Classes

B&N: "Scientific and engineering problems are rarely posed directly in terms of the computer's intrinsic types: bits, bytes, integers and floating point numbers"

Shocking statement?

In a detector's tracking code, for example, the problem is posed in terms of…

- tracks
- points
- list of points
- chamber
- cylinders
- layers

C++ with its mechanism of *classes* allows defining new types and the operations on these types

When we do object-oriented programming with C++ we will be writing and using classes

Examples from CLHEP

Class Library for High Energy Physics

Why?

- Provide some classes are specific to HEP
- Encourage code sharing between experiments and between experimentalists and theorists.
- Reduce redundant work

Who?

- started by Leif Lönnblad, Nordita (via CERN, DESY and Lund)
- now maintained by committee

Use

- `http://proj-clhep.web.cern.ch/proj-clhep/`
- `will show version 1.4`
- `current is 2.0`

ThreeVector

CLHEP**'s ThreeVector class (simplified)**

```
class Hep3Vector {
public:
  Hep3Vector();
  Hep3Vector(double x, double y, double z);
  Hep3Vector(const Hep3Vector &v);
  double x();
  double y();
  double z();
  double phi();
  double cosTheta();
  double mag();
  // much more not shown
private:
  double dx, dy, dz;
};
```

- this is the declaration in the header file
- keyword `class` starts the declaration which is contained within the `{}`
- class contains member functions
- an object can be an instance of a class
- an object of a class contains data members

Using a class object

Consider

```
#include <iostream>
#include <CLHEP/ThreeVector.h>
using namespace std;

int main() {
  double x, y, z;

  while ( cin >> x >> y >> z ) {
    Hep3Vector aVec(x, y, z);

    cout << "r: " << aVec.mag();
    cout << "  phi: " << aVec.phi();
    cout << "  cos(theta): " << aVec.cosTheta() << endl;
  }
  return 0;
}
```

- `Hep3Vector aVec(x, y, z);` declares `aVec`, a object of type `Hep3Vector` and initializes it
- `aVec.mag()` calls the member function `mag()` of the object
- the “`.`” is the *class member access operator*
- use “`->`” access operator when one has pointer to object:

Data members

Look again

```
class Hep3Vector {
public:
  // member functions

private:
  double dx, dy, dz;
};
```

- `Hep3Vector` contains 3 data members
- declaration is like any other except no initializers are allowed
- every instance of the class `Hep3Vector` will have its own 3 data members.

```
Hep3Vector x(1.0, 0.0, 0.0);
Hep3Vector y(0.0, 1.0, 0.0);
Hep3Vector z(0.0, 0.0, 1.0);
```

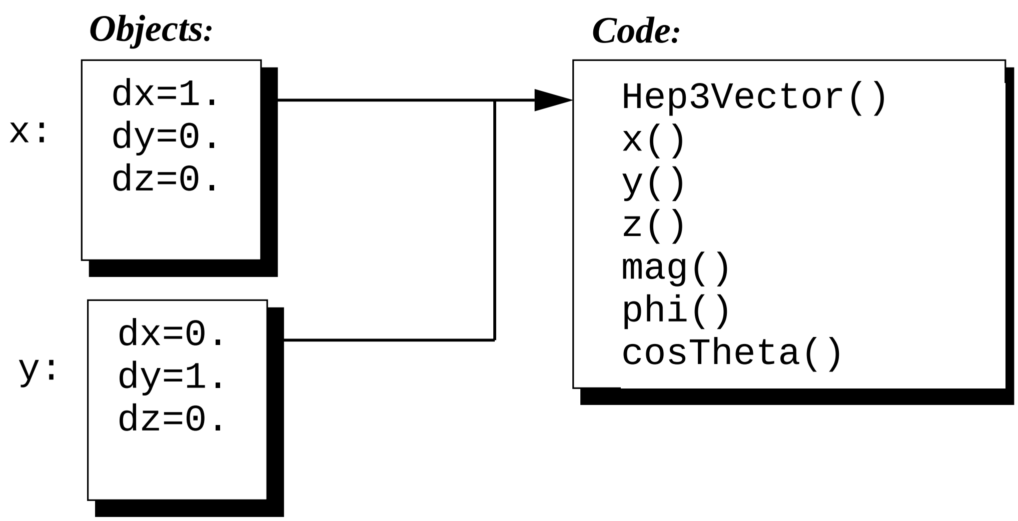
- `Hep3Vector` is a type
- an object of type `Hep3Vector` has a value (or state) that is represented by the values of its data members (like a complex number)
- the size of a `Hep3Vector` object is likely to be `3*sizeof(double)`

