C++ II

Bálint Joó (bjoo@jlab.org)

Jefferson Lab, Newport News, VA

given at

Jefferson Lab Graduate Lecture Series

July 19, 2006

Recap from Last Time

- We discussed C or Fortran 77 in C++
  - basic types, loops, conditionals, references & pointers
    - native arrays ARE pointers
  - classes
    - private data, public functions
      - Information hiding and encapsulation
    - inheritance
      - of functions and data
      - virtuality (overriding base class members properly)
      - pure virtuality (specifying interfaces)

Recap of Inheritance

- An inheriting class
  - gets copies of the functions and data of the base class. Private data
  - is called a derived class
- To manipulate a derived class as if it was the base class (polymorphism) we must declare override functions to be virtual. Base classes provide default implementations
- When an overriding function cannot provide a default we declare it pure virtual (= 0) - it is an interface

Modularizing the code

- We don't want to write large 'mammoth' programs
- We would like to split the code up into small pieces
  - eg: 1 or 2 files per class
  - a few short main file to 'drive it all'
- C++ features for supporting this:
  - Separating declarations from definitions
  - Include guards
  - Separate compilation
- O/S features: object files, libraries

Jefferson Lab
Exploring the Nature of Matter

Name mangling

- How do we distinguish between functions that have the same name?
  - void foo(int x) in class X
  - void foo(double x) in class X
  - void foo(int x) in class Y
- C++ 'mangles' the names into something unique
  - _ZN1X3fooEi - class X, void foo(int x)
  - _ZN1X3fooEd – class X, void foo(double x)
  - _ZN1Y3fooEi – class Y, void foo(int x)

Preventing Mangling

- We can stop the compiler from mangling a name:

```
extern "C" {
   void foo( double x)  { cout << x;  }
};
```

- Useful for calling/providing non C++ routines
  - we write C++ routines with extern "C"
    - can be used from C or assembler
  - we can call C/Fortran/Assembler routines
    - we declare them as extern "C"

Qualification/Disambiguation

- Mangling is great for the compiler for internal use but not for humans
- We can instead disambiguate by using the :: qualifier
  - void X::foo(int x)
  - void X::foo(double x)
  - void Y::foo(int x)
- We can separate declaration and definition of functions in classes using the disambiguator
- Move the declarations to separate files for reuse

Example - separate compilation

originally was:

```
class X {
  public:
    void foo(int x) { cout << x ; }
    void foo(double x) { cout << x; }
};
```

now move declarations into file classX.h:

```
#ifndef CLASSX_H         /* Trigger guard. So it is included */
#define CLASSX_H         /* only once. */

class X {                // Declarations only
  public:
    void foo(int x);     // Declaration – no function body
    void foo(double y);  // Declaration – no function body
};
#endif                   /* End of trigger guard */
```

and now move definitions to file classX.cc :

```
#include "classX.h"      // Include the declarations

#include <iostream>
using namespace std;

void X::foo(int x) {  cout << x;  }    // Definition
void X::foo(double x) { cout << x; }   // Definition
```

Example continued

file main.cc:

```
#include <iostream>
using namespace std;

#include “classX.h”   // Include the declarations from the .h file

int main(int argc, char *argv[])
{
    X class_X;         // Can now use the classes in the .h file
    class_X.foo(5);
}
```

- Compile as:
  - g++ -o program main.cc classX.cc
- Or can do it piecemeal:
  - g++ -c classX.cc (This makes an object file: classX.o)
  - g++ -o program main.cc classX.o (Link it together)

Libraries

- Can compile classX.cc into a library (UNIX/Linux)
  - g++ -c classX.cc
  - ar -cru libclassX.a class.o
  - ranlib libclassX.a
  - Install
    - libclassX.a into /foo/lib/libclassX.a
    - classX.h into /foo/include/classX.h
  - Use library as
    - g++ -I /foo/include -o program main.cc -L/foo/lib -lclassX

More about Libraries

- This is technically compiler specific but is mostly standard on UNIX
  - -I flag tells compiler in which directory to look for .h files for inclusion
  - -L flag tells compiler/linker in which directory to look for libraries (libX.a files) for linking
  - -l flag tells compiler which libraries to link to the program
    - -lfoo will try to link libfoo.a ('lib' prepended '.a' appended internally

