Plan of the day

Few more language features

Particle data table

Polymorphic inheritance

Enumerations

mnemonic names for integer codes grouped into sets

```
enum Color { red, orange, yellow, green, blue, indigo, violet };

Color c = green;

enum Polygon { triangle = 3, quadrilateral, pentagon };
```

- `Color` is programmer defined type
- `red`, `orange`, *etc* are constants of type `Color`
- `c` is declared as type `Color` with inital value of `green`
- `c` can change, but `red`, `orange` *etc* can not
- `enum` values are converted to int when used in arithmetic or logical operations
- default integer values start at 0 and increment by 1
- can override the default.
- but valued stored in variable which is an enumerated type is limited to the values of the `enum`
- uniqueness of the enumerated values is guaranteed
- slightly different from C

PdtLund Class

Extract from this class

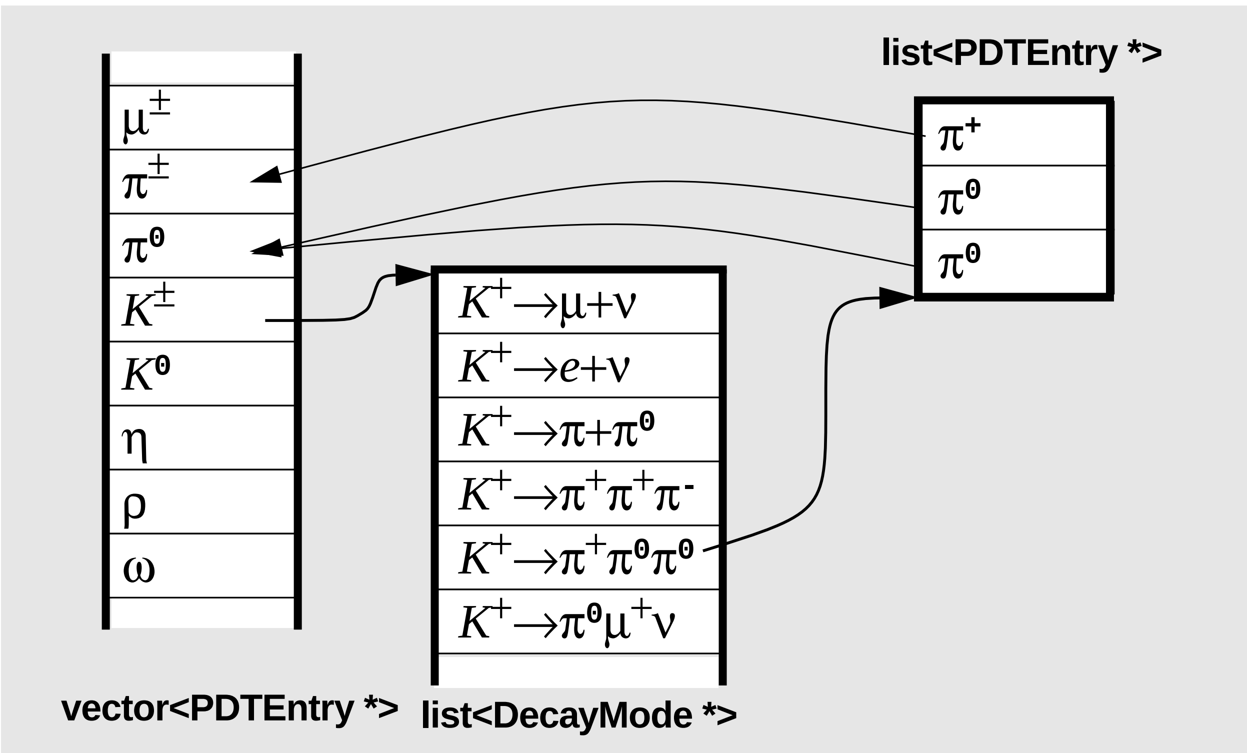
```
class PdtLund
{
public:
// a list of common particles
// the numbers are PDG standard particle codes
  enum Type {
    e_minus = 11, nu_e, mu_minus, nu_mu,
    e_plus = -11, nu_e_bar = -12
// many more not shown
  };
};
```

- `enum` nested in class
- must use scoping to access outside of class

  ```
  PdtLund::Type t= PdtLund::e_minus;
  ```

- the scoping helps the readability and avoids name conflicts
- scope type and constants

Layout



- `Pdt` has one data member: `vector<PdtEntry*> s_entries`
- `PdtEntry` has data members for particle properties and `list<DecayMode *>` for list of decay modes
- `DecayMode` has data members for branching fraction and an `list<PdtEntry *>` for list of children.