Memory model

Consider

```
Hep3Vector x(1.0, 0.0, 0.0);
Hep3Vector y(0.0, 1.0, 0.0);
```

In computer's memory we have



- an object is an instance of a class (type)
- each object has its own data members
- one copy of the code for a class is shared by all instances of the class
- hidden argument `this` is how it all works

Use of `private` keyword

We have

```
class Hep3Vector {
public:
  double mag();
  double x();
  double dummy;
  // member functions

private:
  double dx, dy, dz;
};
```

- the following compiles

```
Hep3Vector x(1.0, 0.0, 0.0);
cout << x.dummy;
```

- the following does not compile

```
Hep3Vector x(1.0, 0.0, 0.0);
cout << x.dx;  // WRONG
```

- this is called *data hiding*
- by disallowing direct access, you hide how data is stored.
- one can change how data is stored without breaking user code because you disallowed direct access

Initializing a class object

At least 3 ways we would like to initialize an object

- no initial value

```
Hep3Vector a;
```

- with three `double` values

```
Hep3Vector a(1.0, 1.0, 1.0);
```

- copy of another object

```
Hep3Vector a(1.0, 1.0, 0.0);
Hep3Vector b = a;
```

- each calls a special member function called a *constructor*

There are three constructors in the class

```
class Hep3Vector {
public:
  Hep3Vector();
  Hep3Vector(double x, double y, double z);
  Hep3Vector(const Hep3Vector &v);
  // much more not shown
private:
  double dx, dy, dz;
};
```

Constructor Implementations

The constructor member functions

```
Hep3Vector::Hep3Vector(double x, double y, double z) {
  dx = x;
  dy = y;
  dz = z;
}

Hep3Vector::Hep3Vector(const Hep3Vector &vec) {
  dx = vec.dx;
  dy = vec.dy;
  dz = vec.dz;
}

Hep3Vector::Hep3Vector(){
}
```

- called after memory space has been allocated
- when the class name and member name are the same, then the member function is a constructor
- `Foo::bar()` says that `bar()` is a member function of the class `Foo`
- `::` is the *scope resolution operator*
- note that copy constructor uses a const reference

Data Hiding

Violation of private parts?

```
Hep3Vector::Hep3Vector(const Hep3Vector &vec) {
  dx = vec.dx;
  dy = vec.dy;
  dz = vec.dz;
}
```

- objects of the same class have access to private data members
- the purpose of data hiding is to hide implementation from other classes
- can’t hide implementation from object of same class
- `const` qualifier says we wouldn’t change argument

Access member functions

The declaration was

```
class Hep3Vector {
public:
  double x();
  double y();
  double z();
  // much more not shown
private:
  double dx, dy, dz;
};
```

The implementation is

```
double Hep3Vector::x() {
  return dx;
}
double Hep3Vector::y() {
  return dy;
}
double Hep3Vector::z() {
  return dz;
}
```

- inefficient?
- make function in-line
- always ask: “do I want the data to do some work or do I want the object to do the work”

Inline access member functions

Change declaration to

```
inline double Hep3Vector::x() {
  return dx;
}
inline double Hep3Vector::y() {
  return dy;
}
inline double Hep3Vector::z() {
  return dz;
}
```

- can be used when execution of function body is shorter than time to call and return from function
- any decent compiler should produce inline code instead of function call for above
- inline keyword is just a hint, however
- data hiding is preserved
- implementation needs to be in the header file
- sometimes put in file with `.icc` suffix that is included by the header file (not BaBar practice)
- program could be faster
- program could be larger

More Implementation

Recall

```
class Hep3Vector {
public:
  double mag();
  double phi();
  double cosTheta();
  // much more not shown
private:
  double dx, dy, dz;
};
```