Summary of Physical Modularization

- Classes allow modularization of concepts
- Separation of declarations and definitions allows
  - separate compilation
  - physical modularisation into
    - "include files" (.h files)
    - libraries (libXXX.a files)
- True for other languages too
  - eg separate compilation in C etc.
- Libraries from vendors typically delivered this way

Namespaces

- Suppose you want to use a class called vector
- BUT you already have a different class also called vector that behaves differently from your class?
- OR you may want to write a function called
  - void print(int x)
- BUT there is already a function in a library called
  - void print(int x)
- which prints x in a different way from how you want

Namespaces

- Clearly it is just the names of the functions/classes that clash
- Solution 1: Use a different name -> avoid clash
- Solution 2: Use a namespace
- A namespace is:
  - An extra level of indirection on names
  - different from classes (no objects are involved)
  - it just allows you to modularize the space of your function or class names

Namespace Example

```
#include <iostream>
using namespace std;

namespace Foo {
    void print(int x)  {
    cout << “Foo has one way of printing x: x = “ << x << endl;
   }
};

namespace Bar {
    void print(int x)  {
        cout << “A different way to print: x is “ << x << endl;
    }
};

int main(int argc, char *argv[]) {
    int x = 5;

    Foo::print(x);
    Bar::print(x);

    return 0;
}
```

Namespaces

- You can put anything with a name into a namespace
  - functions, classes, globals, structs etc
- You can get at names in the namespace using ::
  - like before, it qualifies the name
- There is a default namespace which needs no qualification
- You can import from one namespace into the default one using the: using namespace incantation
- I/O functions live in namespace 'std'

Another namespace example

```
#include <iostream>
using namespace std;  // Import names from std into default namespace

namespace Foo {
    void print( int x )  {

    // Note I don't need std::cout because 'std' has been imported
    cout << "Foo's way of printing x: x = " << x << endl;
   }
};

namespace Bar {
    void print(int x)  {
        // But I can explicitly qualify std
        std::cout << "A different way to print: x is " << x << endl;
    }
};

using namespace Foo;
int main(int argc, char *argv[]) {
    int x = 5;

    print(x);     // Will call Foo::print(int x)
    Bar::print(x);

    return 0;
}
```

Careful when using using

- If you import two namespaces that have the same names in them into the default namespace you may still get a clash

```
// Namespace clash example. Import both Foo &Bar

using namespace Foo;
using namespace Bar;

int main(int argc, char *argv[]) {
    int x = 5;
    print(x); // Error: Ambiguity

    return 0;
}
```

- C++ compiler produces error
- Use full qualification (eg Foo::print) to remove ambiguity

What's the use of namespaces

- Protection
  - Put your code in a namespace
  - isolate it from the names other packages use
- Makes your package more reusable too
  - Your names less likely to clash with other names
- Hide implementation details when not using classes
  - eg: in QDP++ we have QDPIO::cout
    - like std::cout except on a parallel machine only one processor writes

"It all works except in exceptional cases"

- Occasionally unexpected conditions can occur
  - Index out of range in []
  - Failure of new
  - Inability to open a requested file
  - Failure to convert one type to another type (casting)
- How to deal with this?
  - Print error message and exit (as seen in examples)
  - return an error status code (eg new returns 0)
  - "throw" an "exception"

What does it mean to "throw an exception"

- Program flow halts
- An object representing an exception is created
- This object is propagated up through the calling functions until someone "deals with it"
  - dealing with it is called "catching the exception" or handling the exception
  - execution continues from the handler
- If the exception is not handled by our program, the C++ runtime environment's handler catches it and then the program terminates

Example

create a string object to represent error and "throw" it

```
double& MyCheckingVector::operator[]( int index )
{
    if ( index >= size )  {
        std::string error_message="Index out of range";
        throw error_message;
       }

      return vector[ index ];
}

int main(int argc, char *argv[])
{
    MyVector vec(3);

    vec[0]=1.0;  vec[1]=2.0; vec[2] = 3.0;

    try {
       vec[5] = 5.0;
    }
    catch( const std::string& e) {
        std::cerr << " Caught exception: " << e << endl;
    }
    // execution continues here after catch
    vec[6] = 6.0;

    return 0;
}
```

