SUPA Graduate C++ Course

Lecture 3

S. Allwood-Spiers
University of Glasgow

Recap lecture 2:

Classes:

- objects,
- constructors/destructors,
- new and delete,
- object communication,
- operator overloading

Lecture 3 Overview

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

Inheritance

- Section 2, 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.

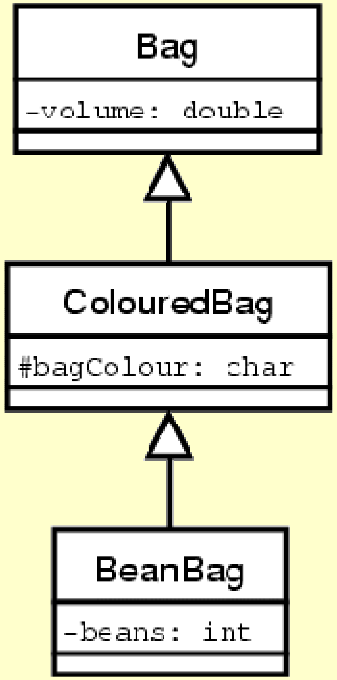
Inheritance

```
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

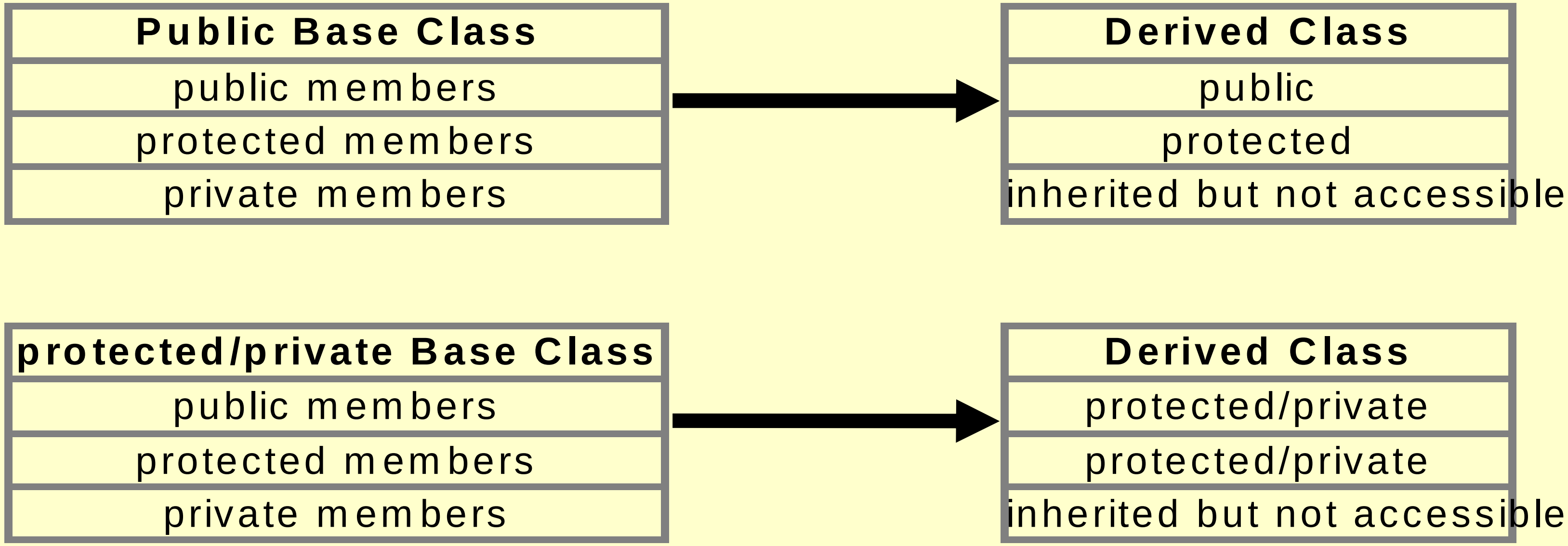
```
Bag bag(30.0);

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



Public Base Class	Derived Class
public members	public
protected members	protected
private members	inherited but not accessible

protected/private Base Class	Derived Class
public members	protected/private
protected members	protected/private
private members	inherited but not accessible

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.

