath>

// other member functions same as before

Datum Datum::operator+( const Datum& rhs ) const {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated: sum errors in quadrature
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

The Syntax of Overloading Operators

```
Datum Datum::operator+( const Datum& rhs ) const {

  // sum of central values
  double val = value_ + rhs.value_;

  // assume data are uncorrelated.
  // sum in quadrature of errors
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

Why is `operator+` constant?

- If not declared constant you cannot call it on constant objects, but adding constant objects is perfectly reasonable

Remove `const` and/or `const` and see the error if you enforce `d1` and/or `d2` to be constant

`operator+` is a member function of class `Datum`

- it returns a `Datum` object in output by value
- it has one argument called `rhs`
- it is a constant function: cannot modify the object it is applied to

Using Operators with Objects

- Operators can be called on objects exactly like any other member function of a class
  - `operator+` is called on object `d1` with argument `d2` and the returned value is stored in `d4`

```
Datum d1( 1.2, 0.3 );
Datum d2( -0.4, 0.4 );

Datum d4 = d1.operator+( d2 );
```

- However, since they are operators, they can also be used like the operators for the built-in C++ types

```
Datum d1( 1.2, 0.3 );
Datum d2( -0.4, 0.4 );

Datum d3 = d1 + d2;
```

Operator vs. Function

Based on `examples/06/OverloadSum.cpp` and `Datum.*`

```
class Datum {
  public:
    // interface same as before

    Datum operator+( const Datum& rhs ) const;
    Datum sum( const Datum& rhs ) const;


  private:
    // same data members
};
```

```
#include <iostream>
using namespace std;

#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  Datum d3 = d1 + d2;
  Datum d4 = d1.sum( d2 );
  d3.print();
  d4.print();

  return 0;
}
```

```
Datum Datum::operator+( const Datum& rhs) const {
  // sum of central values
  double val = value_ + rhs.value_;
  // assume data are uncorrelated. sum in quadrature of errors
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}


Datum Datum::sum( const Datum& rhs) const {
  // sum of central values
  double val = value_ + rhs.value_;
  // assume data are uncorrelated. sum in quadrature of errors
  double err = sqrt( error_*error_ + (rhs.error_)*(rhs.error_) );

  // result of the sum
  return Datum(val,err);
}
```

```
$ g++ -o OverloadSum OverloadSum.cpp Datum.cc
$ ./OverloadSum
datum: 0.8 +/- 0.5
datum: 0.8 +/- 0.5
```

Rules of the Game

- You **CAN** overload any of the built-in C++ operators for your classes
- Overloading operators for classes should mimic functionality of built-in operators for built-in types
  - e.g, operator `*` should not be implemented as a division
  - Purpose of overloading operators is to extend the C++ language for custom user types (classes)
    - Overload only operators that are meaningful (what would the meaning of the ++ operator be for class `Datum`?)

- You **CANNOT**
  - create new operators but only overload existing ones
  - change meaning of operators for built-in types
  - change parity of operators: a binary operator cannot be overloaded to become a unary operator

Assignment Operator

Based on `examples/06/Overload=.cpp` and `Datum.*`

```
class Datum {
  public:
    // interface same as before

    const Datum& operator=( const Datum& rhs );


  private:
    // same data members
};
```

This operator cannot be constant… We must modify the object it is applied to! What do we do?

```
#include "Datum.h"

// other member functions same as before

const Datum& Datum::operator=(const Datum& rhs)
{
  value_ = rhs.value_;
  error_ = rhs.error_;

  return *this;
}
```

Remember `this`?

```
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  const Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  Datum d3 = d1;
  d3.print();

  Datum d4;
  d4.operator=(d2);
  d4.print();

  return 0;
}
```

```
$ g++ -Wall -o Overload= Overload=.cpp Datum.cc
$ ./Overload=
datum: 1.2 +/- 0.3
datum: -0.4 +/- 0.4
```

Assignment Operator vs Copy Constructor

Based on `examples/06/Overload=.cpp` and `Datum.*`

```
class Datum {
  public:
    // interface same as before

    const Datum& operator=( const Datum& rhs );


  private:
    // same data members
};
```

```
#include "Datum.h"

// other member functions same as before

const Datum& Datum::operator=(const Datum& rhs)
{
  value_ = rhs.value_;
  error_ = rhs.error_;
  cout << "Hi, this is the operator=" << end;
  return *this;
}
```

This calls the copy constructor, even implicitly if necessary, because `d3` needs to be constructed

Interpreted as `Datum d3(d1)`

```
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  const Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  Datum d3 = d1;
  d3.print();




  return 0;
}
```

???