"try{} catch{}" block means we expect an exception may be thrown. execution goes into "try"

the thrown "error message" is "caught" in the catch clause

Uncaught exception (no try{} catch{}) Handled by runtime (crash)

Exceptions are typed

- Exceptions throw objects of concrete types/classes
- Can have many catch() {} clauses to deal with different exceptions
- catch(...) matches any exception (catchall)

```
try {
    MyVector foo(5);
    foo[5] = 10;
}
catch( std::bad_alloc ) { // Handle allocation failures
    cerr << "new() failed" << endl;     exit(1);
}
catch( const string& e ) { // Handle an exception raised as a string
    cerr << "Caught a string: " << e << endl;  exit(2);
}
catch( ... ) { // Handle all other kinds of exceptions
    cerr << "Some (unknown) exception occurred" << endl ; exit(3);
}
```

More about exceptions

- The exceptions are objects belonging to classes
  - string, std::bad_alloc, std::bad_cast etc
- Can have hierarchy (inherit from each other)
  - eg: c++ standard exception (std::exception) is a base class of a hierarchy of exception classes
- The subject can get quite complex
  - When should we throw exceptions?
    - should we return a status code instead?
    - "Throw exceptions in exceptional situations!" see books

Templates

- Let us return to MyVector
  - it uses an array of doubles
  - but I may want to use floats (for whatever reason)
  - or even have vectors of integers.
    - Do I really have to duplicated the code for the class for each internal type?
    - I wish I could just “magically” replace the internal types somehow
  - YOU CAN! Using Templates

Templated Class

```
template< typename T > // T is what can be replaced later by a type
                       // of your choice
class MyVector {
private:
    T* vector;
    int length;
public:

  // Constructor (initFunction)
  MyVector(int size) : vector( new T [size] ), length(size) {}


  // Destructor (clean up function )
  ~MyVector(){ delete [] vector; length=0; }

  // Want to know length of vector for loops, but can't touch it
  // because it is now private. Here I return a copy.
  int getLength(void) const { return length; }

  // Array indexing - so I can treat vector like an array
  // This allows me to change the value in the vector (LHS of =)
  T& operator[]( int i ) { return vector[i]; }

  // Array indexing - this is read only access (RHS of =)
  const T& operator[]( int i )  const { return vector[i]; }

};
```

Using the templated class

```
#include <iostream>
using namespace std;

#include “myVector.h” // Put the myVector code into file myVector.h
                      // We include the definition here

int main(int argc, char *argv[] )
{
    MyVector<double> newVecD(3);    // A vector of doubles is created

    MyVector<float> newVecF(3);     // A vector of floats is created

    MyVector<string> newVecS(2);   // A vector of strings

    newVecs[2 = “String 1”;
    newVecS[3]= ”String 2”;

    for(int i=0; i < newVecS.getLength(); i++) {
       cout << “newVecS[“ << i << “] = “ << newVecS[i] << endl;
    }

}
```

Template functions

- You can also template functions

```
template < typename F >
void print( const F&  f) {
    f.printMyself() ;
}
```

- In this case the class F has to have a member function F::printMyself()
  - This is so called 'duck typing'
    - "If it walks like a duck and looks like a duck it is probably a duck"
- Otherwise the compiler will report an error

Specialization

- Can specify special behaviour depending on the template type (sort of a template version of a virtual function).
- This is called "Template specialization"

```
template < typename F >   // Deals with arbitrary type F
void print( const F&  f) {
    f.printMyself() ;
}

template<>                                        // Deals only with doubles
void print( const double& d) {
    cout << d;       // Special case: for doubles use <<, not printMyself()
}
```

- Template matching order: for some type T
  - first check specializations for match
  - then try more general case

