Concepts of Object Oriented Programming

Dr. Axel Kohlmeyer

Senior Scientific Computing Expert

Information and Telecommunication Section
The Abdus Salam International Centre
for Theoretical Physics

http://sites.google.com/site/akohlmey/

akohlmey@ictp.it

The Abdus Salam International Centre for Theoretical Physics

Workshop on Computer Programming and Advanced Tools for Scientific Research Work

Defining Object Oriented Programming

- Data is combined with functions modifying it into a “class” => data is acted on only indirectly
- A class can be extended through “Inheritance” => data and functions can be added
- A class can be modified with “Polymorphism” => data or functions can be replaced
- A class can decide at runtime which method(s) to execute from inside a class hierarchy
  => “Dynamic dispatch” or “Late binding”
  => calling code only needs to know parent class

Nomenclature in OOP

- Class: the module combining data and functions
- Member: data element of a class
- Method: function inside a class
- Object: instance of a class
- Constructor: method to initialize an instance of a class; called implicitly on creating the instance
- Destructor: clean up an instance of a class; automatically called when an instance is deleted
- Overloading: methods differ in arguments only

More Nomenclature in OOP

- Class hierarchy (class inheritance tree with child and parent classes)
- Virtual function: a method flagged so it can be overridden (not extended) by a derived class
- Pure method: a virtual function that must be overridden by a derived class
- Abstract class: a class with only pure methods => used to define an interface only
- Static member: refers to the same storage for all instances (e.g. to generate unique object ids)

Benefits of OOP

- OOP encourages modularity and consistency
- Side effects from changing data are controlled
- Separate interface and implementation
- Control visibility and read/write access to data, violations can be found be the compiler
- Top level code becomes terse (-> less errors)
- Natural semantics for stateful items
- More compile time checking of correct use

Problems of OOP

- Overhead of dynamic dispatch
- Objects get bloated by unneeded members
- Inefficient data access for caching, vectorization
- Flow of control scattered across source files, especially with very deep class hierarchies
- Inconsistent implementation (methods that have the same name don't do the same thing)
- Implicit actions (copy constructor, assignment operator) can become very expensive

What to Recommend?

- Like all good things: use in moderation; use OOP where it helps modularity, but not everything that can be an object needs to be
- At the upper level(s) functional programming (using collections of objects) is often cleaner
- Consistency of the interface:
  - Methods in a class should be as orthogonal as possible (no methods that do almost the same)
  - Methods with the same name do the same thing
- Access data with “getter” / “setter” methods

More Recommendations

- Pass objects as a const reference or pointer to avoid copying without need
- Avoid multiple inheritance if it is not obvious which method is inherited from where
- Use abstraction where details need not to be known, but do not hide what is important
- Object oriented programming is not bound to a specific programming language; some require less code to be written; the important part is sticking to the established conventions

Example 1a: Stack

```
class Stack {
  public:
    Stack() {top=-1;}
    void push(int val) {++top; data[top] = val;}
    int pop() {int rv=data[top]; --top; return rv;}
    int size() {return top+1;}
  private:
    int top, data[100];
};

/*************************************************************/
Stack a;
a.push(10); a.push(-2); a.push(30);
while(a.size() > 0) { std::cout << a.pop() << std::endl;}
```

Constructor

Methods

Members

Instance

Example 1b: Stack

```
class Stack {
  public:
    Stack() {top=-1;}
    void push(int val) {data[top++] = val;}
    int pop() {return data[top--];}
    int size() {return top+1;}
  private:
    int top, data[100];
};
/*********************************************************/
Stack *b = new Stack;
b->push(10); b->push(-2); b->push(30);
while(b->size() > 0) { std::cout << b->pop() << std::endl;}
delete b;
```

