Plan of the day

Inheritance is last major feature of the language that we need to learn

- used to express common implementation
- used to express common behavior
- used to express common structure

Will divert from the textbook in order to introduce HEP specific classes

- Examples from CLHEP
- Examples from Gismo (next session)
Recall ThreeVector

CLHEP’s ThreeVector class (simplified)

```cpp
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();
    // much more not shown
private:
    double dx, dy, dz;
};
```

and some of the implementation

```cpp
inline double Hep3Vector::x() {
    return dx;
}
inline double Hep3Vector::mag() {
    return sqrt(dx*dx + dy*dy + dz*dz);
}
```
Recall our test program

The object does the work

```cpp
#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;
}
```

including algebraic operators

```cpp
Hep3Vector p, q, r;
double z;
// ...
z = p*q;
r = p + q;
```
Possible 4-Vector Class

Might look like...

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

Compare with 3-Vector class

• some member functions must be exactly the same
• some member functions are added
• some member functions must be re-implemented
• some data is the same
• one new data item
Another Possible 4-Vector Class

** Might look like...**

```cpp
class HepLorentzVector {
public:
    HepLorentzVector();
    HepLorentzVector(double x, double y, double z, double t);
    HepLorentzVector(const HepLorentzVector &v);
    inline double x();
    inline double y();
    inline double z();
    inline double t();
    inline double mag();
    // much more not shown
private:
    Hep3Vector vec3;
    double dt;
};
```

- **HepLorentzVector has-a** Hep3Vector
- **could also say** HepLorentzVector is built by aggregation
- **or with containment**
Possible implementation

Constructors

HepLorentzVector::HepLorentzVector() :
  vec3(), dt(0.0) {}

HepLorentzVector::
HepLorentzVector(double x, double y, double z, double t) :
  vec3(x, y, z), dt(t) {}

HepLorentzVector::
HepLorentzVector(const HepLorentzVector &v) :
  vec3(v.vec3), dt(v.dt) {}

- note use of initializers
- must construct data members when constructing class object

Let 3-vector component do part of the work

double HepLorentzVector::mag() {
  return sqrt(dt*dt - vec3.mag2());
}

must still implement functions like

double HepLorentzVector::x() {
  return vec3.x();
}
Constructors

class HepLorentzVector {
public:
    HepLorentzVector();
    HepLorentzVector(double x, double y, double z, double t);
    HepLorentzVector(const HepLorentzVector &v);
    inline double x();
    inline double y();
    inline double z();
    inline double t();
    inline double mag();
    // much more not shown
private:
    Hep3Vector *vec3;
    double dt;
};

• still have containment, but use a pointer
• makes sense in some situations (probably not here)
YAPI implementation

Constructors might be

```cpp
HepLorentzVector::HepLorentzVector() : dt(0.0) {
    vec3 = new Hep3Vector(0, 0, 0);
}

HepLorentzVector::HepLorentzVector(double x, double y, double z, double t) :
    dt(t) {
    vec3 = new Hep3Vector(x, y, z);
}

HepLorentzVector(const HepLorentzVector &v) : dt(v.dt) {
    vec3 = new Hep3Vector( *v.vec3); // copy constructor
}
```

- using `new` operator to create one object
- will need to implement destructor!


Inheritance

Part of the header file

```cpp
class HepLorentzVector : public Hep3Vector {
public:
    HepLorentzVector();
    HepLorentzVector(double x = 0., double y = 0.,
                      double z = 0., double t = 0.);
    HepLorentzVector(const HepLorentzVector &v);
    HepLorentzVector(const Hep3Vector &p, double t);
    double t();
    double mag();
    // much more not shown
private:
    double dt;
};
```

- HepLorentzVector is-a Hep3Vector
- All public members of Hep3Vector are also public members of HepLorentzVector by use of keyword public in class declaration.
- member function \( t() \) is added
- member function \( \text{mag()} \) overrides function of same name in Hep3Vector
- constructors take different arguments
- one new data member: \( dt \)
Consider

```cpp
int main() {
    double x, y, z, t;
    while ( cin >> x >> y >> z >> t ) {
        Hep3Vector a3Vec(x, y, z);
        HepLorentzVector a4Vec(x, y, z, t);

        cout << "3-vector x and mag: "
             << a3Vec.x() << " " << a3Vec.mag() << endl;
        cout << "4-vector x and mag: "
             << a4Vec.x() << " " << a4Vec.mag() << endl;
    }
    return 0;
}
```

- HepLorentzVector behaves like any other class
- how does a4Vec.x() work since no member function has been defined?... by inheritance
- a4Vec.mag(), however, is completely different from a3Vec.mag()

