SUPA Graduate C++ Course

Lecture 2

Tom Doherty

University of Glasgow

Lecture 1 Recap

- Syntax
- Types
- Functions
- Pointers
- Arrays
- Scope
- Header Files
- Compilation and Makefiles

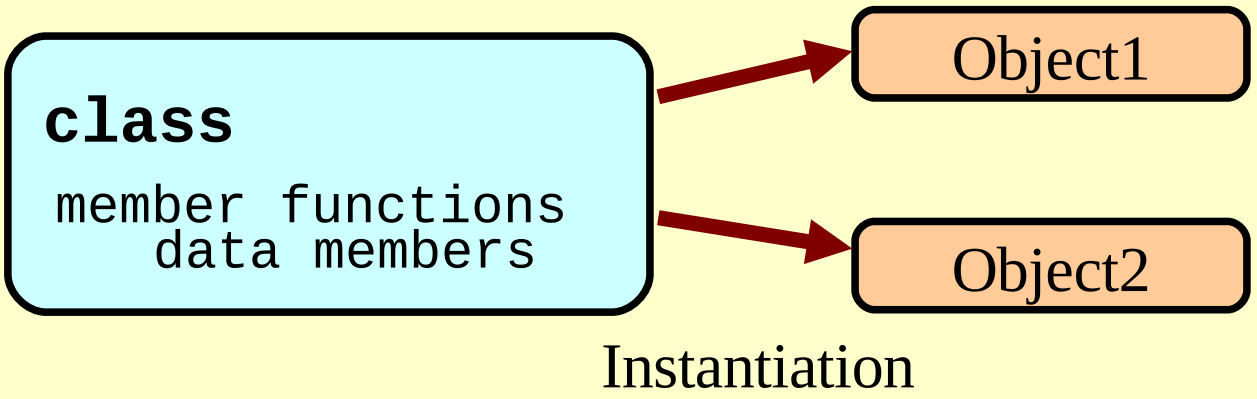
Recap lecture 1 and tutorial

- Passing an array name as an argument to a function passes a pointer to the first element in the array.
- e.g. void myFunction(int array[3][3])
- The function can read and change the value of any of the elements of the array.

Lecture 2 Overview

- Introducing Objects
  - Concept Introduction
  - Implementing Objects
  - Constructors, destructors, `new` and `delete`.
- Object-Object Communication
- Operator Overloading
- Inheritance

Objects - Introduction



- A class is the building block of Object Oriented programming.
  - A class defines a new data ‘type’, and what can be done with that ‘type’
  - An object is an instance of a class .

Particle Physics Example

Particle:

- Has momentum (px, py, pz), energy (E).
- We work in terms of 4-vectors: (px, py, pz, E)
- From these, we can calculate:
  - mass = $(E^2 - \underline{p}^2)$
  - transverse momentum $p_T = \sqrt{(p_x^2 + p_y^2)}$

Electron:

- Has all of the above + charge and an identification code.

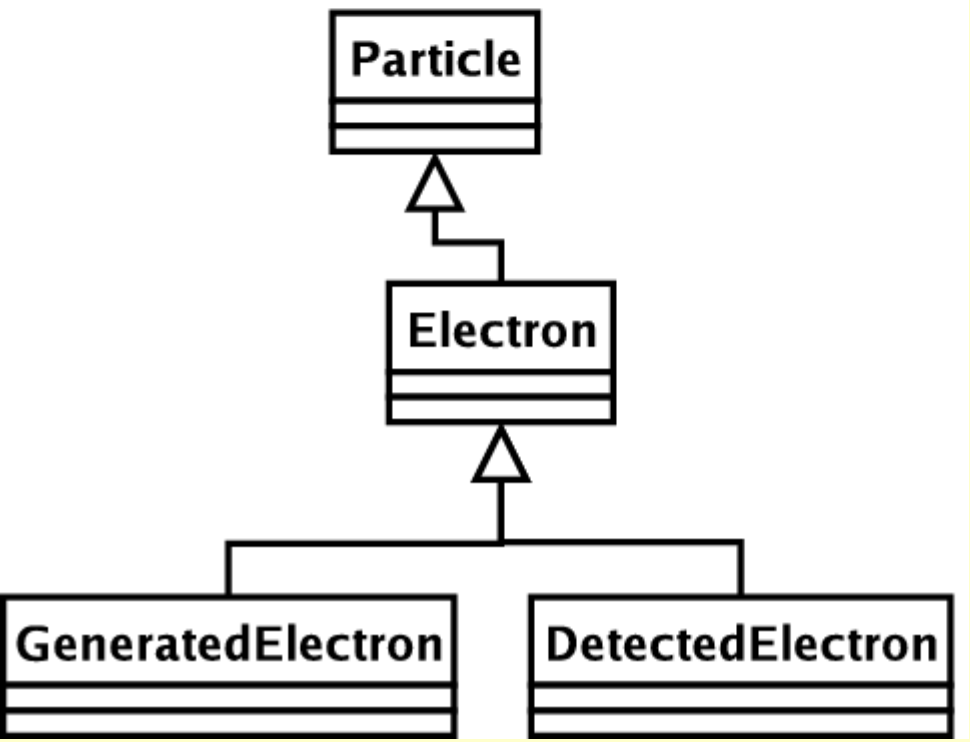
DetectedElectron:

- same as electron + information about tracks in the detector.