Implementation

```
inline double Hep3Vector::mag() {
  return sqrt(dx*dx + dy*dy + dz*dz);
}

inline double Hep3Vector::phi() {
  return dx == 0.0 && dy == 0.0 ? 0.0 : atan2(dy,dx);
}

inline double Hep3Vector::cosTheta() {
  double ptot = mag();
  return ptot == 0.0 ? 1.0 : dz/ptot;
}
```

- note how object calls its own member function
- examples of letting object do the work

Design decisions

Fortran style

```
common/points/hits(3,100)
real*4        hits
real*4 x, y, z, r
! do some work
x = hits(1,i) ! or from ZEBRA bank
y = hits(2,i)
z = hits(3,i)
r = sqrt(x*x + y*y + z*z);
```

Another Fortran style

```
common/points/hits(3,100)
real*4        hits
real*4 x, y, z, r
! do some work
x = hits(1,i)
y = hits(2,i)
z = hits(3,i)
r = mag(x, y, z) ! or mag(hits(1,i))
```

Mark II VECSUB style

```
common/points/hits(3,100)
real*4 r
! do some work
r = hitsmag(i)
```

C++ design

C++ style

```
Hep3Vector hits[100];
// do some work
double r = hits[i].mag();
```

- efficient with inline functions
- don't need knowledge of data structure
- modular
- re-usable
- later, we'll get rid of the fixed or dynamic arrays

Homework

Suppose

```
class Hep3Vector {
public:
  Hep3Vector();
  Hep3Vector(double x, double y, double z);
  Hep3Vector(const Hep3Vector &v);
  inline double x();
  inline double y();
  inline double z();
  inline double phi();
  inline double cosTheta();
  inline double mag();
private:
  double r, cos_theta, phi;
};
```

- write the implementation for this class
- constructors take x, y, and z as arguments, but must intialize r, cos(theta), and phi data members
- try test program shown before, it should still work with this small change

```
// #include <CLHEP/ThreeVector.h>
#include “ThreeVector.h”
```

- write a program to exercise `x()`, `y()`, and `z()` member functions

Another look at `Hep3Vector`

We'll now look at the real `Hep3Vector` class and explain those new language elements we need to understand it

```
class Hep3Vector {
public:
  Hep3Vector(double x=0., double y=0., double z=0.);
  Hep3Vector(const Hep3Vector&);
  double x() const;
  double y() const;
  double z() const;
  double phi() const;
  double cosTheta() const;
  double mag() const;
  // much more not shown
private:
  double dx, dy, dz;
};
```

- uses default arguments
- `const` keyword after function means no data member of the object will be changed by invoking function
- this `const` is enforced when compiling the class
- the above are obvious, but it will be less obvious with other classes in the future

Initializing syntax

Two forms to invoke copy constructor

```
Hep3Vector x(1.0, 0.0, 0.0);
Hep3Vector y = x;  // C style
Hep3Vector y(x);   // C++ class style
```

- the two are equivalent if argument is same type as object being declared
- both invoke copy constructor
- the `=` form allows user defined conversions when argument is not same type
- both forms allowed for built-in type

Consider

```
Hep3Vector x = 1.0;
```

- might be equivalent to

```
Hep3Vector tmp(1.0);
Hep3Vector x = tmp;
```

- but following has no suprises

```
Hep3Vector x(1.0);
```

Member Initializers

The constructor can be implemented like any other member function…

```
Hep3Vector::Hep3Vector(double x, double y, double z){
  dx = x;
  dy = y;
  dz = z;
}
```

- but data members need to be constructed before assignment
- for `Hep3Vector` the custom constructor would be called

An alternate form is use of member initializers

```
Hep3Vector::Hep3Vector(double x, double y, double z) :
    dx(x), dy(y), dz(z){}
```

- note the `:` preceding the opening `{`
- `dx(x)` notation calls a constructor directly
- which constructor depends on argument matching
- in the above case, it is the copy constructor
- the function body is required, even if empty

Function Return Types

A function returns a temporary hidden variable that is initialized by the return statement

Consider

```
float f() {
    return 1;
}
float x;
// ...
x = f();
```

- it is as if

```
float tmp = 1;
x = tmp;
```

Consider

```
float & Vector3::x() {
    return dx;
}
Vector3 vec;
// ...
vec.x() = 1.0; // uh?
```

- it is as if

```
float &tmp = vec.dx;
tmp = 1.0;
```

Operators are functions?

