Scientific Computing (Phys 2109/Ast 3100H)
I. Scientfic Software Development

SciNet HPC Consortium

University of Toronto

Winter 2013

Part I

Introduction to Software Development

Lecture 5

Modular programming (summary)

Modular programming summary

- Scientific software can be large, complex, subtle
- Should enforce boundaries between sections of code by making self-contained modules of functionality.
- Make each module separately testable.
- Keep purpose of module clear.
- Think about everything you might want to use these routines for; *then* design the interface.
- But also keep it as simple as possible.
- Implementation separate from interface.

Object-Oriented Programming

Limits to (Just) Structured Programming

Stuctured programming is the methodology of breaking a given programming task up into smaller bits using subroutines and loops, to be decomposed into yet smaller bits, ..., until one has simple tasks to implement.

Good approach to solving one particular programming task.

Problems

- What if multiple actions are to be performed on data?
- Code needs to know about the data structure.
- Leads to reinventing the wheel.

One would instead like to build "components" with known properties and known ways to plug them into your program.

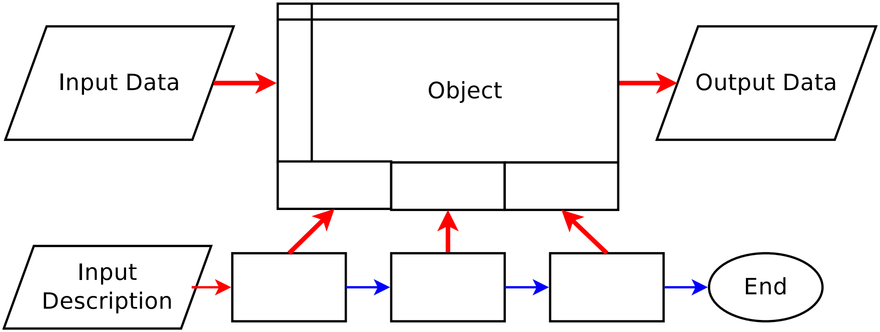
Limits to (Just) Modular Programming

- Modular programming is a good way of improving code reusablility.
- Also good for hiding implementation data.
- But still code-centric.
- Code needs to know about the details of the data structure.
- When a data structure has to change a bit, e.g., the functionality of a module needs to be extended, potentially have change all the code and the interface of the module.

Object-Oriented Programming

Definition

Object-oriented programming treats data and the procedures that act on them as single "objects".



Object-Oriented Programming

- Complexity can be hidden inside each component.
- Can separate interface from the implementation.
- Allows a clear separation of tasks in a program.
- Reuse of components.
- Same interface can be implemented by different objects.
- Objects' functionality can be extended using inheritence.

Warning: Be sure you know:

- the computational cost of the operations
- what temporary objects may be created.

At a low level, OOP may need to be broken for best performance.

C objects: Structs

```
// Create a new type
struct Point2D {
   double x,y;
   int i;
};
void print1(struct Point2D a) {
   std::cout << a.i << ' ' << a.x << ' ' << a.y;
}
void print2(struct Point2D* a) {
   std::cout << a->i << ' ' << a->x << ' ' << a->y;
}
```

- Calling `print1` make a copy of the content of `a`.
- Calling `print2` only copies address of `a`.
- Memory copies are not cheap!
- If we do this with classes, the *constructor* is called every time `print1` is called.

C objects: Structs

```
// Create a new type
struct Point2D {
   double x,y;
   int i;
};
void print1(Point2D a) {
   std::cout << a.i << ' ' << a.x << ' ' << a.y;
}
void print2(Point2D* a) {
   std::cout << a->i << ' ' << a->x << ' ' << a->y;
}
```

- Calling `print1` make a copy of the content of `a`.
- Calling `print2` only copies address of `a`.
- Memory copies are not cheap!
- If we do this with classes, the *constructor* is called every time `print1` is called.

C objects: Structs

```
// Create a new type
struct Point2D {
   double x,y;
   int i;
};
void print1(Point2D a) {
   std::cout << a.i << ' ' << a.x << ' ' << a.y;
}
void print2(Point2D& a) {
   std::cout << a.i << ' ' << a.x << ' ' << a.y;
}
```

- Calling `print1` make a copy of the content of `a`.
- Calling `print2` only copies address of `a`.
- Memory copies are not cheap!
- If we do this with classes, the *constructor* is called every time `print1` is called.