- output of program

```
hpkaon> a.out
1 1 1 2
3-vector x and mag: 1 1.73205
4-vector x and mag: 1 1
```
Consider

\begin{align*}
\text{Hep3Vector} & \ x(1.0, 0.0, 0.0); \\
\text{HepLorentzVector} & \ y(1.0, 0.0, 0.0, 5.0);
\end{align*}

In computer’s memory we have

- inheritance of data members
- inheritance of member functions
Constructor Implementations

Constructors

HepLorentzVector::
HepLorentzVector(double x, double y, double z, double t) :
Hep3Vector(x, y, z), dt(t) {}  

HepLorentzVector::
HepLorentzVector(const Hep3Vector &v, double t) :
Hep3Vector(v), dt(t) {}  

HepLorentzVector::
HepLorentzVector(const HepLorentzVector &v) :
Hep3Vector(v), dt(v.dt) {}  

• super class will be constructed before subclass
• use initializers to direct how to construct superclass
More of Implementation

As you might expect

```cpp
inline double HepLorentzVector::t() const {
    return dt;
}
```

- the \texttt{t()} member function is like we’ve seen before

This doesn’t work

```cpp
inline double HepLorentzVector::mag2() const {
    return dt*dt - (dx*dx + dy*dy + dz*dz);
}
```

- \texttt{dx}, \texttt{dy}, and \texttt{dz} were declared \texttt{private}

- \texttt{private} means access to objects of the same class
  and \texttt{HepLorentzVector} is a different class

- could modify \texttt{Hep3Vector} to

```cpp
class Hep3Vector {
public:
    // same as before
protected:
    double dx, dy, dz;
}
```

- \texttt{protected}: means access to members of the same class and all subclasses
More on Implementation

Keep the base class data members private

```cpp
inline double HepLorentzVector::mag2() const {
    return dt*dt - Hep3Vector::mag2();
}
```

- use scope operator `::` to access function of same name in super class
- now we can re-write Hep3Vector to use \( r \), \( \cos\theta \) and \( \phi \) without needing to re-write HepLorentzVector
- less dependencies between classes is good

Finally, we have

```cpp
inline double HepLorentzVector::mag() const {
    double pp = mag2();
    return pp >= 0.0 ? sqrt(pp) : -sqrt(-pp);
}
```

- did you remember that 4-vector can have negative magnitude?
Even more of Implementation

The dot product

```c++
inline double HepLorentzVector::dot(const HepLorentzVector & p) const {
    return dt*p.t() - z()*p.z() - y()*p.y() - x()*p.x();
}
```

- use of accessor functions \( x() \), \( y() \), and \( z() \) because data members are private in the super class
- scope operator \( :: \) not needed because these functions are unique to the base class

The \( += \) operator

```c++
inline HepLorentzVector & HepLorentzVector::operator += (const HepLorentzVector& p) {
    Hep3Vector::operator += (p);
    dt += p.t();
    return *this;
}
```

- example of directly calling operator function

Many other functions will not be shown

They implement the vector algebra for Lorentz vectors
What’s new?

A Lorentz boost function

```cpp
void HepLorentzVector::boost(double bx, double by, double bz){
    double b2 = bx*bx + by*by + bz*bz;
    register double gamma = 1.0 / sqrt(1.0 - b2);
    register double bp = bx*x() + by*y() + bz*z();
    register double gamma2 = b2 > 0 ? (gamma - 1.0)/b2 : 0.0;

    setX(x() + gamma2*bp*bx + gamma*bx*dt);
    setY(y() + gamma2*bp*by + gamma*by*dt);
    setZ(z() + gamma2*bp*bz + gamma*bz*dt);
    dt = gamma*(dt + bp);
}
```

- **register** keyword advises compiler that variable should be optimized in machine registers

Also have

```cpp
inline Hep3Vector HepLorentzVector::boostVector() const {
    Hep3Vector p(x()/dt, y()/dt, z()/dt);
    return p;
}
inline void HepLorentzVector::boost(const Hep3Vector & p){
    boost(p.x(), p.y(), p.z());
}
```
Diagrams

The old ones

• Booch’s “clouds”, supported by Rational/Rose
• Rumburgh’s OMT

The new one

• UML: Unified Modeling Language
• Booch and Rumburgh working together
• later joined by Jacobsen
• the “three amigos”
• submitted for standardization
Aggregation

If we have a *has-a* relationship we draw it thus

![Diagram showing the relationship between LorentzVec and ThreeVec]

- corresponding code…

```cpp
class LorentzVec {
    // much more not shown
    private:
        ThreeVec vec3;
        double dt;
};
```