```
$ g++ -Wall -o Overload= Overload=.cpp Datum.cc
$ ./Overload=
datum: 1.2 +/- 0.3
```

Assignment Operator vs Copy Constructor

Based on `examples/06/Overload=.cpp` and `Datum.*`

```
class Datum {
  public:
    // interface same as before

    const Datum& operator=( const Datum& rhs );


  private:
    // same data members
};
```

```
#include "Datum.h"

// other member functions same as before

const Datum& Datum::operator=(const Datum& rhs)
{
  value_ = rhs.value_;
  error_ = rhs.error_;
  cout << "Hi, this is the operator=" << end;
  return *this;
}
```

This calls `operator=` because `d3` is **already** constructed!

The copy constructor initializes new objects, whereas the assignment operator replaces the contents of existing objects.

```
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  const Datum d1( 1.2, 0.3 );
  Datum d2( -0.4, 0.4 );

  Datum d3;
  d3 = d1;
  d3.print();




  return 0;
}
```

```
$ g++ -Wall -o Overload= Overload=.cpp Datum.cc
$ ./Overload=
Hi, this is the operator =
datum: 1.2 +/- 0.3
```

Special Pointer `this` in a Class

- Special pointer provided in C++
- Allows an object to get a **pointer to itself** from within any member function of the class
- Useful when an object (instance of a class) has to compare itself with other objects
- Particularly useful for overloading operators
  - Several operators are used to modify an object: e.g., =, +=, *=, etc.
  - All these operators should return an object of the type of the class
  - With their overloading, you want an object to modify itself AND return itself

`this`, Class Functions and `const`

- You can think of a class function as a normal function taking an implicit `this` pointer as first argument
  - `int Foo::Bar(int arg)` results in a function `int Foo_Bar(Foo* this, int arg)`
  - A call such as `Foo f; f.Bar(4)` will internally correspond to something like `Foo f; Foo_Bar(&f, 4)`
- A `const` at the end of the declaration of `Foo::Bar(int arg)` makes these
  - `int Foo_Bar(const Foo* this, int arg)`
  - `Foo f; Foo_Bar(const &f, 4)`

Assignment Operator: A Second Example

Based on `examples/06/Overload=2.cpp` and `Datum.*`

```
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  const Datum d2 = d1; // OK.. init the constant

  Datum d3( -0.2, 1.1 );
  d1 = d3; // Fine
  d2 = d3; // error!

  return 0;
}
```

```
$ g++ -Wall -o Overload=2 Overload=2.cpp Datum.ccOverload4.cpp:11:6: error: no viable overloaded '='
  d2 = d3; // error!
  ~~ ^ ~~
./Datum.h:24:18: note: candidate function not viable: 'this' argument has type 'const Datum', but method is not marked const
    const Datum& operator=( const Datum& rhs );
                 ^
1 error generated.
```

One More Example of `this`

Based on `examples/06/this.cpp`

```
#include <iostream>
#include <string>
using namespace std;

class Example {
  public:
    Example() { name_ = ""; }
    Example(const string& name);
    void printSelf() const;
  private:
    string name_;
};


Example::Example(const string& name) {
  name_ = name;
}

void Example::printSelf() const {
  cout << "name: " << name_
       << "\t this: " << this
       << endl;
}
```

```
int main() {
  Example ex1("ex1");
  ex1.printSelf();

  cout << "&ex1: " << &ex1 << endl;

  return 0;
}
```

```
$ g++ -o this this.cpp
$ ./this
name: ex1     this: 0x16d80b640
&ex1: 0x16d80b640
```

`this` is the reference of `ex1` accessible from within `ex1`

Division and Multiplication of `Datum`

Based on `examples/06/OverloadTimesDiv.cpp` and `Datum.*`

