SUPA Graduate C++ Course

Lecture 2

S. Allwood-Spiers

University of Glasgow

Lecture 1 Recap

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

In lecture 1 examples. Will cover today.

Lecture 2 Overview

- Pointers and Functions
- References
- Header Files and Makefiles (carried over from section 1)
- Introducing Objects
  - Concept Introduction
  - Implementing Objects
  - Constructors, destructors, `new` and `delete`.
- Operator Overloading
- Streams (carried over from section 1)

Pointers and Functions

```
void fun(int, int *);

int main() {
  int np = 1, p = 1;

  cout << "Before fun(): np=" << np << " p=" << p << endl;
  fun(np, &p);
  cout << "After fun(): np=" << np << " p=" << p << endl;
  ...
}

void fun(int np, int *p) {
  np = 2;
  *p = 2;
}
```

Extract from ex6/Pointers.cc

Pointers and Functions

- Passing an array name to a function passes a pointer to the first element
- e.g. void myFunction(int array[3][3])
- The function can read and change the value of any of the elements of the array.
- Any other objects passed into functions behave in a similar way to simple variables in the given example
  - If changes made within a function are needed after the function has executed Pointers or References should be used.

References

- References: Similar to pointers in many ways, but different syntax and less flexible. Declare with <type> &<name>: e.g.

```
int myVar=1;
int &refToVar = myVar; //refToVar is a reference to myVar.
```

- Must be initialised at creation, and cannot be changed to refer to another object.
- Use a reference as if it was a value: Value accessed by `refToVar`, address accessed by `&refToVar`.
- When used as an argument to functions, the caller does not need to explicitly say they are using a reference.

```
void fun(int nr, int &r);  //function fun expects an int and
                          // a reference to an int as arguments.
int main() {
  int nr=1, r=1; //nr and r are both ints.

  fun(nr, r); // nr will be passed by value,
              // r will be passed by reference.
}
```

Extract from section1: ex8/references.cc

Header Files

- Can contain:
  - Pre-definition of functions
  - Class declarations
  - Variable declaration
- Processed during pre-compilation.
  - Pre-compiler has its own syntax

Header Files

```
#ifndef STDIO_TESTS_HH
#define STDIO_TESTS_HH

void numFingers(int);
void pickColour(void);
bool quitTime(void);

#endif
```

Extract from Section 1 ex7/StdioTests.hh

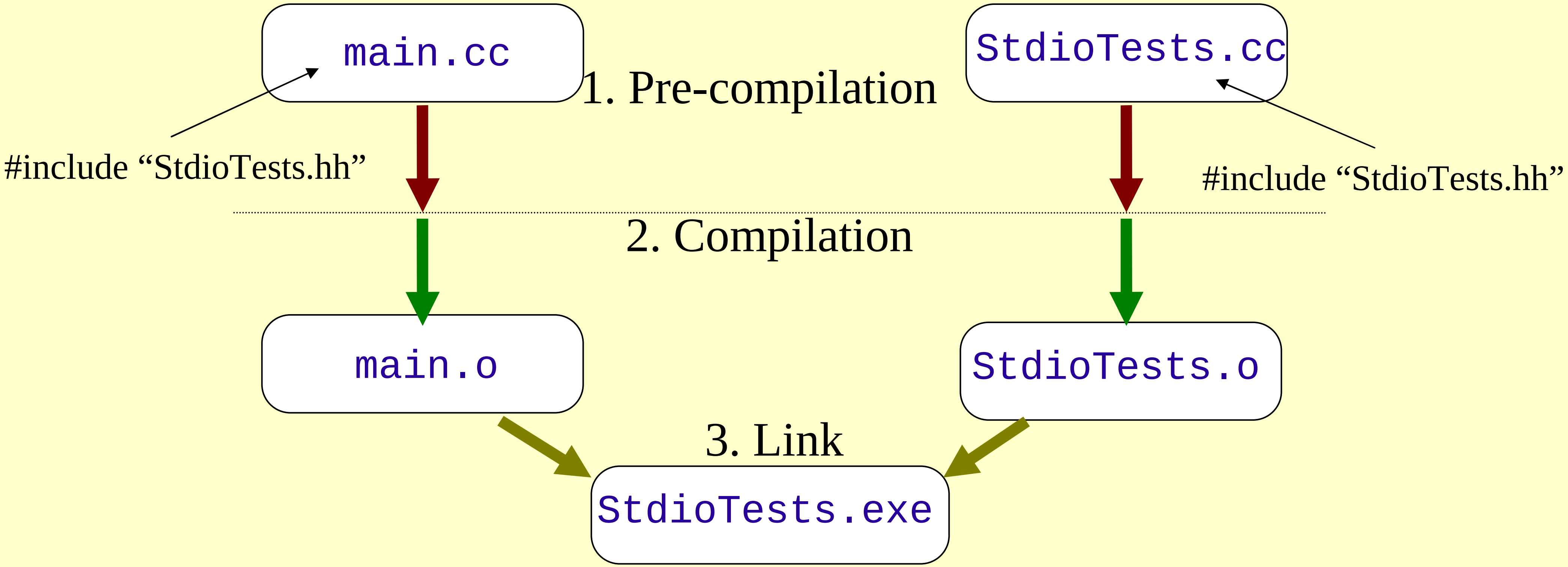
Prevent multiple declarations

```
...
#include “StdioTests.hh"
...

int main() {
  ...
    pickColour();
  ...
}
```