Multiple templates, value templates

```
#include <iostream>
using namespace std;

template<typename T, int N>  // N is a Value template
class MyVector {
private:
  T vector[N];  // N known at compile time, so can do automatic allocation
public:
  T& operator[](int i) {
    return vector[i];
  }
  int getSize() {
    return N;
  }
};

typedef MyVector<float,4> Float4Vec;   // Different templates -> different classes
typedef MyVector<double,3> Double3Vec; // sctually different types

int main(int argc, char *argv[])
{
  Float4Vec f;

  f[0]=0;  f[1]=1; f[2]=2; f[3]=4;

  for(int i=0; i < f.getSize(); i++) {
    cout << "f[" << i<< "]=" << f[i] << endl;
  }
}
```

Template Type Magic

- We can do surprisingly many things with templates

```
template< typename T >
class DoublePrecisionType {         // Note: Empty Body (Base case)
};

template<>
class DoublePrecisionType< float > { // Specialisation for floats
public:
    typedef double Type_t;           // Double prec type of float is double
};

template<>
class DoublePrecisionType< double > { // Specialisation for doubles
public:
    typedef long double Type_t;       // Double prec type of double is long
};                                    // long double

int main( in t argv, char *argv[] )
{

 DoublePrecisionType<float>::Type_t  really_a _double;  // Type computation
 DoublePrecisionType<double>::Type_t a_long_double;

 DoublePrecisionType<int>::Type_t  an_error; // General class has no Type_t;
}
```

- Templates & compiler do computation on Types!!

A glance in the direction of...

- Generic Programming
  - DoublePrecisionType<T> is a so called "Traits Class"
  - Uses templates and type definitions to provide information (traits) about the class T
  - Can do more sophisticated things with templates...
    - ... but sadly beyond the scope of this lecture
  - Templates and generic programming underlie several important C++ libraries: Boost, Pooma, MTL etc
  - and of course also: QDP++ and Chroma for lattice QCD

The Standard Template Library (STL)

- A set of templated classes for various kinds of useful advanced data types (ADTs)
  - Vectors
  - Maps
  - Sets
  - Lists
- Mostly containers and their manipulation
- Look here first if you need an ADT
- Details at eg: http://en.wikipedia.org/wiki/Standard_Template_Library

Vectors

- A 'growable' vector

```
#include <iostream>
#include <vector>

using namespace std;
int main(int argc, char *argv[])
{
    vector<int> v;
    v.push_back(4);
    v.push_back(5);
    v.push_back(6);

    for(int i=0; i < v.size(); i++) {
        cout << "Element v[" << i << "]=" << v[i] << endl;
       }

    for(vector<int>::iterator iter=v.begin();
         iter != v.end();
         iter++ ) {
            cout << *iter << endl;
    }
}
```

STL iterators

- An iterator is a uniform interface to the elements in an STL container
- Abstracts away indexing
  - vector<int>::iterator iter = v.begin(); // first element
  - vector<int>::iterator iter = v.end(); // last element
- Pointer like behaviour
  - *iter; // value of the iterator
- Move amongst elements using
  - iter++ (forward), iter-- (backward)

STL Maps

- A Map is an associative container to store pairs of
  - keys (indices, not necessarily a numerical ones) AND
  - values belonging to the keys
  - keys have to be unique (no duplicates keys)

```
#include <iostream>
#include <map>
using namespace std;

int main(int argc, char *argv[])
{
  map<string, int> the_map;  // The key type is string, the value type is int
  the_map[ “foo” ] = 5;
  the_map[ “bar” ] = 6;

  cout << “the value associated with bar is “ << the_map[“bar”] << endl;

  for( map<string, int>::iterator iter=the_map.begin(), iter != the_map.end(), iter++) {
        //                         Key                            Value
        cout << “String: “ << (*iter).first << “ Int: “ << (*iter).second << endl;

     }
}
```

Notes on maps

- We don't know what underlying container is
  - Depends on the implementation of STL
    - can be a tree – logarithmic retreival
- Iterator ordering is implementation dependent
  - Keys are not assumed to be ordered
    - Ordering can be order of insertion
    - Or order of tree traversal (eg: alphabetic etc)
- Use to implement Factories