```
Datum Datum::operator*(const Datum& rhs) const {
  double val = value_*rhs.value_;

  // propagate correctly the error for x*y
  double err = sqrt( rhs.value_*rhs.value_*error_*error_ +
                     rhs.error_*rhs.error_*value_*value_ );
  return Datum(val,err);
}

Datum Datum::operator/(const Datum& rhs) const {
  double val = value_ / rhs.value_;

  // propagate correctly the error for x / y
  double err = fabs(val) * sqrt( (error_/value_)*(error_/value_) +
                           (rhs.error_/rhs.value_)*(rhs.error_/rhs.value_) );

  return Datum(val,err);
}
```

```
$ g++ -o OverloadTimesDiv OverloadTimesDiv.cpp Datum.cc
$ ./OverloadTimesDiv
datum: 1.2 +/- 0.3
datum: -3.4 +/- 0.7
datum: -4.08 +/- 1.32136
datum: -4.08 +/- 1.32136
datum: -0.352941 +/- 0.114305
datum: -2.83333 +/- 0.917613
```

```
#include <iostream>
using namespace std;

#include "Datum.h"


int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( -3.4, 0.7 );
  d1.print();
  d2.print();

  Datum d3 = d1 * d2;
  Datum d4 = d1.operator*(d2);

  d3.print();
  d4.print();

  Datum d5 = d1/d2;
  Datum d6 = d2/d1 ;
  d5.print();
  d6.print();

  return 0;
}
```

Interactions between `Datum` and `double`

Based on `examples/06/OverloadTimesDouble.cpp` and `Datum.*`

No problem: provide another overload of `operator*`

```
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d2 = d1 * 1.5;
  d2.print();

  return 0;
}
```

```
class Datum {
  public:
    // interface same as before
    Datum operator*( const Datum& rhs ) const;

    Datum operator*( const double& rhs ) const;

  private:
    // same data members
};
```

```
Datum Datum::operator*(const double& rhs) const {
  return Datum(value_*rhs,error_*rhs);
}
```

```
$ g++ -o OverloadTimesDouble OverloadTimesDouble.cpp Datum.cc
$ ./OverloadTimesDouble
datum: 1.2 +/- 0.3
datum: 1.8 +/- 0.45
```

What about `double*Datum`?

Based on `examples/06/OverloadTimesDouble2.cpp` and `Datum.*`

- No reason to limit users to multiply always in a specific order
- Not natural and certainly not intuitive
- This code does not compile with the overloading carried out so far
  - Do you understand why?
- Which operator must be overloaded?
  - `operator*` of class `Datum`?
  - `operator*` of type `double`?

```
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();

  return 0;
}
```

```
$ g++ -o OverloadTimesDouble2 OverloadTimesDouble2.cpp Datum.cc
OverloadTimesDouble2.cpp:9:18: error: invalid operands to binary expression ('double' and 'Datum')
  Datum d3 = 0.5 * d1;
             ~~~ ^ ~~
1 error generated.
```

What about `double*Datum`?

- The statement

```
double x = 0.5
Datum d3 = x * d1;
```

- is equivalent to

```
double x = 0.5
Datum d3 = x.operator*( d1 );
```

- This means that we need `operator*` of type `double` to be overloaded
- Something like

```
class double {
  public:
   Datum operator*( const Datum& rhs );
};
```

is not allowed though!

  - We cannot overload operators for built-in types!

- Defining a new type `MyDouble` is not a practical solution

A New Global Function

- We could define a global function to do this
  - Declaration in header file **outside class scope**
  - Implementation in source file
- It works but not as natural to use

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
Datum productDoubleDatum(const double& lhs, const Datum& rhs);
#endif
```

```
#include "Datum.h"
// implement all member functions

// global function!
Datum productDoubleDatum(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value(), lhs*rhs.error() );
}
```

```
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = productDoubleDatum(0.5, d1);
  d3.print();

  return 0;
}
```