`static` keyword

Part of the Pdt class declaratin

```
class Pdt
{
public:
  // return entry pointer given particle id or name
  static PdtEntry* lookup(const char *name);
  static PdtEntry* lookup(PdtLund::Type id);
  static PdtEntry* lookup(PdtGeant::Type id);
  static float mass(PdtLund::Type id);
  static float mass(PdtGeant::Type id);
  static float mass(const char* name);
// more not shown
private:
  static std::vector<PdtEntry *> s_entries;
};
```

- a `static` data member is one that is shared by all instances of the class, *e.g.* a global within the scope of the class
- a `static` member function is one that is global within the scope of the class
- access a data member or member function with scope operator

```
    mass = Pdt::mass( PdtLund::pi_plus);
```

`PDTEntry` class

Parts of the header file

```
class DecayMode;

class PdtEntry {
public:
  inline const char *name() const {return m_name;}
  inline float charge() const {return m_charge;}
  inline float mass() const {return m_mass;}
  inline float width() const {return m_width;}
// more not shown
protected:
  char *m_name;
  float m_mass;     // nominal mass (GeV)
  float m_width;    // width (0 if stable) (GeV)
  float m_lifeTime; // c*tau, (cm)
  float m_spin;     // spin, in units of hbar
  float m_charge;   // charge, in units of e
  float m_widthCut; // used to limit range of B-W
  float m_sumBR;    // total branching ratio
  std::list<DecayMode *> m_decayList;
  PdtLund::Type m_lundid;
  PdtGeant::Type m_geantid;
};
```

- note forward declaration of class

DecayMode class

From the header file

```
class DecayMode {
public:
  DecayMode ( float bf,
              const list<PdtEntry *> & l );

  inline float BF() const
  {
    return m_branchingFraction;
  }

  inline const vector<PDTEntry *> & childList() const
  {
     return m_children;
  }

private:

  float  m_branchingFraction;
  std::list<PdtEntry *> m_children;

};
```

- nothing new

Detector Simulation

What classes are involved?

- 3-vector
- geometry
- track
- detectors
- fields
- *etc*

Will take examples from Gismo project

- C++ framework for detector simulation and reconstruction;
- we'll see how it differs from the Fortran *black box* approach, *e.g.* GEANT 3

Gismo History

Version 0, the prototype

- written by Bill Atwood (SLAC) and Toby Burnett (U Washington)
- completed in Spring 1991