Encapsulation in Objects

- In true OOP, data is encapsulated and accessed using methods specific for that (kind of) data.
- The interface (collection of methods) should be designed around the meaning of the actions: abstraction.
- Programs typically contain multiple objects of the same type, called instances.

Polymorphism and Inheritence

- Programs typically contain different types of objects.
- Types of objects can be related, and their methods may act in the same ways, such that the same code can act on different types of object, without knowing the type: polymorphism.
- Types of object may build upon other types through inheritance.

Classes and Objects

- Objects in C++ are made using 'classes'.
- A class is a type of object.
- From a class, one creates 1 or more instances.
- These are the objects.
- Syntactically, classes are structs with member functions.

How do we add these member functions?

```
class classname // new keyword 'class'
{
   public:
      type1 name1;
      type2 name2;
      type3 name3(arguments); // function
      ...
};
```

- `public` allows use of members from outside the class.

Example

```
class Point2D {
   public:
      int j;
      double x,y;
      void set(int aj,double ax,double ay);
};
```

How do we define these member functions?

- Use the scope/namespace operator `::`

```
type3 classname::name3(arguments) {
   statements
}
```

Example

```
void Point2D::set(int aj,double ax,double ay) {
   j = aj;
   x = ax;
   y = ay;
}
```

Classes: How do we use the class?

Definition

```
classname objectname;
classname* ptrname = new classname;
```

Access operator . and ->

```
objectname.name             // variable access
objectname.name(arguments);// member function
ptrname->name               // variable access
ptrname->name(arguments);   // member function
```

Example

```
Point2D myobject;
myobject.set(1,-0.5,3.14);
std::cout << myobject.j << std::endl;
```

Data hiding

- Good components hide implementation details
- Each member function or data member can be
  1. `private:` only member functions of class have access
  2. `public:` accessible from anywhere
  3. `protected:` accessible only to this and derived classes.
- These are specified as sections within the class.

Example (Declaration)

```
class Point2D {
   private:
      int j;
      double x,y;
   public:
      void set(int aj,double ax,double ay);
      int get_j();
      double get_x();
      double get_y();
};
```

Data hiding

Example (Definition)

```
int Point2D::get_j() {
   return j;
}
double Point2D::get_x() {
   return x;
}
double Point2D::get_y() {
   return y;
}
```

Example (Usage)

```
Point2D myobject;
myobject.set(1,-0.5,3.14);
std::cout << myobject.get_j() << std::endl;
```

Data hiding

WARNING:

When hiding the data through these kinds on accessor functions, now, each time the data is needed, a function has to be called, and there's an overhead associate with that.

- The overhead of calling this function can sometimes be optimized away by the compiler, but often it cannot.
- Considering making data is that is needed often by an algorithm just `public,` or use a `friend` .

Class > Struct

- A class defines a type, and when an instance of that type is declared, memory is allocated for that struct.
- A class is more than just a chunk of memory.
  For example, arrays may have to be allocated (`new` ...) when the object is created.
- When the object ceases to exist, some clean-up may be required (`delete` ...).
- Constructor
  ...is called when an object is created.
- Destructor
  ...is called when an object is destroyed.

Constructors

- Declare constructors in the class with no return type:

```
class classname{
     ...
  public:
     classname(arguments);
     ...
}
```

- Define them in the usual way,

```
classname::classname(arguments) {
  statements
}
```

- Use them by defining an object or with new.

```
classname object(arguments);
classname* object = new classname(arguments);
```

- You usually want a constructor without arguments too.

Constructors

Example

```
class Point2D {
   private:
      int j;
      double x,y;
   public:
      Point2D(int aj,double ax,double ay);
      int get_j();
      double get_x();
      double get_y();
};
Point2D::Point2D(int aj,double ax,double ay) {
   j = aj;
   x = ax;
   y = ay;
}
Point2D myobject(1,-0.5,3.14);
```

Destructors

- . . . are called when an object is destroyed.
- This occurs when a non-static object goes out-of-scope, or when `delete` is used.
- Good opportunity to release memory.