Instance created dynamically

Example 1c: Stack

```
class Stack {
  public:
    Stack(const Stack &in) {
      for (int i=0; i <= in.top; ++i) { data[i] = in.data[i]; }
      top = in.top; }
    const Stack &operator =(const Stack &in) {
       for (int i=0; i <= in.top; ++i) { data[i] = in.data[i]; }
       top = in.top;  return *this;}
    (...)
/*****************************  Reference to itself  *********/
Stack a, *b, c;
a.push(10); a.push(-2); a.push(30); b = new Stack(a); c=a;
while(b->size() > 0) { std::cout << b->pop() << std::endl;}
while(c.size() > 0) { std::cout << c.pop() << std::endl;}
```

Copy Constructor

Assignment Operator

Example 2a: Animals

```
class Animal {
  public:
     void say() { std::cout << word << std::endl; }
  protected:
      const char *word;
};

class Dog: public Animal { public: Dog() { word = “woof!”; } };
class Cat: public Animal { public: Cat() { word = “meow!”; } };
/*************************************************/
Animal *a, *b;
a = new Dog; b = new Cat;
a->say();
b->say();
```

Child class

Parent class

Method inherited from parent

Example 2b: Animals

```
class Animal {
  public:
     void say() { std::cout << word << std::endl; }
  protected:  const char *word;
};
class Dog: public Animal {
  public: Dog() { word = “woof!”; tail = 1;}
  void wag() { tail = -tail; }
  private: int tail; };
/*****************************************************/
Animal *a;
a = new Dog;
a->say();
a->wag();
```

Protected == visible to derived class, but not public

ERROR: wag() is not a method of Animal

Example 2c: Animals

```
class Animal {
  public:
     virtual void say() = 0;
};

class Dog: public Animal { public:
  virtual void say() {std::cout << “woof!\n”;}};
class Cerberos: public Dog() { public:
  virtual void say() {Dog::say(); Dog::say(); Dog::say();};
/*************************************************/
Animal *a, *b;
a = new Dog; b = new Cerberos;
a->say();
b->say();
```

Pure method, abstract class

Use scoping to refer to parent

Real World Example: LAMMPS

- Highly parallel classical MD simulation code
- Original implementation in Fortran 77, then Fortran 90, now C++ (mostly C with classes)
- Main program: create instance of LAMMPS class, pass MPI comm and process input
  => easy to couple with other codes
  => easy to embed into Python
- (Plug: special session on LAMMPS development, Saturday March 23rd)

More on LAMMPS

- LAMMPS toplevel class uses 'composition' instead of inheritance, i.e. it contains instances of classes that manage different aspects of the simulation (atom data, output, communication, force computation, and so on)
- Those classes use base classes that define the interface for actual models via virtual functions
- e.g.: LAMMPS -> Force -> Pair -> PairLJCut PairLJCut contains the physics of a 12-6 LJ potential, Force and Pair most of the “ugly stuff”

And There is More...

- Classes PairLJCutOMP and PairLJCutGPU are derived from PairLJCut and replace/extend only those methods that are needed for OpenMP or GPU acceleration, respectively
  - Interface for all three computations is the same
  - Parameter input is inherited from parent
- Models can be added by deriving from Pair without knowing about the rest of the code => very easy to add new functionality
- Problem: copy-and-modify -> redundancy

Real World Example: Boolean Matrix

- Optimization / parallelization problem: C code for cluster analysis of (social) networks was too slow and required too much storage
- Performance and memory critical part was construction of a “relation matrix” with 1 or 0 whether a relation between nodes exists or not
- Data storage was 64-bit data type replaced with 8-bit data type => 2x faster
- Re-implemented using std::vector<bool> => 2x faster (transparently uses 1 bit per entry)

Concepts of Object Oriented Programming

Dr. Axel Kohlmeyer

Senior Scientific Computing Expert

Information and Telecommunication Section
The Abdus Salam International Centre
for Theoretical Physics

http://sites.google.com/site/akohlmey/

akohlmey@ictp.it

The Abdus Salam International Centre for Theoretical Physics

Workshop on Computer Programming and Advanced Tools for Scientific Research Work