Extract from Section 1 ex7/main.cc

Must be in the include path

Building an Executable



```
g++ -c main.cc
g++ -c StdioTests.cc
g++ main.o StdioTests.o -o StdioTests.exe
```

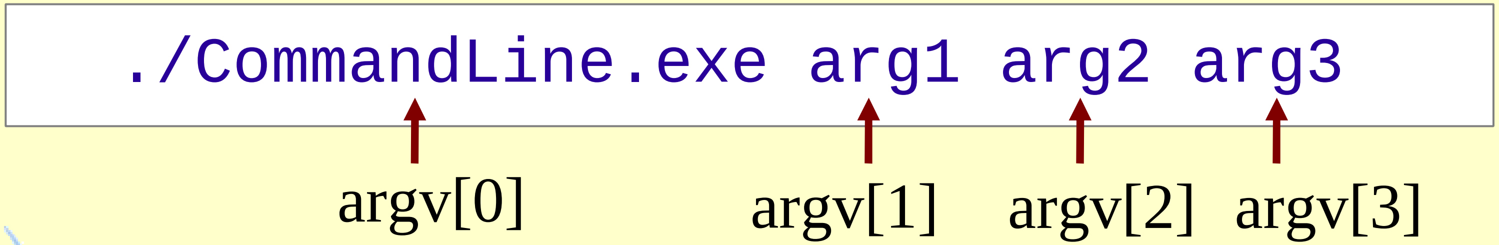
- When linking with `g++`, `ld` is used
- The `ld` command line depends on which gnu compiler is used

Command Line Arguments

Number of arguments given to the command line

```
int main(int argc, char *argv[]) {
  cout << "argc=" << argc
       << " (argc => size of argv array)" << endl;
  for(int i=0;i<argc;i++) {
    cout << "argv[" << i << "]=" << argv[i] << endl;
  }
  return 0;
}
```

Extract from CommandLine.cc



```
./CommandLine.exe arg1 arg2 arg3
```

Make

- A useful tool for building executables and libraries
- Documentation:
  - Man pages   `man make`
  - Info pages   `info make`
  - Web pages
    http://www.gnu.org/software/make/manual/make.html