Overloading Operators as Global Function

Based on `examples/06/OverloadTimesDouble2.cpp` and `Datum.*`

Define a global operator to do exactly what we need

- Declaration in header file **outside class scope**
- Implementation in source file. No scope operator needed
  - Not a member function

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
Datum operator*(const double& lhs, const Datum& rhs);
#endif
```

```
$ g++ -o OverloadTimesDouble2 OverloadTimesDouble2.cpp Datum.cc
$./OverloadTimesDouble2
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
```

```
// global function!
Datum operator*(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value(), lhs*rhs.error() );
}
```

```
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();

  return 0;
}
```

Another Example: Overloading `operator<<()`

Based on `examples/06/OverloadInsertion.cpp` and `Datum.*`

```
#ifndef Datum_h
#define Datum_h
// Datum.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
  // the rest of the class

};
//other global functions
ostream& operator<<(ostream& os, const Datum& rhs);
#endif
```

```
#include "Datum.h"
// implement all member functions
// implement all global functions
ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << rhs.value() << " +/- "
     << rhs.error();
  return os;
}
```

```
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  d1.print();

  Datum d3 = 0.5 * d1;
  d3.print();
  cout << d3 << endl;

  return 0;
}
```

```
$ g++ -o OverloadInsertion OverloadInsertion.cpp Datum.cc
$./OverloadInsertion
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
0.6 +/- 0.15
```

Overloading Boolean `operator<`

Based on `examples/06/OverloadBool.cpp` and `Datum.*`

```
class Datum {
  public:

    bool operator<(const Datum& rhs) const;

   //...
}
```

```
bool Datum::operator<(const Datum& rhs) const {
  return ( value_ < rhs.value_ );
}
```

Return type is boolean: constant method since does not modify the object being applied to

Do you agree that `error_` should not affect the comparison?

```
int main() {
  Datum d1( 1.2, 0.3 );
  Datum d3( -0.2, 1.1 );
  cout << "d1: " << d1 << endl;
  cout << "d3: " << d3 << endl;

  if( d1 < d3 ) {
    cout << "d1 < d3" << endl;
  } else {
    cout << "d3 < d1" << endl;
  }

  return 0;
}
```

```
$ g++ -o OverloadBool OverloadBool.cpp datum.cc
$./OverloadBool
d1: 1.2 +/- 0.3
d3: -0.2 +/- 1.1
d3 < d1
```

Overhead of Operator Overloading with Global Functions

```
ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << rhs.value() << " +/- "
     << rhs.error();
  return os;
}
```

```
Datum operator*(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value(), lhs*rhs.error() );
}
```

- Global functions do not have access to private members of class
- Necessary to call public methods to access information
  - Two calls per `cout` or simple product
- Overhead of calling functions can become significant if a frequently used operator is overloaded via global functions

`friend` Methods

Based on `examples/06/OverloadInsertion.cpp` and `DatumNew.*`

```
#ifndef DatumNew_h
#define DatumNew_h
// DatumNew.h
#include <iostream>
using namespace std;

class Datum {
  public:
    Datum();
   // … other methods

    const Datum& operator=( const Datum& rhs );
    bool operator<(const Datum& rhs) const;

    Datum operator*( const Datum& rhs ) const;
    Datum operator/( const Datum& rhs ) const;

    Datum operator*( const double& rhs ) const;

    friend Datum operator*(const double& lhs, const Datum& rhs);
    friend ostream& operator<<(ostream& os, const Datum& rhs);

  private:
    double value_;
    double error_;
};
#endif
```

Global methods declared `friend` within a class can access private members without being member functions

```
$ g++ -o OverloadInsertion OverloadInsertion.cpp DatumNew.cc
$./OverloadInsertion
datum: 1.2 +/- 0.3
datum: 0.6 +/- 0.15
0.6 +/- 0.15
```

```
// DatumNew.cc
#include "DatumNew.h"
// implement all member functions

// global functions
Datum operator*(const double& lhs, const Datum& rhs){
  return Datum(lhs*rhs.value_, lhs*rhs.error_);
}

ostream& operator<<(ostream& os, const Datum& rhs){
  using namespace std;
  os << "Datum: " << rhs.value_ << " +/- "
      << rhs.error_; // NB: no endl!
  return os;
}
```

Overloading `operator+=()`

Based on `examples/06/Overload+=.cpp` and `Datum.*`

- Overloading =, +, and < separately does not automatically imply anything for, e.g., += and <=

