SUPA Graduate C++ Course

Lecture 3

Tom Doherty
University of Glasgow

Issues from Section 1 and 2: 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.
}
```

Issues from Section 1 and 2: Constructors

- Constructor: builds objects belonging to a class and initializes the data members.
- e.g. Lecture 2, example 1: `BasicParticle particle1(myfourvector);`
- invoked the constructor:

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

- 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.

Issues from Section 1 and 2: Constructors

- The class members can be initialized in the constructor using an initializer list, before the body of the function be executed

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

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

- All references and const attributes must be initialized in this way

Constructors and inheritance

When the constructor of a derived class is called, the order is:

- Allocate memory for the object (base class and derived class members)
- Call the base class constructor to initialize the base class parts of the object
- Initialize the members of the derived class specified in the initialiser list
- Execute the body of the derived class constructor.

If we want to call anything other than the default base class constructor, we need to specify this in the initializer list.

Issues from Section 1 and 2: 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).

Comments on problem 2

- Operator overloading: use only when the definition of the operator would be obvious to all users. Think of the user
  - if you provide +, you should also provide +=.
  - If you provide ==, you should also provide !=.
- Example of operator overloading in problem 2 was to introduce you to the concept: understand when an assignment operator is being called and when a copy constructor is being called.
- If the copy constructor and assignment operator provided by your compiler are adequate for your class, then don't provide your own.
- In general, if you need to overload the assignment operator you must also provide a copy constructor.

Recap lecture 2:

- Classes: objects, constructors/destructors, object communication, operator overloading…
- Inheritance (so far):
  - Generic base class, e.g. BasicParticle, with declarations and definitions for some public methods.
  - Derived class, e.g. DetParticle, inherits these methods exactly as they are, and also has some extra methods of its own.

Lecture 3 Overview

- Polymorphism
  - Basics
  - Interfaces
- Templates
  - Basics
  - The Standard Template Library (STL)
    - Introduction
    - Complex Numbers
    - Vectors
    - Iterators
    - Algorithms
    - Strings

Polymorphism

- Dynamic member function resolution within an inheritance structure.
- Requires:
  - Inheritance
  - A `virtual` member function in the base class
  - A method of the same name and parameter types in the derived class
  - Pointers or references are used to access the created object.

Polymorphism Example

- Base class called Shape
- Two more specified derived classes – Square and Triangle
- Define a virtual member function called Draw
- Provide individual implementations of Draw in the Square and Triangle classes
- Create a pointer to a Square or Triangle object but give it a type called Shape – For example: **Shape *p1= new Triangle;**
- The static type of the p1 object (at compile time) is Shape
- The dynamic type of p1 determined at run time is Triangle
- If we use Draw at run time – the Triangle implementation of Draw is used

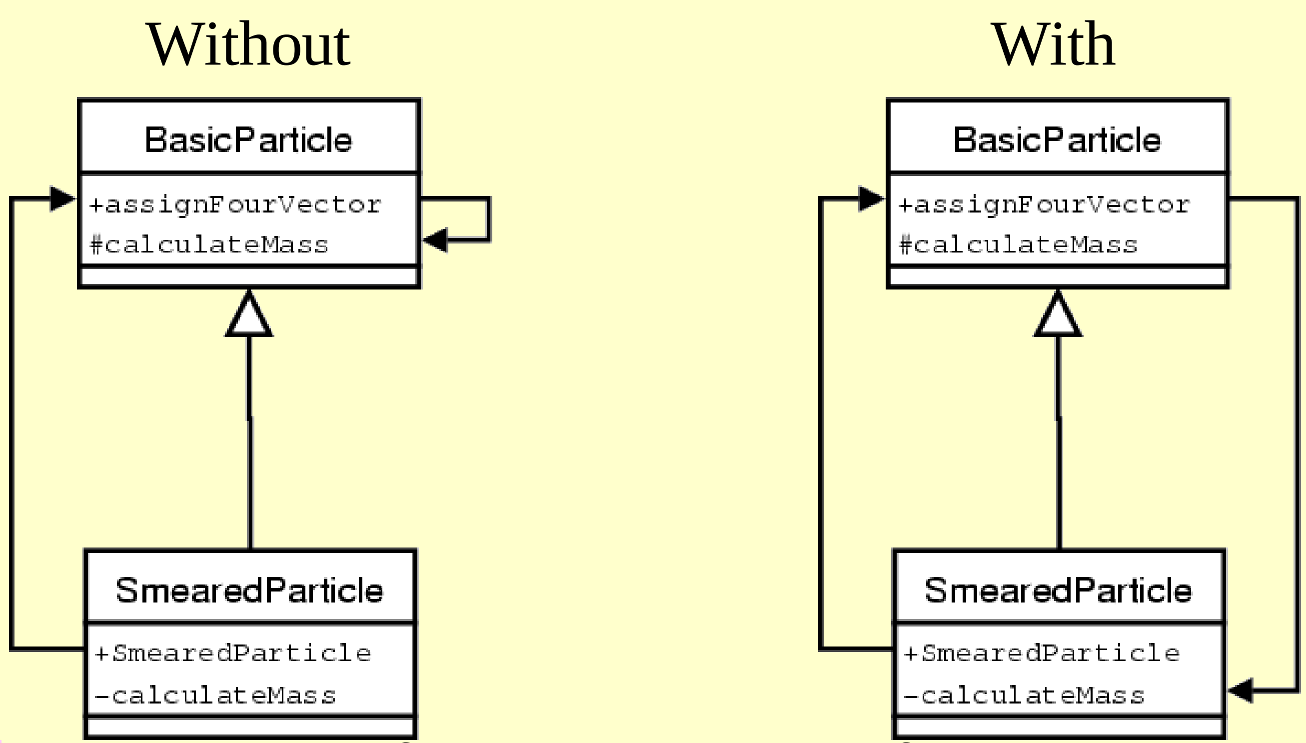
Polymorphism

- A virtual member function is selected by the type of the object that the pointer points to (resolved at run time).
- Small overhead required: look up table for dynamic member function resolution

An Example of Polymorphism

There's a public member function of our base class called assignFourVector. From within assignFourVector, another member function, calculateMass, is called.

We would like calculateMass to be different in our derived class and our base class – need Polymorphism.



A simplified UML version of example 1

An Example of Polymorphism