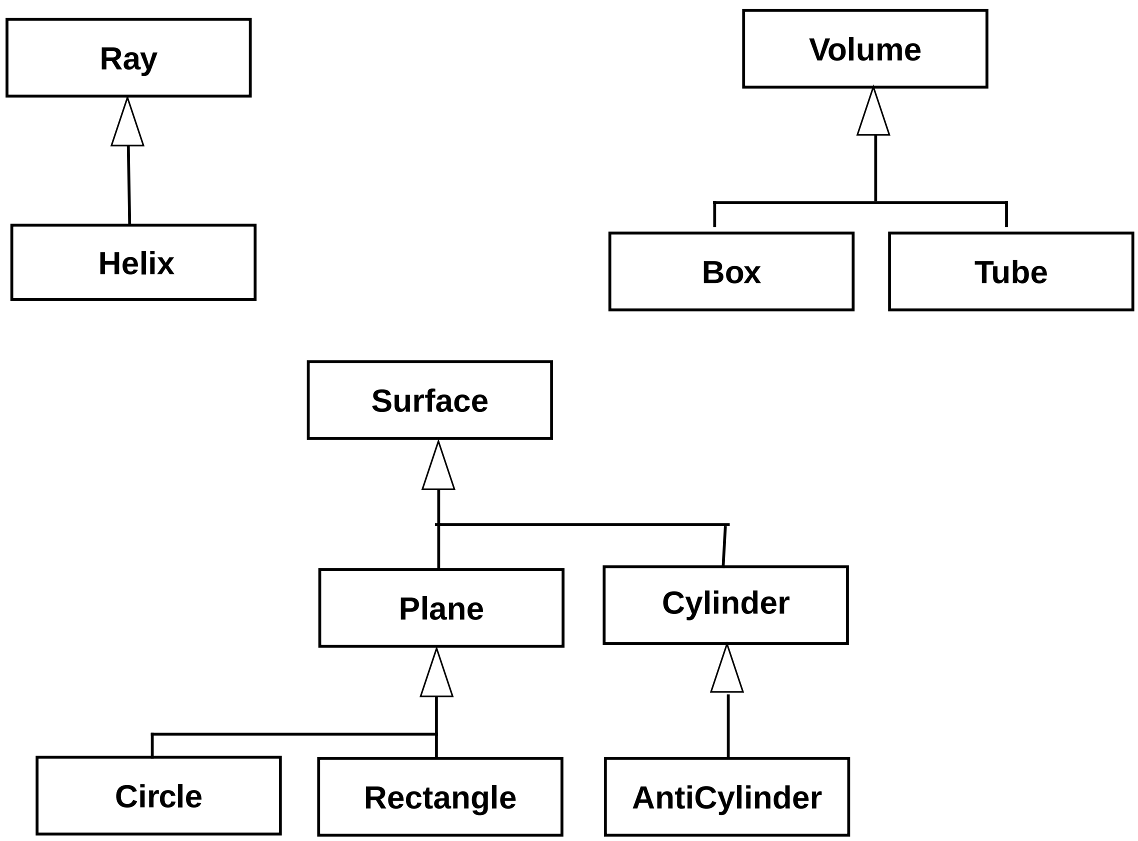
Version 1, previous release

- written by Atwood, Burnett, Alan Breakstone (Hawaii), Dave Britton (McGill) and others
- used C++ but without templates and without CLHEP
- first release was summer 1992
- `ftp://ftp.slac.stanford.edu/pubic/software/gismo-0.5.0.tar.Z`
- will show code based on this version, but updated with STL

Version 2, current version

- written by Atwood and Burnett
- C++ with templates, CLHEP and STL

Some Gismo Classes



- other Gismo classes are not shown
- we see several independent class hierarchies
- objects from these hierarchies will work together

Let’s browse some of the classes

Ray class

Part of the header

```
class Surface;
class Ray
{
public:
  Ray();
  Ray( const ThreeVec& p, const ThreeVec& d );
  virtual ~Ray() {};
  Ray( const Ray& r );
  virtual ThreeVec position( double s ) const;
  inline const ThreeVec& position() const {return pos;}
  virtual double curvature() const;
  virtual double
  distanceToLeaveSurface( const Surface* s, ThreeVec& p ) const;
// more not shown
protected:
  ThreeVec pos;
  ThreeVec dir;
  float arclength;
};
```

- you can pretty well guess the significance of the data members and many of the member functions
- a ray is clearly a straight line
- we have some virtual functions whose signifance will be explained shortly

`Helix` class

Part of the header

```
class Helix : public Ray
{
public:
  Helix();
  Helix( const ThreeVec& p, const ThreeVec& d,
         const ThreeVec& a, double r );
  virtual ~Helix() {};
  Helix( const Helix& r );
  virtual ThreeVec position( double step ) const;
  virtual double curvature() const;
  virtual double
  distanceToLeaveSurface( const Surface* s, ThreeVec& p ) const;
  // many more not shown
protected:
  ThreeVec axis;  // helix axis direction (unit vector)
  double rho;     // helix radius, sign significant
  ThreeVec perp;  // perpendicular direction
  double parallel;// component along axis
};
```

- many member functins must be re-implemented here, so probably a `Helix` is not a `Ray`
- we have some more virtual functions