Example

```
classname* object = new classname(arguments);
...
delete object;// object deleted:  calls destructor
```

```
{
   classname object;
}// object goes out of scope:  calls destructor
```

Destructors

- Declare destructor as a member functions of the class with no return type, with a name which is the class name plus a ~ attached to the left.

```
class classname{
     ...
  public:
     ~classname();
     ...
}
```

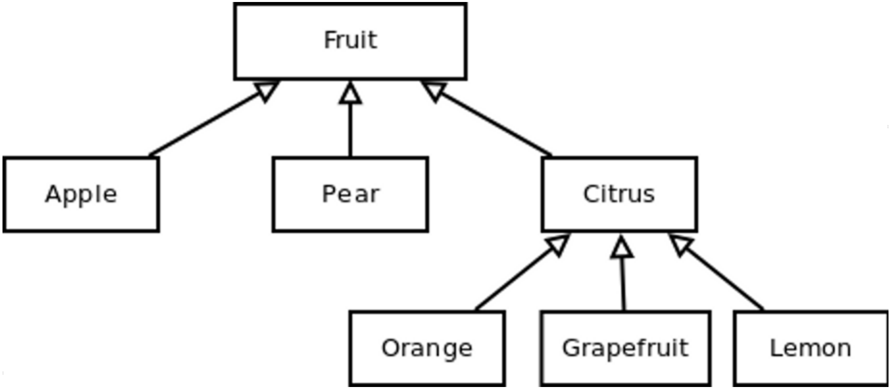
- Define a destructor as follows:

```
classname::~classname() {
  statements
}
```

- A destructor cannot have arguments.

OOP: Inheritance (Derived Classes)

Example (abstract object hierarchy)



Inheritance

- child classes are derived from other parent classes
- automatically include parent's members
- inherit all the accessible members of the base class

Inheritance

Base Class

```
class baseclass {
   protected:
   ...
   public:
      baseclass ()
      ...
};
```

Derived Class

```
class derivedclass : public baseclass {
   ...
   public:
      derivedclass : baseclass ()
      ...
};
```

Inheritance

Example (Matrix Base Class)

```
class matrix {
   protected:
      int rows, cols;
      double *elements;
   public:
      matrix(int r, int c);
      ~matrix();
      int get_rows();
      int get_cols();
      void fill(double value);
};
```

Inheritance

Example (Square Matrix Derived Class)

```
class squarematrix :  public matrix {
   public:
      squarematrix(int r, int c) :  matrix(r,c) {
         if(r!=c) std::cerr<<"not a square matrix";
         exit(1);
      }
      double trace() {
         double sum(0.0);
         for(int i=0; i <rows ; i++)
         sum += elements[i*cols+i];
         return sum;
      }
};
```

Inheritance

Example

```
matrix P(5,5);
squarematrix Q(5,5);
P.fill(1.6);
Q.fill(1.6);
std::cout<<" Trace = "<<Q.trace();
```

Polymorphism

- Objects that adhere to a standard set of properties and behaviors can be used interchangeably.
- Implemented by Overloading and Overriding

Why bother?

- Avoid code duplication/reuse where not necessary
- Simplifies and structures code
- Common interface
- Consistency of design should be more understandable
- Debugging

Polymorphism in Inheritance

Idea

- Use base class as framework for derived classes usage.
- Define member functions with `virtual` keyword.
- Override base class functions with new implementations in derived classes.
- If `virtual` keyword not used, overloading won't occur.

Polymorphism comes from the fact that you could call the based method of an object belonging to any class that derived from it, without knowing which class the object belonged to.

Inheritance

Example (Matrix Base Class)

```
class matrix {
   protected:
      int rows, cols;
      double *elements;
   public:
      matrix(int r, int c);
      ~matrix();
      int get_rows();
      int get_cols();
      virtual void fill(double value);
};
```

Inheritance

Example (Square Matrix Derived Class)

```
class squarematrix :  public matrix {
   private:
   protected:
   public:
      squarematrix(int r, int c) :  matrix(r,c) {
         if(r!=c) std::cerr<<"not a square matrix";
         exit(1);
      }
      double trace();
      void fill(double value) {
         for (int i=0; i < rows*cols; i++)
            elements[i] = value;
      }
};
```

Inheritance

Example (non-virtual)

```
squarematrix Q(5,5);
Q.fill(1.6);
std::cout<<" Trace = "<<Q.trace();
```

Example (virtual)

```
matrix *Q;
Q = new squarematrix(5,5);
Q->fill(1.6);
std::cout<<" Trace = "<<Q->trace();
```

Gotcha:

- Virtual functions are run-time determined
- Equivalent cost to a pointer dereference
- Not as efficient as compile time determined (ie non-virtual)
- Should be avoided for small functions that are called often