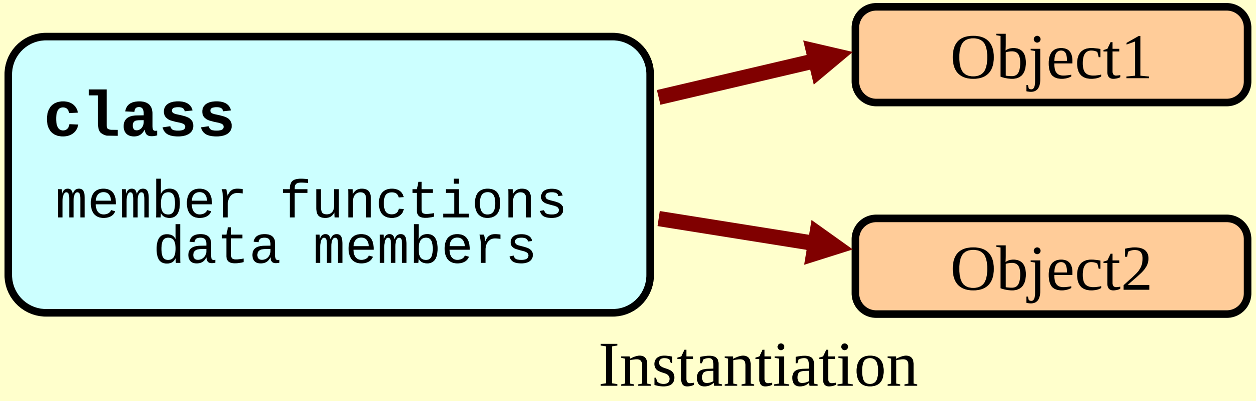
Make Files

```
# S. Allwood-Spiers
# A Makefile to build FileIO.exe

CC=g++
TARGET=FileIO
OBJECTS=main.o FileIO.o

$(TARGET).exe: $(OBJECTS)
	@echo "**"
	@echo "** Linking Executable"
	@echo "**"
	$(CC) $(OBJECTS) -o $(TARGET).exe

clean:
	@rm -f *.o *~

veryclean: clean
	@rm -f $(TARGET).exe
```

```
%.o: %.cc
	@echo "**"
	@echo "** Compiling C++ Source"
	@echo "**"
	$(CC) -c $(INCFLAGS) $<
```

- Provided the file is called Makefile, just type make to build
- make without any arguments builds the default target

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.

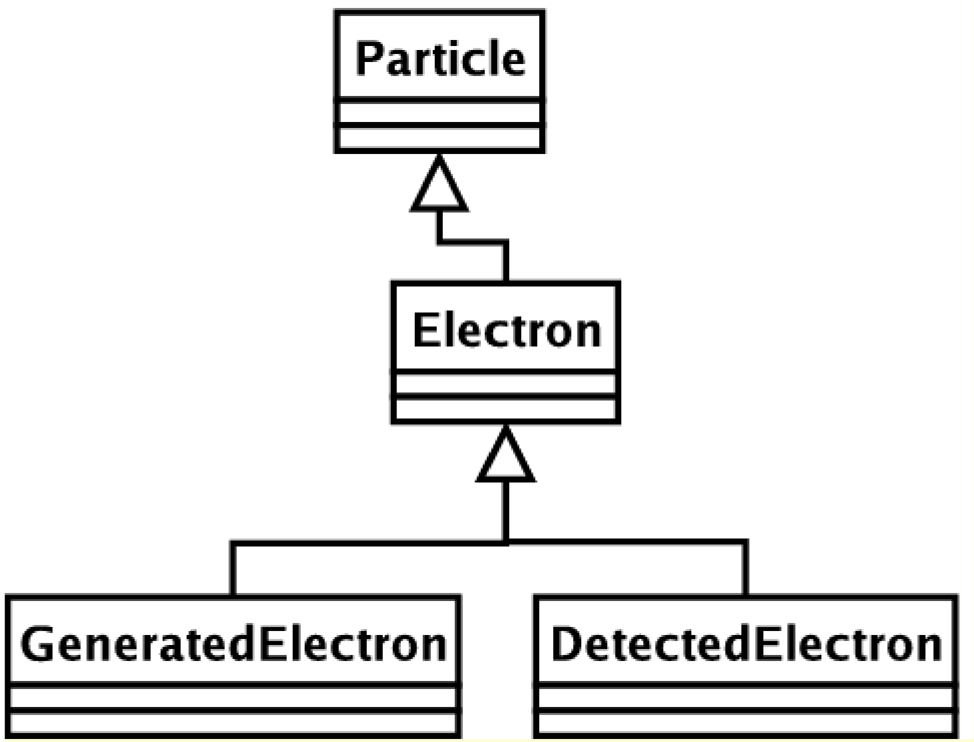
DetectedElectron:

- 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.

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

Extract from ex2/Parent.cc

```cpp
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. If a function needs to return the object (or a reference to the object) that it is working with:

```
return *this;
```

(Also see backup slides and example 2)

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];
};
```

x+y is equivalent to:

x.operator+(y)

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);


particle3.getFourVector(fourvector3);
for (int i=0;i<4;i++){
    cout << "fourvector3[" << i << "]="
         << fourvector3[i] << endl;
}
cout << "particle 3 mass = " << particle3.getMass() << endl;
cout << "particle 3 pt = " << particle3.getPt() << endl;

delete particle1;
delete particle2;
```

Extract from ex3/main.cc

Streams

- A stream is an object that characters can be inserted to (e.g. cout) or extracted from (e.g. cin).
- Streams provide a uniform basis for input and output independent of device
- Streams allow access to i/o devices, e.g.:
  - files stored on a hard drive
  - the terminal or console
  - a printer
  - a database

Output File Streams

```
#include <fstream>

using namespace std;

void fileWrite(char *filename) {
  ofstream file(filename);

  for(int i=1;i<=20;i++) {
    file << i;
    if(i%5==0) {
      file << endl;
    }
    else {
      file << " ";
    }
  }
  file.close();
}
```

Extract from examples 1 FileIO.cc

Input File Streams

```
#include <fstream>
...
void fileRead(char *filename) {
  int i;
  ifstream file(filename);

  if(!file) {
    cerr << "Error: could not open " << filename << endl;
  }
  else {
    cout << "Reading file " << filename << endl;
    while(!file.eof()) {
      file >> i;
      cout << i << " ";
      if(i%5==0) cout << endl;
    }
    file.close();
  }
}
```

Extract from FileIO.cc

Constructors

- When the constructor is invoked,
  1. Memory is allocated for the object
  2. The members are initialized
  3. The body of the constructor is executed.
- The class members can be initialized in the constructor using an initializer list, before the body of the function be executed

```
Parent::Parent(int id, double mass)
{
   m_id = id;
   m_mass = mass;
}
```

```
Parent::Parent(int id, double mass): m_id(id),
m_mass(mass)
{
}
```

All references and const attributes must be initialized in this way

Copy Constructor

- A constructor whose only argument is a reference to an object of the same kind is called the *copy constructor*

```
DataContainer::DataContainer(const DataContainer& dataContainer) {
. . .
}
```

- The copy constructor is invoked when a copy of an object is made:
  - when an object is initialized by assignment:

    DataContainer container2 = container1;
  - when an object is passed by value to a function
  - when an object is returned by a function
- If a copy constructor is not provided explicitly by the user, the compiler will provide one. This will copy the data members – which may not be what you need if the class has pointer data members (it will copy their addresses).

Exercises

- Session 2:
  - Download examples from My.SUPA
  - Build and test examples
  - Attempt section 2 problems 1 & 2.
- Tutorial for session 2:

Monday 5th November 11am

Room 320 Kelvin Building

- Deadline for Section 1 problem 1: 2nd November.

  (Section 1 problems 2 and 3 not assessed)
- Deadline for Section 2 problem 1 and 2: 16th November

  (Section 2 problem 3 moved to after lecture 3).

Extra Slides

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