`Surface` class

Part of the header

```
class Surface
{
protected:
    ThreeVec origin; // origin of Surface
public:
    Surface() : origin() {}
    Surface( const ThreeVec& o ) : origin( o ) {}
    virtual ~Surface() {}
    Surface( const Surface& s ) {
        origin = s.origin; }
    virtual double distanceAlongRay(
        int which_way, const Ray* ry, ThreeVec& p ) const = 0;
    virtual double distanceAlongHelix(
        int which_way, const Helix* hx, ThreeVec& p ) const = 0;
    virtual bool withinBoundary( const ThreeVec& x ) const = 0;
/// more not shown
};
```

- data members can be first in file, but not usual practise
- the `distanceAlong` member functions are pure virtual
- an instance of `Surface` can not be instanciated
- `Surface` exists to define an interface

`Plane` class

Part of header

```
class Plane: public Surface
{
public:
  Plane( const Point& origin, const Vector& n );
  Plane( const Point& origin, const Vector& nhat,
         double dist );
    virtual double distanceAlongRay(
        int which_way, const Ray* ry, ThreeVec& p ) const;
    virtual double distanceAlongHelix(
        int which_way, const Helix* hx, ThreeVec& p ) const;
 // more not shown
private:
   double d;
   // offset from origin to surface
};
```

- `Plane` is infinite since it has no data members to describe boundary
- distance along ray to infinite plane can be calcutated, so implementatin does exist here

`Circle` class

Part of header

```
class Circle: public Plane
{
public:
  Circle() : Plane() { radius = 1.0; }
  Circle( const ThreeVec& o,
          const ThreeVec& n, double r );
  virtual ~Circle() {}
  Circle( const Circle& c );
  virtual bool withinBoundary( const ThreeVec& x ) const;
// more not shown
protected:
  double radius;
};
```

- has data member to describe boundary
- also has member function to give the answer

`Rectangle` class

Part of the header

```
class Rectangle: public Plane
{
public:
  Rectangle();
  Rectangle( const ThreeVec& o, const ThreeVec& n,
             double l, double w, const ThreeVec& la);
  virtual ~Rectangle() {}
  Rectangle( const Rectangle& r );
  virtual bool withinBoundary( const ThreeVec& x ) const;
protected:
  double length, width;
  ThreeVec length_axis;
};
```

- data members to describe boundary
- member function to test for boundary
- data member to describe direction

Gismo Volume

Part of the header

```
class Volume
{
// a lot not shown
  virtual double distanceToLeave( const Ray& r,
          ThreeVec& p, const Surface*& s ) const;
protected:
  std::list<Surface *> surface_list;
  ThreeVec center; // center of Volume
  double roll, pitch, yaw;
};
```

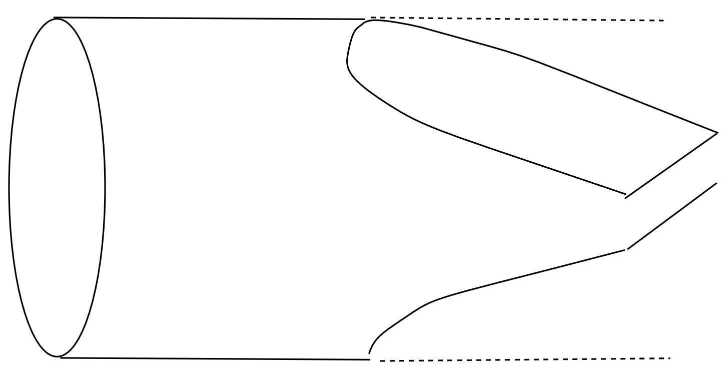
- `Volume` is a base class with common functionality of all volumes
- it contains a list of surfaces that describe the volume
- it contains a 3-vector for its center and 3 doubles for its rotation
- member functions not shown allow one to build abitrary volumes, move them, and rotate it.
- for tracking, key member function is `distanceToLeave`

Subclasses of Volume

Box

```
class Box : Volume
{
  Box( float len, float width, float height);
  Box(const Box &);
  virtual ~Box();
  // very little not shown
};
```