- **LorentzVec** contains **ThreeVec**

- contained object will be destroyed with the containing object is destroyed
Association

If we have a association relationship we draw it thus

![Diagram of association between Car and Motor]

• corresponding code…

```cpp
class Car {
    // much more not shown
private:
    Motor *m;
};
```

• not 100% sure just because we have pointer
• only association if motor is replaceable
• depends on what kind of application this Car class is being used for.
Inheritance

If we have *is-a* relationship we draw it thus

```
class LorentzVec : public ThreeVec {
    // much more not shown
    private:
        double dt;
};
```

• corresponding code

• this is class relationship, not object relationship

• don’t be confused with our memory model diagrams

• we say ThreeVec is base class and LorentzVec is derived class
Bad inheritance

When a square is a rectangle and when it isn’t

```
class Rectangle {
    // much more not shown
    void setLength(float);
    void setHeight(float);
    //...
    float length, height;
};
```

• corresponding code

• now what’s the Square going to do about these member functions?

• in math, a square is a subset of all rectangles, but in C++ a Square is not a subclass of Rectangle
A Possible Particle class

Take Lorentz vector and add to it

```cpp
class Particle : public HepLorentzVector
{
public:
    Particle();
    Particle(HepLorentzVector &, PDTEntry *);
    Particle(const Particle &);  
    ~Particle() {}  
    float charge() const;  
    float mass() const;  
    // more methods not shown
protected:
    float m_charge;  // units of e
    PDTEntry * m_pdtEntry;  
    std::list<Particle *> m_children;  
    Particle * m_parent;
};
```

• note one can inherit from a class which is derived class

• added features are charge, pointer to entry in particle data table, list of children, and pointer to parent

• owns list of children

• `m_pdtEntry` and `m_parent` are pointers because of shared objects

• not very useful class
Data Model

In computer’s memory we have

**Objects:**
- \(dx\)
- \(dy\)
- \(dz\)

**Code:**
- \(\text{Hep3Vector}()\)
  - \(x()\)
  - \(y()\)
  - \(z()\)
  - \(\text{mag}()\)
  - \(\text{phi}()\)
  - \(\text{cosTheta}()\)

- \(\text{HepLorentzVector}()\)
  - \(t()\)
  - \(\text{mag}()\)

- \(\text{Particle}()\)
  - \(\text{charge}()\)
  - \(\text{mass}()\)

**Code:**
- \(dx\)
- \(dy\)
- \(dz\)
- \(dt\)
- \(\text{m\_charge}\)
- \(\text{m\_pdtEntry}\)
- \(\text{m\_children}\)
- \(\text{m\_parent}\)
Class Diagram

Inheritance and relationships

- Particle has 0 to n children and 0 or 1 parents
- Particle has association with PdtEntry
- We leave the list<> out of the picture
Object Hierarchy

In computer memory we have

- the class and object hierarchies are different in dimensions
The 3 hierarchies of OOP

It’s a three dimensional space

- Class hierarchy describes behavior
- Object hierarchy describes data structure
- hierarchy of levels of abstraction, *e.g.* float, vector, lists, arrays, particle, *etc.*
Multiple Inheritance

One can inherit from more than one class

class AsTrack : public HepLockable, public Particle
{
public:
    AsTrack();
    AsTrack(AsEvent *e, int type, int index);
    AsTrack(const AsTrack &);
    virtual ~AsTrack();
    // more member functions not shown
}

• AsTrack inherits from both Particle and HepLockable

• both data members and member functions are inherited from both classes
Class hierarchy

For both data members and functions we have

- **AsTrack** has the functions defined in itself and all of its super classes
- **AsTrack** has data members defined in itself and all of its super classes
**AsTrack**’s constructor

### Beginning of constructor

```cpp
AsTrack::AsTrack(AsEvent *e, int type, int index) : Lockable(), Particle()
{
    _type = type;
    _index = index;
    int ftype = type + 1;
    int find = index + 1;
    float p[20];
    trkallc(&ftype, &find, p);

    setX(p[0]);
    setY(p[1]);
    setZ(p[2]);
    setT(p[3]);
    _charge = p[10];
    // more not shown
}
```

- note calling the constructors of the super classes
- careful: the super class constructors are called in order of the class definition, not necessarily in the order listed in the constructor.
- `trkallc` is a Fortran subroutine that fetches data out of ASLUND’s COMMON blocks
Summary

We now know enough C++ to do a physics analysis

Next session we’ll look at polymorphic uses of inheritance with examples from Gismo

Then, we’ll be pretty much done with learning the language

It’s soon time to start some mini-projects using C++