```
class BasicParticle {

protected:
  virtual void calculateMass();
};
```

Extract from ex1/with/BasicParticle.hh

```
class SmearedParticle: public BasicParticle {

private:
  virtual void calculateMass();
};
```

Extract from ex1/with/SmearedParticle.hh

An Example of Polymorphism

```
#include "BasicParticle.hh"
#include "SmearedParticle.hh"

using namespace std;

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

  BasicParticle *basicParticle =
      new BasicParticle(fourvector1);
  SmearedParticle *smearedParticle =
      new SmearedParticle(fourvector1);

  cout << basicParticle->getMass() << endl;
  cout << smearedParticle->getMass() << endl;
```

Extract from ex1/with/main.cc

An Example of Polymorphism

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

Extract from ex1/with/SmearedParticle.cc

- In this example:
  - The SmearedParticle constructor is calling a function in the base class.
  - The function in the base class is calling calculateMass in the derived class SmearedParticle.

Pure Virtual Functions

```
virtual void calculateMass() = 0;
```

- No implementation is given for pure virtual functions of a class.
  - Implementation must be provided in a derived class.
- An abstract base class containing only pure virtual functions is called an interface.
  - Allows code to be written that operates on interface member functions.

Interfaces

```
int main(int argc, char *argv[]) {
  ...
  IDataRecord *dataRecord;
  if(!strcmp(argv[1],"-a")) {
    dataRecord = new AsciiRecord("ascii_file.txt",10);
  }
  else if(!strcmp(argv[1],"-b")) {
    dataRecord = new BinaryRecord("binary_file.bin",10);
  }
  ...
  fillRecord(dataRecord);
  ...
}

void fillRecord(IDataRecord *record) {
  int arr[] = {1,2,3,4,5,6,7,8,9,10};
  record->appendRow(arr);
}
```

If record is a pointer to a BinaryRecord object, the member function for BinaryRecord is called.

Extract from ex2/main.cc

Interfaces

```
class IDataRecord {
 public:
  virtual int appendRow(int *rowData) = 0;
};
```

Extract from ex2/IDataRecord.hh

```
#include "IDataRecord.hh"

class BinaryRecord : public IDataRecord {
 public:
  BinaryRecord(char *filename, int columns);
  ~BinaryRecord(void);
  virtual int appendRow(int *rowData);
...
```