- constructor builds six surfaces, positions them, and adds them to surface list
- hardly any other member functions, nor any data members
- same for Cylinder and other classes
- any one could add a new volume subclass in a smiliar way, for example a light pipe



Part of implementation

The key member function

```
double Volume::distanceToLeave( const Ray& r,
             ThreeVec& p, const Surface *&sf ) const
{
  double d = 0.0, t = FLT_MAX;
  ThreeVec temp ( t, t, t );
  p = temp;
  sf = 0;
  list< Surface *>::iterator it
     = surface_list.begin();
  for( ; it != surface_list.end(); ++it ) {
    Surface * s = *it;
    d = r.distanceToLeaveSurface( s, temp );
    if ( ( t > d ) && ( d >= 0.0 ) ) {
       t = d;
       p = temp;
       sf = s;
    }
  }
  return t;
}
```

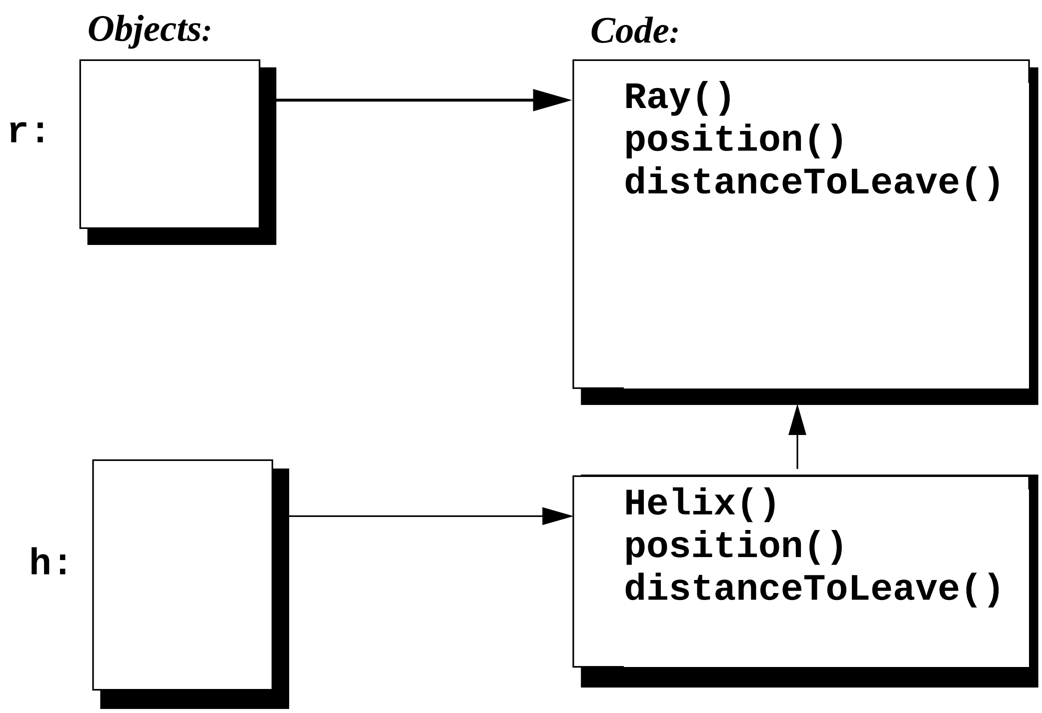
- loop over all surfaces to find the shortest distance
- the `Ray` object appears to do the work
- we don't know if the `Ray` object is-a `Ray` or the `Helix` subclass