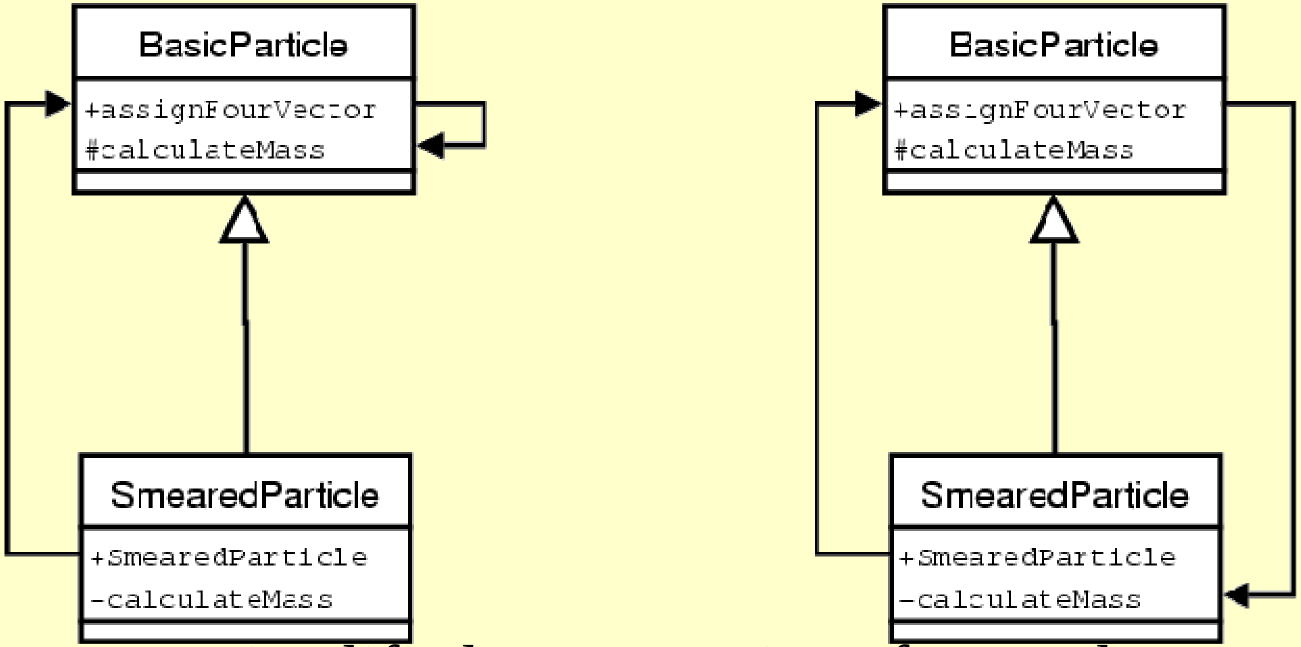
Polymorphism

- 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.


Without

With

A simplified UML version of example 1

An Example of Polymorphism

```
class BasicParticle {

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

In base class:
virtual is required to allow polymorphism

Extract from ex1/with/BasicParticle.hh

```
class SmearedParticle: public BasicParticle {

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

Extract from ex1/with/SmearedParticle.hh

In derived class:
virtual is good practice to show that polymorphism is being used

An Example of Polymorphism

```cpp
#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

Pointer to a derived class can be assigned to a base class pointer

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

Extract from ex2/main.cc

Interfaces

IDataRecord is an interface (all functions are pure virtual):
No implementation is defined.

```
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

BinaryRecord inherits from IDataRecord, and defines an implementation for appendRow (in BinaryRecord.cc).

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>;
```

Here “T” denotes a type.

Allowed template instantiations – i.e. “T” can be char, int, float or 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.hh

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

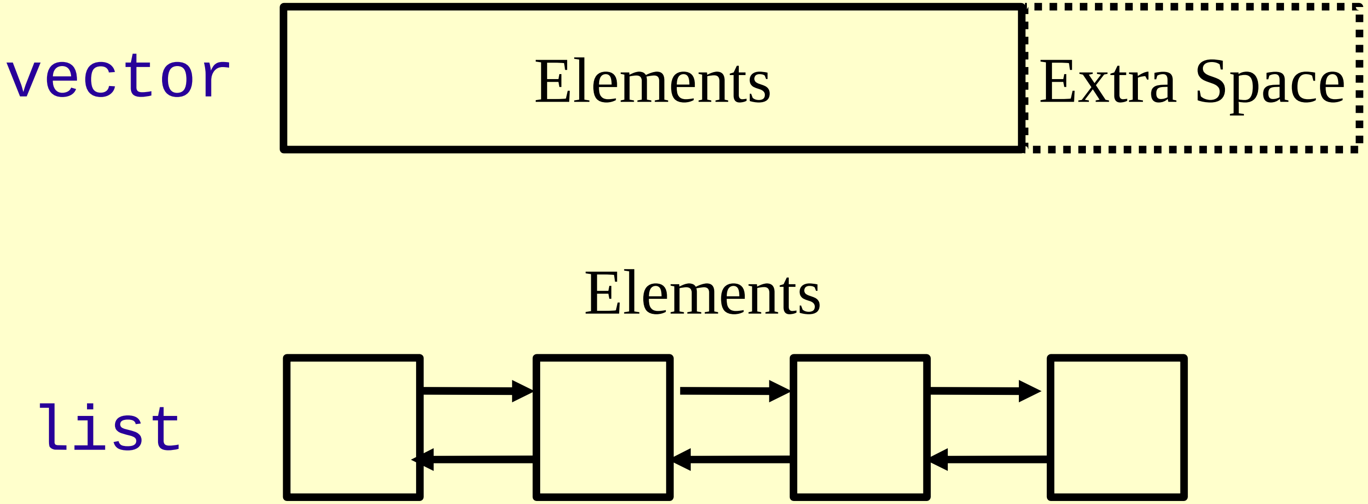
STL Complex Numbers

```cpp
#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

Elements

Extra Space

Elements

list

- 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());
```

the number of elements in intVector

append an element to intVector

reference to the last element in intVector

remove the last element in intVector

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;
```

iterator that denotes the 1st element of charList

iterator that denotes one past the last element in charList

the element to which itr refers.

Extract from ex6/Iterators.cc

STL Algorithms

```
#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);
```

vector(input_iterator start, input_iterator end);

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
  - Deadline Section 2 Problems 1 and 2: 16th November
- Tutorial: Monday 19th November, 11am, room 320 Kelvin Building.