Extract from ex2/BinaryRecord.hh

Virtual Destructors

- Uses polymorphism to destroy objects within an inheritance structure in order.
  - If α inherits from β and an object of α class in instantiated via new, then calling delete on a pointer to the α object will call both α and then β destructors
- Special case of polymorphism since the name of the destructors is not the same for each class.
  - See text books for more information

Introducing Templates

- Templates allow code re-use where the same functionality is needed to operate on many different classes or types.
  - Templates provide code generation
- Can write Class and function templates
  - This course only looks at class templates.

Using a Class Template

```
Array<int> arrayInt(N);
Array<double> arrayDouble(N);

for(i=0;i<N;i++) {
  arrayInt.setElement(i,i);
  arrayDouble.setElement(i,(double)i/N);
}
```

Extract from ex3/main.cc

- Syntax “class name” <type1, type2,...> object
- Once an object has been instantiated call member functions as normal

Class Template Declaration

```
template <class T> class Array {
public:
  Array(int);
  ~Array(void);
  int getSize(void);
  T getElement(int );
  void setElement(int , T);

protected:
  T *m_array;
  int m_size;
};

/* Templates instantiations needed by g++ */
template class Array<char>;
template class Array<int>;
template class Array<float>;
template class Array<double>;
```

Extract from ex3/Array.hh

Class Template Implementation

```
template <class T> Array<T>::Array(int size) {
  m_array = new T[size];
  m_size = size;
}

template <class T> T Array<T>::getElement(int element) {
  if(element<m_size && element>=0) {
    return m_array[element];
  }
  else {
    return 0;
  }
}

template <class T> void Array<T>::setElement(int element, T value) {
  if(element<m_size && element>=0) {
    m_array[element]=value;
  }
}
```

Extract from ex3/Array.cc

Standard Template Library (STL)

- Contains a number of class templates, providing:
  - Data containers of many types
    - Iterators to access the elements
    - Types of container more suitable to some tasks than others
  - General purpose and numeric algorithms
  - Complex numbers

STL Complex Numbers

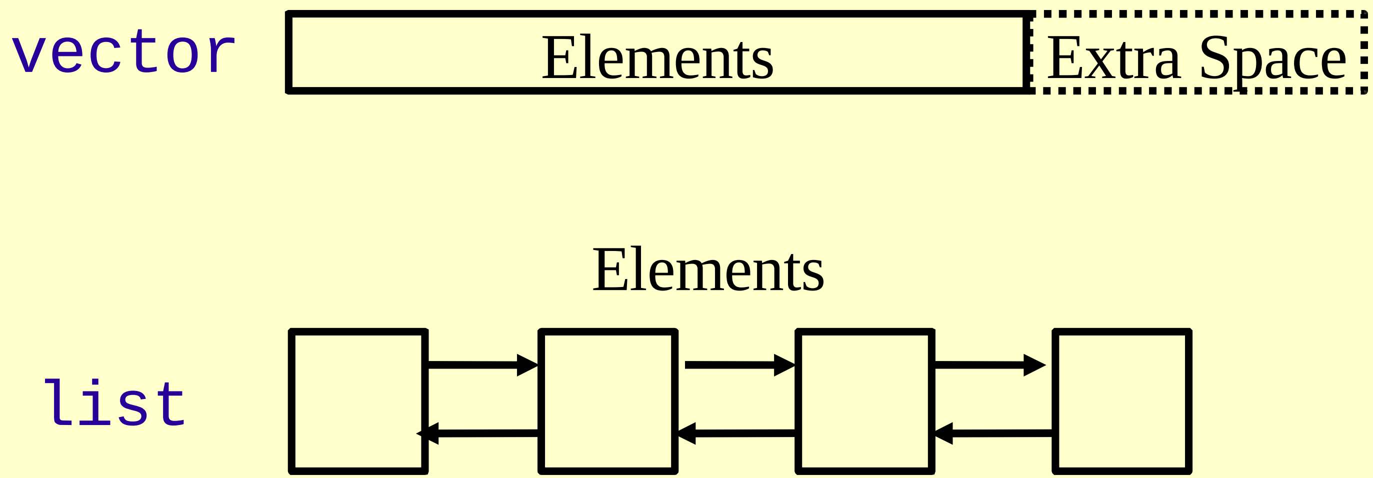
```
#include <complex>