```
class Datum {
  //...
  Datum operator+( const Datum& rhs ) const;
  const Datum& operator+=( const Datum& rhs );
  //...
};
```

```
const Datum& Datum::operator+=(const Datum& rhs) {
  value_ += rhs.value_;
  error_ = sqrt( rhs.error_*rhs.error_ + error_*error_ );
  return *this;
}
```

```
#include <iostream>
using namespace std;
#include "Datum.h"

int main() {
  Datum d1( 1.2, 0.3 );
  Datum d2( 3.1, 0.4 );

  cout << "d1: " << d1 << "\t d2: " << d2 << endl;

  d1 += d2;
  cout << "d1+d2 = " << d1 << endl;

  return 0;
}
```

```
$ g++ -o Overload+= Overload+=.cpp Datum.cc
$./Overload+=
d1: 1.2 +/- 0.3             d2: 3.1 +/- 0.4
d1+d2 = 4.3 +/- 0.5
```

The Problem with Returning by-value

Based on `examples/06/FooApp.cpp`

```
class Foo {

public:
  Foo() { name_ = ""; x_= 0; }
  Foo(const std::string& name, const double x) { name_ = name; x_ = x; }
  double value() const { return x_; }
  std::string name() const { return name_; }

  Foo operator=(const Foo& rhs) {
    Foo aFoo(rhs.name_,rhs.x_);
    cout << "--> In Foo::operator=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

  Foo operator+=(const Foo& rhs) {
    Foo aFoo(std::string(name_+"+"+rhs.name_), x_ + rhs.x_);
    cout << "--> In Foo::operator+=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

//Continues...
```

```
private:
  double x_;
  std::string name_;
};

// global functions
ostream&  operator<<(ostream& os, const Foo& foo) {
  os << "Foo name: "  << foo.name() << " value: " << foo.value()
     << " address: " << &foo;
  return os;
}

int main() {
  Foo f1("f1",1.), f2(“f2",2.), f3("f3",3.);

  cout << "Before f1+=f2 " << endl;
  f1 += f2;
  cout << "After f1+=f2\n" << f1 << endl;

  cout << "Before f1 = f3 " << endl;
  f1 = f3;
  cout << "After f1 = f3\n" << f1 << endl;

  return 0;
}
```

The Problem with Returning by-value

Based on `examples/06/FooApp.cpp`

```
class Foo {

public:
  Foo() { name_ = ""; x_= 0; }
  Foo(const std::string& name, const double x) { name_ = name; x_ = x; }
  double value() const { return x_; }
  std::string name() const { return name_; }

  Foo operator=(const Foo& rhs) {
    Foo aFoo(rhs.name_,rhs.x_);
    cout << "--> In Foo::operator=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

  Foo operator+=(const Foo& rhs) {
    Foo aFoo(std::string(name_+"+"+rhs.name_), x_ + rhs.x_);
    cout << "--> In Foo::operator+=: value: " << aFoo.value()
         << ", name: " << aFoo.name() << ", &aFoo: " << &aFoo
         << endl;
    return aFoo;
  }

//Continues...
```

```
pri
  d
  s
};

//
ost
  o

  r
}
```

Assignment never happens! The left-hand-side is never modified by the operators!

If you do not return by reference you implicitly invoke copy constructors, temporary copies, etc.

Returning a simple non-const reference works. Try it!