Recall Memory model

Consider

```
Ray r;
Helix h;
```

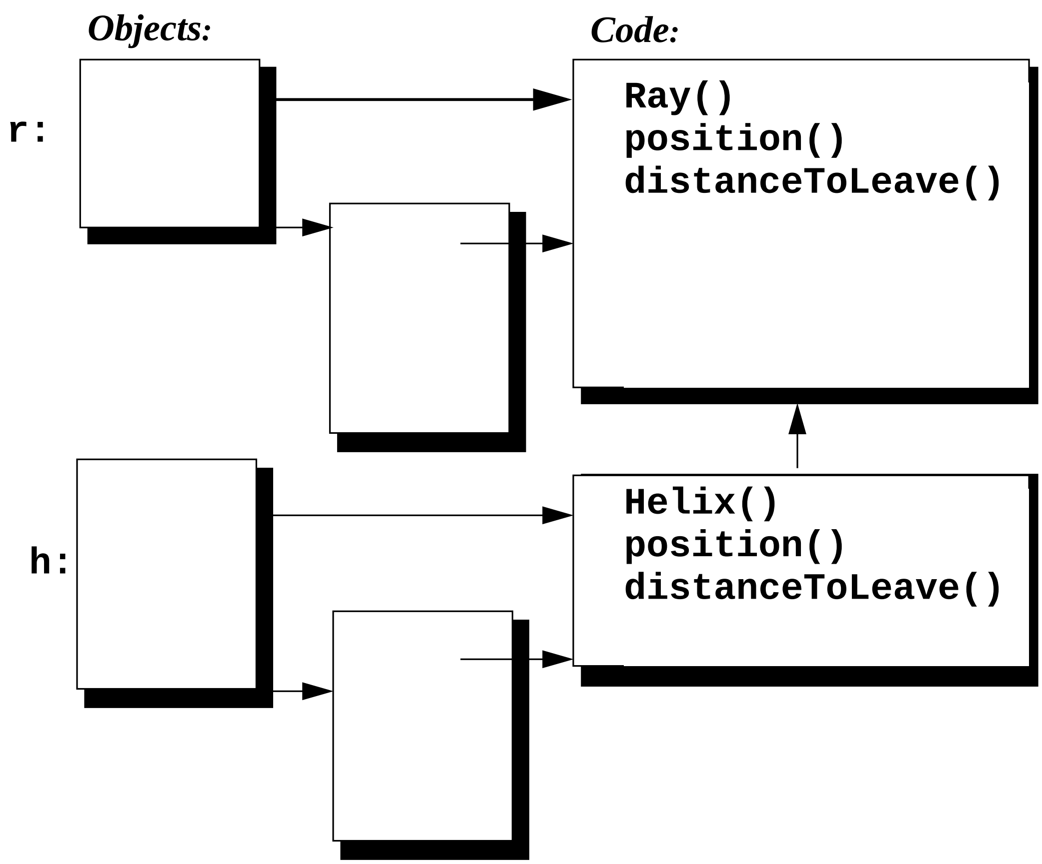
In computer's memory we have



- but now, we want `Volume` to invoke `Helix::distanceToLeaveSurface`

The virtual function table

Memory model with virtual functions



- virtual member functions are invoked indirectly via the virtual function table
- the table contains pointers to the member functions
- each class initializes the table with its functions

Back to implementation

We have

```
double Volume::distanceToLeave( const Ray& r,
             ThreeVec& p, const Surface *&sf ) const
{
  double d = 0.0, t = FLT_MAX;
  ThreeVec temp ( t, t, t );
  p = temp;
  sf = 0;
  list< Surface *>::iterator it
     = surface_list.begin();
  for( ; it != surface_list.end(); ++it ) {
    Surface * s = *it;
    d = r.distanceToLeaveSurface( s, temp );
    if ( ( t > d ) && ( d >= 0.0 ) ) {
       t = d;
       p = temp;
       sf = s;
    }
  }
  return t;
}
```

- compiler creates different machines instructions to invoke a virtual member function
- `distanceToLeaveSurface` was declared `virtual` so correct function gets called
- can even add another subclass of `Ray` without recompiling this code