GeneratedElectron:

- same as electron + information about the decay it originates from.

Designing a Program with OO



A simple class inheritance structure in UML

- Build up complexity using class building blocks.
  - Create more general base classes
  - Use inheritance to build on existing functionality.
  - Could use objects from any part of the inheritance tree within a program.

A Class Declaration

```
class BasicParticle {
public:
  BasicParticle(void);
  BasicParticle(double *fourvector);
  void assignFourVector(double *);
  double getPt();
  double getMass();

private:
  void calculatePt();
  void calculateMass();

  double m_fourvector[4];
  double m_pt;
  double m_mass;
};
```

Member functions

Member functions

Data Members

Extract from ex1/BasicParticle.hh

Protection labels

- `public` methods/member functions :
  - Generally accessible – can be accessed through any object of the class.
  - Constructors and accessors (to get or set values).
- `private` methods and data members / attributes / variables:
  - Accessible only from within the class.
  - useful for containing the code, because they hide the implementation from the user.

Naming Conventions

- Class Names
  - Start with a capital letter
- Member Functions
  - Start with a lower case letter. (Use camel text e.g. assignFourVector)
- Private Data Members:
  - A common convention is to prefix each private data member with `m_`

Constructors

- special member function that builds objects belonging to a class and initializes the data members.

```
BasicParticle(void);

BasicParticle(double *fourvector);
```

Extract from ex1/BasicParticle.hh

Can have many constructors, differing in type or number of arguments.

When an object myparticle is instantiated by:

```
BasicParticle myparticle;
```

or

```
BasicParticle myparticle(myfourvector);
```

the appropriate constructor is called automatically.

Implementation of Example Class

```
#include "BasicParticle.hh"
#include <cmath>
#include <iostream>

/** Constructors ********************/
BasicParticle::BasicParticle()
{
}

BasicParticle::BasicParticle(double *fourvector)
{
  assignFourVector(fourvector);
}
...
```

Extract from ex1/BasicParticle.cc

- When the constructor is invoked,
  - Memory is allocated for the object
  - The members are initialized
  - The body of the constructor is executed.

Implementation of Example Class

```
void BasicParticle::assignFourVector(double *fourvector) {
  cout << "Assigning fourvector to particle:" << endl;
  for(int i=0;i<4;i++) {
    m_fourvector[i] = fourvector[i];
    cout << "fourvector[" << i << "]="
      << fourvector[i] << endl;
  }
  cout << endl;

  calculatePt();
  calculateMass();
}

double BasicParticle::getPt() {
  return m_pt;
}
```

Extract from ex1/BasicParticle.cc

- Private variables are 'globals' within the class

Using the Example Class

```
#include <iostream>
#include "BasicParticle.hh"

using namespace std;

int main() {
  double fourvector1[4] = {3.0, 4.0, 5.0, 7.35};
  double fourvector2[4] = {2.0, 2.0, 1.0, 3.0};

  BasicParticle particle1(fourvector1);
  BasicParticle particle2(fourvector2);

  cout << "Mass of particle 1=" << particle1.getMass() << endl;
  cout << "pt of particle 1=" << particle1.getPt() << endl << endl;
  cout << "Mass of particle 2=" << particle2.getMass() << endl;
  cout << "pt of particle 2=" << particle2.getPt() << endl;

  return 0;
}
```

Extract from ex1/main.cc

Constructing Objects

Can instantiate objects in two ways:

```
BasicParticle particle1(fourvector1);
```

- Objects created this way harmlessly go out of scope.
- When the block ends, object goes out of scope and memory is automatically deallocated.
- Any pointers to the object are then invalid

Constructing Objects

```
BasicParticle *particle1 = new BasicParticle(fourvector1);
```

- `new` allocates memory dynamically, and returns a pointer to the object.
- The object stays around until the program ends or until it is deleted.
- Objects created with `new` must be deleted to prevent memory leaks.

```
delete particle1;
```

- returns the memory to the heap.