Factory I: The particles

```
#include <iostream>
#include <map>
using namespace std;

// The Particle Interface
class Particle {
public:
  virtual const string getName(void) const = 0; // Pure virtual
  virtual double getMass(void) const = 0;
};

// Two Particle Implementations
class Photon : public Particle {
public:
  const string getName(void)  const { return string("Photon"); }
  double getMass(void) const  { return 0; }
};

class Electron : public Particle {
public:
  const string getName(void) const { return string("Electron"); }
  double getMass(void) const { return 0.51100; }
};
```

Particles II: Building The Factory

```
// A function to 'create' a photon object
Particle* makePhoton(void)
{
  return new Photon();
}

// A function to 'create' an electron object
Particle* makeElectron(void)
{
  return new Electron();
}

// The factory is a map between a string name and a creation function
//
// Particle* (*)(void)
//
// is the C++ (proto)type of a function
// which takes no parameters and returns a Particle*
// ie: our creation functions
static map< string, Particle* (*)(void) > the_factory;

// Make association between names and creation functions
void setup()
{
  the_factory["ELECTRON"] = makeElectron;
  the_factory["PHOTON"]   = makePhoton;
}
```

Producing With The Factory

```
int main(int argc, char* argv[])
{
  setup();  // Build the factory

  // Read Particle Name from the user
  cout << "What Particle shall I create? " << endl;
  std::string particle_name;
  cin >> particle_name;

  // Create the particle of your choice with the factory
  // remember factory contains functions, which we have to call
  // hence the () at the end
  Particle* your_particle = the_factory[ particle_name ]();

  // Print properties
  cout << "Particle Name: " << (*your_particle).getName() << endl;
  cout << "Particle Mass: " << (*your_particle).getMass() << endl;

  delete your_particle;
  return 0;
}
```

Try running the program with inputs: ELECTRON or PHOTON
(for simplicity there is no checking that the value asked for is in the map. So using eg PION will cause this program to fail)

Use of factories

- Maintain separately (in separate files)
  - The classes themselves
  - The method of creation (the_factory and setup)
  - Uniform creation of objects
- Extensible
  - Write new classes as needed
  - Only need to update setup function with new creation method
  - Main code using factory is unchanged

Design Patterns

- Factory is a technique that is commonly used in object oriented programming (Java too)
- It is what is known as a design pattern
  - an idiom that solves a particular programming problem
  - not exactly an algorithm, not exactly a class
- Design patterns originally catalogued by the so called “Gang of Four”: Gamma, Heim, Johnson & Vlissides in their classic book: Design Patterns: Elements of Reusable Object Oriented Software

Summary Of Lecture

- We have recapped Classes, Objects & Virtual Functions
- We introduced code modularisation through namespaces and separate compilation
- Looked at Exceptions
- Introduced Templates and the STL
- Introduced Design Patterns through an STL map implementation of an Object Factory

Topics for the interested

- Building software (make, autoconf, automake)
  - eg: R. Mecklenburg: Managing Projects with GNU Make (O'Reilly)
- Templates, Template Metaprogramming, Generic Programming
  - Boost, Pooma and MTL Libraries (Google them)
  - D. Abrahams, A. Gurtovoy: C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond (Addison Wesley)

More topics for the interested

- Design Patterns and implementing them with Templates:
  - Gamma, Heim, Johnson & Vlissides: Design Patterns: Elements of Reusable Object Oriented Software
  - A. Alexandrescu: Modern C++ Design, Generic Programming and Design Patterns Applied
    - Both Published by Addison Wesley
- Other Object Oriented Languages
  - Python: An Object Oriented “scripting” language
    - http://www.phython.org

Software Design and Engineering

- Hunt & Thomas: The Pragmatic Programmer: From Journeyman to Master
- Software Carpentry:
  - Lectures on Scientific Programming in Python at
    - http://www.swc.scipy.org
- A lot of great books are available to you free of charge through the Safari Tech Bookshelf of the JLAB (eg: Most O'Reilly Titles)
  - http://www.jlab.org/div_dept/cio/IR/library/copyright1.html?site=safari