Following the trail

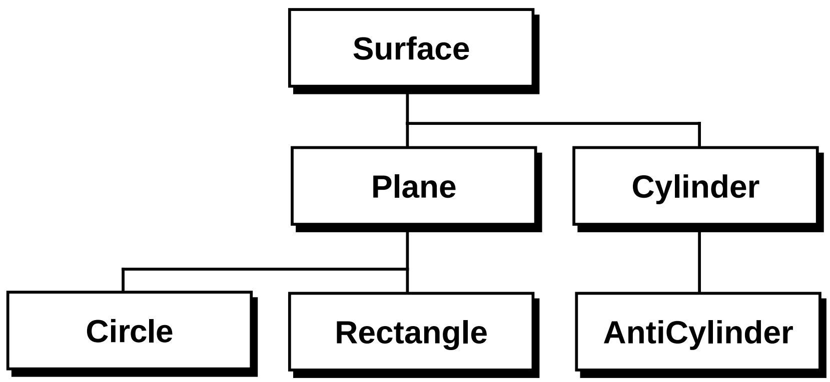
In `Ray` and `Helix` we have

```
double Ray::distanceToLeaveSurface
            ( const Surface* s, ThreeVec& p ) const
{
    return s->distanceAlongRay(  1, this, p );
}
//
double Helix::distanceToLeaveSurface
            ( const Surface* s, ThreeVec& p ) const
{
    return s->distanceAlongHelix(  1, this, p );
}
```

- so `Surface` will do the work
- this design pattern is called the Visitor pattern or the Double-Dispatch pattern
- via the `Ray` or `Helix`, we invoke the correct member function of `Surface` subclass
- recall that these functions were pure virtual in `Surface`

Where's the implementation?

Where will we find `distanceAlongRay`?



- it's not in `Surface`
- one implementation in `Plane`
- but we really instansiate objects of type `Circle` or `Rectangle`
- another in `Cylinder`

Implementation

In `Plane`, we have

```
double Plane::distanceAlongRay( int which_way,
       const Ray* ry, ThreeVec& p ) const
{
  double dist = FLT_MAX;
  ThreeVec lv ( FLT_MAX, FLT_MAX, FLT_MAX );
  p = lv;
//  Origin and direction unit vector of Ray.
  ThreeVec x = ry->position();
  ThreeVec dhat = ry->direction( 0.0 );
  ThreeVec nhat = normal(); // Normal to plane
  double denom = nhat * dhat;
  if ( ( denom * which_way ) <= 0.0 )
    return dist;  // return large distance
  double d = ( ( ( getOrigin() - x ) * nhat ) / denom );
  if ( ( d >= 0.0 ) && ( d < FLT_MAX ) ) {
    dist = d;
    p = ry->position( d );
    if ( ! withinBoundary ( p ) ) {
      dist = FLT_MAX;
      p = ThreeVec( FLT_MAX, FLT_MAX, FLT_MAX );
    }
  }
  return dist;
}
```

- `withinBoundary()` member function must be in `Circle` or `Rectangle`
- example of template pattern

As expected

In `Circle` we have

```
bool Circle::withinBoundary( const ThreeVec& x ) const
{
  ThreeVec p = x - origin;
  if ( p.magnitude() <= radius )
    return true;
  else
    return false;
}
```

In `Rectangle` we have

```
bool Rectangle::withinBoundary( const ThreeVec& x ) const
{
  ThreeVec p = x - origin;
  ThreeVec width_axis = norm.cross( length_axis );
  if ( ( fabs( p * length_axis ) <= ( 0.5 * length ) )  &&
       ( fabs( p * width_axis  ) <= ( 0.5 * width  ) ) )
    return true;
  else
    return false;
}
```

Virtual destructor

In Volume, we may have

```
Volume::~Volume()
{
  list< Surface * >::iterator it
    = surface_list.begin();
  while ( it != surface_list.end() ) {
    delete *it++;
  }
}
```

- we need to call the destructor for `Circle`, `Plane`, *etc*
- thus we make the destructor virtual for this heirarchy
- gcc will warn you if you don't

Summary

Inheritance used for

- used to expressed common implementation
- used to expressed common behavior
- used to expressed common structure

Virtual inheritance allows objects to use abstract base functions with concrete classes

We’re Done!

But…

- its like you’ve heard lectures on how to swim, but now you face the deep end of the pool
- its like you know the rules of the game of chess, but have not yet studied stratgies

Further reading:

- Designing object-oriented C++ applications using the Booch method, Robert C. Martin, ISBN 0-13-203837-4, Prentice Hall
- Design Patterns, Gamma, Helm, Johnson, and Vlissides, ISBN 0-201-63361-2, Addison-Wesley