```
$ g++ -o FooApp FooApp.cpp
$./FooApp
Before f1+=f2
--> In Foo::operator+=: value: 3, name: f1+f2, &aFoo: 0x16f077580
After f1+=f2
Foo name: f1 value: 1 address: 0x16f077638
Before f1 = f3
--> In Foo::operator=: value: 3, name: f3, &aFoo: 0x16f077560
After f1 = f3
Foo name: f1 value: 1 address: 0x16f077638
```

Enumerators

Enumerators

- **Enumerators** are set of integers referred to by identifiers
- There is natural need for enumerators in programming
  - Months: Jan, Feb, Mar, …, Dec
  - Status: Successful, Failed, Problems, Converged
  - Shapes: Circle, Square, Rectangle, …
  - Colours: Red, Blue, Black, Green, …
  - Coordinate system: Cartesian, Polar, Cylindrical
  - Wave Polarization: Transverse, Longitudinal, Tensor, Scalar, Vector, +, ×
- Enumerators make the code more user friendly
  - Easier to understand human identifiers instead of hardwired numbers in your code!
- You can redefine the value associated to an identifier without changing your code

Enumerators: Example 1

Based on `examples/06/enum1.cpp`

```
#include <iostream>
using namespace std;

int main() {
  enum FitStatus { Successful, Failed, Converged };

  FitStatus status;

  status = Successful;
  cout << "Status: " << status << endl;

  status = Converged;
  cout << "Status: " << status << endl;

  return 0;
}
```

- Do not forget this
- By default the first identifier is assigned value 0
- Enums can be used as integers but not vice versa

```
$ g++ -o enum1 enum1.cpp
$ ./enum1
Status: 0
Status: 2
```

Enumerators: Example 2

Based on `examples/06/enum2.cpp`

```
#include <iostream>
using namespace std;

int main() {
  enum Colour { Red=1, Blue=45, Yellow=17, Black=342 };

  Colour col;

  col = Red;
  cout << "Colour: " << col << endl;

  col = Black;
  cout << "Colour: " << col << endl;


  return 0;
}
```

- You can use arbitrary integer values for each identifier

```
$ g++ -o enum2 enum2.cpp
$ ./enum1
Colour: 1
Colour: 342
```

Common Errors with Enumuration

Based on `examples/06/enum_bad.cpp`

```
#include <iostream>
using namespace std;

int main() {
  enum Colour { Red=1, Blue=45, Yellow=17, Black=342 };

  Colour col;

  col = Red;
  cout << "Colour: " << col << endl;

  col = Black;
  cout << "Colour: " << col << endl;

  col = 45; //assign int to enum

  int i = Red;

  return 0;
}
```

- You cannot assign an `int` to an `enum`
- But you can assign an `enum` to an `int`

```
$ g++ -o enum_bad enum_bad.cpp
enum3.cpp:15:9: error: assigning to 'Colour' from incompatible type 'int'
  col = 45; //assign int to enum
        ^~
1 error generated.
```

Enumeration in Classes

Based on `examples/06/enum3.cpp` and `Fitter.h`

- Using **`public`** enumerators with complete qualifier to namespace and class

```
#ifndef Fitter_h_
#define Fitter_h_

namespace analysis {
  class Fitter {
    public:
      enum Status { Successful=0, Failed, Problems };

      Fitter() { };

      Status fit() {
        return Successful;
      }
    private:
  }; //class Fitter
} //namespace analysis

#endif
```

```
$ g++ -o enum3 enum3.cpp
fit successful!
```

```
#include "Fitter.h"
#include <iostream>
using namespace std;

int main() {

  analysis::Fitter myFitter;

  analysis::Fitter::Status stat = myFitter.fit();

  if( stat == analysis::Fitter::Successful ) {
    cout << "fit successful!" << endl;
  } else {
    cout << "Fit had problems ... status = "
         << stat << endl;
  }

  return 0;
}
```

Enumerators and Strings

Based on `examples/06/Colour.cpp`

```
#include <iostream>
#include <map>
using std::cout;
using std::endl;

int main(){
  enum Colour { Red=1, Blue=45, Yellow=17, Black=342 };

  Colour col;

  // new map with key: integer     value: string
  std::map<int, std::string> colname;

  colname[Red] = std::string("Red");
  colname[Black] = std::string("Black");

  col = Red;
  cout << "Colour int: " << col << endl;
  cout << "Colour name: " << colname[col] << endl;

  return 0;
}
```

- No automatic conversion from enumeration to strings
- You can use vectors of strings or `std::map` to assign string names to enumeration states

```
$ g++ -o Colour Colour.cpp
$ ./Colour
Colour int: 1
Colour name: Red
```