Calling Member Functions

- From outside the class

```
BasicParticle particle1(fourvector1);
particle1.getMass();
```

```
Parent *parent = new Parent(id, mass);
parent->run();
```

```
Parent *parent = new Parent(id, mass);
(*parent).run();
```

- From inside the class

```
BasicParticle::BasicParticle(double *fourvector)
{
  assignFourVector(fourvector);
}
```

Destructors

- A member function to perform any clean up when the object goes out of scope.
  - Delete any memory associated with the class
- Called automatically when an object goes out of scope, or when an object created with `new` is explicitly deleted.

```
Parent::~Parent()
{
  delete m_child;
}
```

Extract from ex2/Parent.cc

```
class Parent {
public:
  Parent(int, double);
  ~Parent(void);
};
```

Extract from ex2/Parent.hh

Object Communication

- What happens when we write `particle1.getMass()`:

```
double BasicParticle::getMass(){
 return m_mass;
}
```

is actually:

```
double BasicParticle::getMass(){
 return this->m_mass;
}
```

In any class member function there is a hidden argument – a pointer to the object that called the member function.

The pointer `this` contains the address of `particle1.`

Sometimes we need to use `this` explicitly…

Object Communication

Two situations:

- An object creates another object and then needs to access data within the created object.
- An object is created by another object and then needs to access data within the object that created it. Use `this`.

Example 2: Two classes, Parent and Child.

Within a member function of Parent, we create an object of the Child class. Child is instantiated with a pointer to an object of the Parent Class.

Within a member function of Child we call a Parent member function.

Object Communication

```
Parent *parent = new Parent(id, mass);
parent->run();
```

Extract from ex2/main.cc

```
void Parent::run()
{
  // Only create a child if there isn't one already
  if(!m_child) {
    m_child = new Child(this);
    m_child->run();
  }
}
```

Extract from ex2/Parent.cc

1. Create an object
2. Call one of its member functions

Object Communication

- In this case, “Child” constructor is defined with a parameter which is a pointer to an object of the “Parent” class.
- Child’s run() method calls member functions of Parent.

```
Child::Child(Parent *parent)
{
  m_parent = parent;
}

void Child::run() {
  cout << "parent mass = " << m_parent->getMass() << endl;
  cout << "parent id = " << m_parent->getId() << endl;
}
```

Extract from ex2/Child.cc

Operator Overloading

```
float x=0,y=5,z=3;
x = ++y * z;
x = x/2.0;
```

- Simple arithmetic and other functionality can be implemented in a class
- Implementation of `operator` member functions is called Operator Overloading.

```
BasicParticle *particle1 = new BasicParticle(fourvector1);
BasicParticle *particle2 = new BasicParticle(fourvector2);
BasicParticle particle3 = *particle1 + (*particle2);
```

Extract from ex3/main.cc

Operator Overloading

```
class BasicParticle {
public:
  BasicParticle operator+(BasicParticle);

private:
  double m_fourvector[4];
};
```

Extract from ex3/BasicParticle.hh

```
BasicParticle BasicParticle::operator+(BasicParticle particle) {
  double resultant[4];

  for (int i=0;i<4;i++) resultant[i] = m_fourvector[i] +
particle.m_fourvector[i];
  return BasicParticle(resultant);
}
```

Extract from ex3/BasicParticle.cc

Operator Overloading

```
BasicParticle *particle1 = new BasicParticle(fourvector1);
BasicParticle *particle2 = new BasicParticle(fourvector2);
   BasicParticle particle3 = *particle1 + (*particle2);
```

Extract from ex3/main.cc

Inheritance

- Example 4: 3 Classes
- **Bag:** has volume information only
- **ColouredBag:** inherits from Bag, and also has colour.
- **BeanBag:** inherits from ColouredBag, and also has beans.

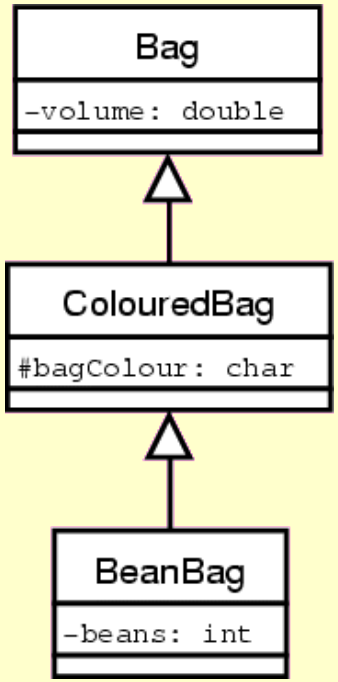
Inheritance

```
class Bag {
public:
  Bag(double volume);
  double getVolume(void);
  void setVolume(double volume);

private:
  double m_volume;
};
```