Operators can be thought of as functions

```
double add( double a, double b) {
    return a + b;
}
double x, y, z;
//
z = x + y;
z = add(x, y);
```

- `add()` operates on two arguments and returns a result
- the symbol `+` operates on two operands and returns a result

Use of mathematical symbols is more concise and easier to read

```
double add( double a, double b);
double mul( double a, double b);
double a, b, x, y, z;
//
z = add(mul(a, x), mul(b,y));
z = a*x + b*y;
```

C, C++, and Fortran all define operators for built-in types

Operator Functions

An operator function in `Hep3Vector`

```
class Hep3Vector {
public:
  inline Hep3Vector& operator +=(const Hep3Vector &);
  // more not shown
```

- the name of the function is the word `operator` followed by the operator symbol
- this function is called when

```
Hep3Vector p, q;
//
q += p;
```

- the function is invoked on `q`; the left-hand side
- the argument will be `p`; the right-hand side
- `q += p;` is shorthand for `q.operator+=(p);`
- the function returns a `Hep3Vector` reference for consistency with built-in types

```
Hep3Vector p, q, r;
//
r = q += p;
// r.operator=( q.operator+=(p) )
```

Operator Function Implementation

Implementation

```
inline Hep3Vector& Hep3Vector::operator+=(const Hep3Vector& p) {
  dx += p.x();  // could have been dx += p.dx
  dy += p.y();
  dz += p.z();
  return *this;
}
```

- does the accumulation as one would expect
- `this` is a hidden argument that is a pointer to the object’s own self
- `this->dx` is thus equivalent to `dx`
- remember: use `->` instead of `.` when you have a pointer
- or `dx` is shorthand for `this->dx`
- recall that `Hep3Vector::x()` is an in-line function itself
- `return *this` returns the address of the object, thus the reference

Compare Fortran and C++

Fortran vector sum

```
real p(3), q(3)
! ...
q(1) = q(1) + p(1)
q(2) = q(2) + p(2)
q(3) = q(3) + p(3)
```

C++ vector sum

```
Hep3Vector p, q;
// ...
q += p;
```

Operator Functions

Essentially all operators can be used for user defined types except `“.”`, `“.*”`, `“::”`, `“sizeof”` and `“?:”`

Can not define new ones

- sorry, can’t do `operator**()` for exponentiation
- and there’s no operator one could use with the correct precedence
- can’t overload operators for built-in types

One should only use when conventional meaning makes sense

```
Hep3Vector p, q;
double z;
// ………
z = p*q; // uh?
```

- is this cross product or dot product?
- `Hep3Vector` defines it to be dot product

Non-member Operator Function

Consider

```
inline Hep3Vector operator*(const Hep3Vector& p, double a) {
  Hep3Vector q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
```

- invoked by

```
double scale = 3.0;
Hep3Vector p(1.0);  // unit vector along x axis
Hep3Vector r(0.0, 1,0);
r += p*scale;
```

- note return by value
- need a new object whose value is `x*scale`
- the temporary object is used as argument to `operator+=()` and then discarded
- such temporary objects are generated by Fortran as well

```
real scale, p(3), r(3)
r(1) = r(1) + p(1)*scale
r(2) = r(2) + p(2)*scale
r(3) = r(3) + p(3)*scale
```

Need Symmetric Operator Functions

CLHEP has

```
inline Hep3Vector operator*(const Hep3Vector& p, double a) {
  Hep3Vector q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
inline Hep3Vector operator*(double a, const Hep3Vector& p) {
  Hep3Vector q( a*p.x(), a*p.y(), a*p.z() );
  return q;
}
```

- second one invoked by

```
double scale = 3.0;
Hep3Vector p(1.0);  // unit vector along x axis
Hep3Vector q(0.0, 1,0);
q += scale*p;
```

- argument matching applies
- must use global function because `scale.operator*(p)` doesn’t exist

The Complete List - I

Constructors

```
Hep3Vector(double x=0.0, double y=0.0, double z=0.0);
Hep3Vector(const Hep3Vector &);
```

- also contains conversion constructor

Destructor

```
~Hep3Vector();
```

- invoked when object is deleted (more next session)