int main() {
  std::complex<float> complexFloat(3,4);
  std::complex<double> complexDouble(1,0);
  std::cout << complexDouble << std::endl << std::endl;
  std::cout
     << complexFloat*(std::complex<float>(complexDouble))
     << std::endl;
```

Extract from ex4/Complex.cc

- All the standard mathematical functionality
- Ability to cast
- Stream interpretation

Choosing an STL Container



- Vector allows random access iterator and [] notation
- List insertion at any point is a constant

STL Vectors

- Larger than an array.
  - Require header information to keep track of elements
- Flexible size
  - Container manages memory allocation
- Elements can be accessed with an index `[i]` or via an iterator.

STL Vectors

```
#include <vector>

int main() {
  std::vector<int> intVector;

  std::cout << "  >> vector size=" << intVector.size() << std::endl;
  for(int i=0;i<NUM;i++) {
    intVector.push_back(i);
    std::cout << "  >> vector size=" << intVector.size()
              << std::endl;
  }

  do {
    std::cout << "  >> Popping element with value="
              << intVector.back() << std::endl;
    intVector.pop_back();
  } while(!intVector.empty());
```

Extract from ex5/PushAndPop.cc

STL Iterators

- Type of smart pointer
  - Syntax is very similar but not identical to that of a pointer
  - Relationship between Iterator and Container is similar to that of a pointer and an array
    - But, no stream interpretation for memory address.
- Use to navigate around elements of container.

STL Iterators

```
#include <iostream>
#include <list>

using namespace std;

int main() {
  list<char> charList;
  list<char>::iterator itr;

  itr = charList.begin();

  cout << endl;
  while (itr != charList.end()) {
    cout << *itr << " ";
    itr++;
  }
  cout << endl;
```

Extract from ex6/Iterators.cc

STL Algorithms

vector(input_iterator start, input_iterator end);

```
#include <vector>
#include <algorithm>

int main() {
  int numberList[] = {1,4,2,5,7,2,5,4,9,4,2,7,8,0};
  std::vector<int> numbers(numberList,numberList+
      sizeof(numberList)/sizeof(int));
  std::vector<int>::iterator first;
  std::vector<int>::iterator last;

  first = numbers.begin();
  last = numbers.end();
  std::sort(first,last);
```

Extract from ex7/Algorithms.cc

- Many different algorithms:
  - explore reference material or header file.

Exercises

- Session 3 examples:
  - Download examples from My.SUPA
  - Build and test examples
- Tutorial: Monday 22th November, 10am
- Solutions due Monday 6th of December.