Extract from ex4/Bag.hh

```
class ColouredBag: public Bag {
public:
  void setColour(char);
  char getColour(void);

protected:
   char m_bagColour;
};
```

Extract from ex4/ColouredBag.hh




```
class BeanBag: public ColouredBag {
public:
  BeanBag(char colour);
  int fillWith(int );
  int removeBeans(int );
  int getNumBeans(void);

private:
  int m_beans;
};
```

Extract from ex4/BeanBag.hh

Inheritance

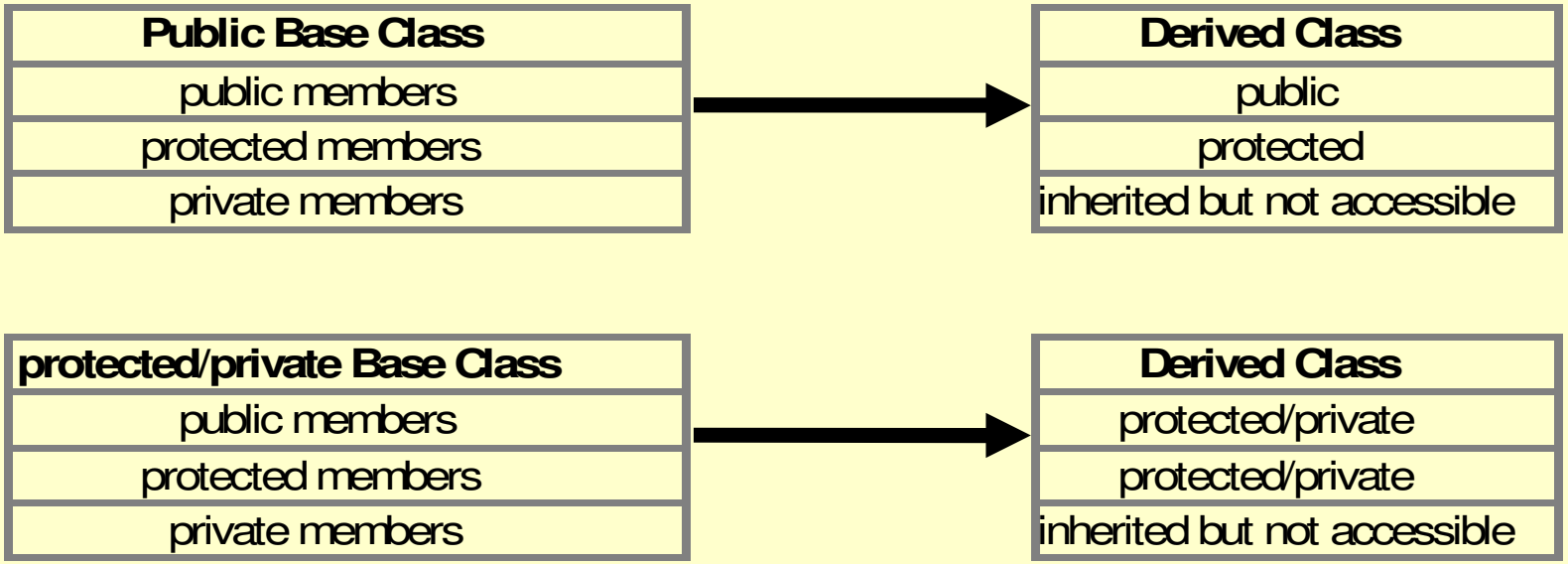
```
Bag bag(30.0);

ColouredBag colouredBag;
colouredBag.setVolume(40.0);
colouredBag.setColour('r');

BeanBag beanBag('b');
beanBag.setVolume(50.0);
beanBag.fillWith(100);

cout << "Volume of bag = " << bag.getVolume() << endl << endl;
cout << "Volume of colouredBag = " << colouredBag.getVolume()
     << endl;
cout << "Colour of colouredBag = " << colouredBag.getColour()
     << endl << endl;
cout << "Volume of BeanBag = " << beanBag.getVolume() << endl;
cout << "Colour of BeanBag = " << beanBag.getColour() << endl;
cout << "Beans in BeanBag = " << beanBag.getNumBeans() << endl;
```

Extract from ex4/main.cc

Inheritance

```
BeanBag::BeanBag(char colour) {
  m_bagColour = colour;
}
```

Extract from ex4/BeanBag.cc



Exercises

- Session 2:
  - Download examples from My.SUPA
  - Build and test examples
  - Section 2 problems will be set this week. Attempt them before coming to the tutorial.
- Tutorial for session 2:

Monday 8th November 10am

Room 320 Kelvin Building

- Deadline for Section 1 problems: 15th November.
- Deadline for Section 2 problems: 22nd November