Accessor-like functions

```
inline double x() const;
inline double y() const;
inline double z() const;
inline double mag() const;
inline double mag2() const;
inline double perp() const;
inline double perp2() const;
inline double phi() const;
inline double cosTheta() const;
inline double theta() const;
inline double angle(const Hep3Vector &) const;
inline double perp(const Hep3Vector &) const;
inline double perp2(const Hep3Vector &) const;
```

The Complete List - II

Manipulators

```
void rotateX(double);
void rotateY(double);
void rotateZ(double);
void rotate(double angle, const Hep3Vector & axis);
Hep3Vector & operator *= (const HepRotation &);
Hep3Vector & transform(const HepRotation &);
```

Set functions

```
inline void setX(double);
inline void setY(double);
inline void setZ(double);
inline void setMag(double);
inline void setTheta(double);
inline void setPhi(double);
```

Output function

```
ostream & operator << (ostream &, const Hep3Vector &);
```

- allows

```
Hep3Vector x(1.0);
// ...
cout << x << endl;
```

The Complete List - III

Vector algebra member functions

```
inline double dot(const Hep3Vector &) const;
inline Hep3Vector cross(const Hep3Vector &) const;
inline Hep3Vector unit() const;
inline Hep3Vector operator - () const;
```

Vector algebra non-member functions

```
Hep3Vector operator+(const Hep3Vector&, const Hep3Vector&);
Hep3Vector operator-(const Hep3Vector&, const Hep3Vector&);
double operator * (const Hep3Vector &, const Hep3Vector &);
Hep3Vector operator * (const Hep3Vector &, double a);
Hep3Vector operator * (double a, const Hep3Vector &);
```

Assignment operators

```
inline Hep3Vector & operator = (const Hep3Vector &);
inline Hep3Vector & operator += (const Hep3Vector &);
inline Hep3Vector & operator -= (const Hep3Vector &);
inline Hep3Vector & operator *= (double);
```

Summary

`Hep3Vector` implements vector algebra

It was long and tedious to implement

Now that we have it (thank you, Leif and Anders), we can use it and never have to expand these details in our own code

Besides objects of type `int`, `float`, and `double`, we can use operators with objects of type `Hep3Vector`

We have a new type with higher level of abstraction

Levels of Abstraction in Physics

Do you recognize these equations?

$$\sum_i \frac{\partial E_i}{\partial x_i} = \frac{\partial E_x}{\partial x} + \frac{\partial E_y}{\partial y} + \frac{\partial E_z}{\partial z} = 4\pi\rho$$

$$\sum_i \frac{\partial B_i}{\partial x_i} = \frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y} + \frac{\partial B_z}{\partial z} = 0$$

$$\sum_i \varepsilon_{ijk} \frac{\partial}{\partial x_j} E^k = -\frac{1}{c}\frac{\partial B_i}{\partial t}$$

$$\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z} = -\frac{1}{c}\frac{\partial B_x}{\partial t}$$

$$\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x} = -\frac{1}{c}\frac{\partial B_y}{\partial t}$$

$$\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y} = -\frac{1}{c}\frac{\partial B_z}{\partial t}$$

Higher Level of Abstraction

Now do you recognize them?

$$\vec{\nabla} \bullet \mathbf{E} = 4\pi\rho$$

$$\vec{\nabla} \times \mathbf{B} = \frac{4\pi}{c}\mathbf{J} + \frac{1}{c}\frac{\partial \mathbf{E}}{\partial t}$$

$$\vec{\nabla} \bullet \mathbf{B} = 0$$

$$\vec{\nabla} \times \mathbf{E} = -\frac{1}{c}\frac{\partial \mathbf{B}}{\partial t}$$

or even

$$\partial_\alpha F^{\alpha\beta} = \frac{4\pi}{c} J^\beta$$

$$\frac{1}{2}\varepsilon^{\alpha\beta\gamma\delta}\partial_\alpha F_{\gamma\delta} = 0 = \partial^\alpha F^{\beta\gamma} + \partial^\beta F^{\gamma\alpha} + \partial^\gamma F^{\alpha\beta}$$

To advance in physics/math, we need higher levels of abstractions, else we get lost in implementation details

C++ allows higher level of abstract as well