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

SciNet HPC Consortium

University of Toronto

November 2011

Part IV

Object-oriented programming (C++/Python)

Limits to structured programming

Problems

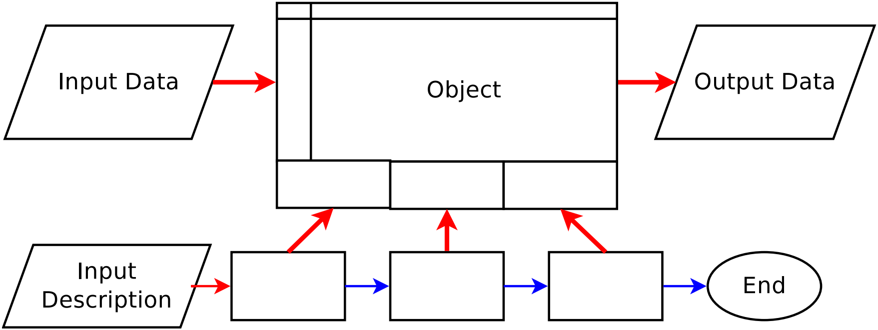
- Complex input data
- Multiple actions to be performed on data
- Separation data+code is bad for reusability
- Leads to reinventing the wheel

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

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.
- Helps maintanance.

Gotcha: Mind The Cost!

- know the computational cost of the operations
- know what temporary objects may be created,
- and know how much creating different types of object costs.

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

Object oriented languages

There are many. Just to pick two:

C++

- was designed for object oriented and generic programming,
- has better memory management, stricter type checking, and easier creation of new types than C,
- while you can still optimize at a low level when needed.

Python

- also supports object oriented programming
- much more explicit; no true encapsulation
- not suitable for number crunching, but you can use Python-C interfaces

C++

First some nice features,

then the heart of the matter

Namespaces

- In larger projects, name clashes can occur.
- No more: put all functions, structs, ... in a namespace:

```
namespace nsname {
   ...
}
```

- Effectively prefixes all of `...` with `nsname::`
- Many standard functions/classes are in namespace `std`.
- To omit the prefix, do "`using namespace nsname;`"
- Can selectively omit prefix, e.g., "`using std::vector`"

I/O streams

Standard input/error/output

- Streams objects handle input and output.
- All in namespace **std**.
- Global stream objects (header: **<iostream>**)
  - **cout** is for standard output (screen)
  - **cout** is the standard error output (screen)
  - **cin** is the standard input (keyboard)
- Use insertion operator **<<** for output:

```
std::cout << "Output to screen!" << std::endl;
```

(**endl** ends the line and flushes buffer)

- Use extraction operator **>>** for input:

```
std::cin >> variable;
```

- These operators figure out type of data and format.

I/O streams

File stream objects (header: `<fstream>`)

- **`ofstream`** is for output to file.
  Declare with filename: good to go!

```
std::ofstream file("name.txt");
file << "Writing to file";
```

- **`ifstream`** is for input from a file.
  Declare with filename: good to go!

```
std::ifstream file("name.txt");
int i;
file >> i;
```

- Can also open and close by hand.

I/O streams

Example

C:

```
double a,b,c;
FILE* f;
scanf(f, "%lf %lf %lf", &a, &b, &c);
f = fopen("name.txt","w");
fprintf(f, "%lf %lf %lf\n", a, b, c);
fclose(f);
```

C++:

```
using namespace std;
double a,b,c;
cin >> a >> b >> c;
ofstream f("name.txt");
f << a << b << c << endl;
```

References

- A reference gives another name to an existing object.
- References are similar to pointers.
- Do not use pointer dereferencing (`->`), but a period `.`

Standalone definition (rare)

```
type & name = object;
```

- *object* has to be of type *type*.
- *name* is a reference to *object*.
- *name* points to *object*, i.e., changing *name* changes *object*.
- Members accessed as *name.membername*.

Definition as arguments of a function

```
returntype functionname(type & name, ...);
```

SciNet compute • calcul CANADA

References

Example

To change a function argument, need a pointer in C:

```
void makefive(int * a) {
   *a = 5;
} ...
int b = 4;
makefive(&b); /* b now holds 5 */
```

C++: can pass by reference using &:

```
void makefive(int & a){
   a = 5;
} ...
int b = 4;
makefive(b); /* b now holds 5 */
```

SciNet compute • calcul CANADA

Using references to avoid copies

Compare these two functions

```
struct Point3D {
   double x,y,z;
};
void print1(Point3D a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z <<
   std::endl;
}
void print2(Point3D& a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z <<
   std::endl;
}
```

- Calling **print1** copies the content of **a** to the stack (24 bytes).
- Calling **print2** only copies address of **a** to the stack (8 bytes).
- Memory copies are not cheap!
- If we do this with classes, the *constructor* is called everytime **print1** is called, whereas **print2** still only copies 8 bytes.

Improved memory allocation

Basic allocation

```
type* name = new type;
```

Allocation with initialization

```
type* name = new type(arguments);
```

Array allocation

```
type* name = new type[arraysize];
```

Basic de-allocation

```
delete name;
```

Array de-allocation

```
delete [] name;
```

Improved memory allocation

Example

```
struct credit {
   long number, balance;
};
```

No more of this mess:

```
#include "stdlib.h"
struct credit* a;
double * b;
a = (struct credit*)malloc(sizeof(struct credit));
b = (double *)malloc(sizeof(double )*10000);
...
free(a); free(b);
```

Instead, simply:

```
credit* a = new credit;
double * b = new double [10000];
...
delete a; delete [] b;
```

Exception handling

The c++ way to handle errors is not to return an error code but to throw an exception.

Example

```
class Error {...}
float mylog(float x) {
    if (x<=0) throw Error();
    ...
} ...
try {
    y = mylog(x);
} catch (Error&e) {
    ...
}
```

- No need to change function signature to add error checking
- Can catch at a higher level
- Uncaught exception exit the application

STL (Standard Template Library)

Offers a lot of basic functionality

- Supplies a lot of data types and containers (templated).
- Often presented as part and parcel of the C++ language itself.
- Also contains a number of algorithms for e.g., sorting, finding
- Efficiency implementation dependent, and generally not great.

Some of the STL data types

`vector`	Relocating, expandable array
`list`	Doubly linked list
`deque`	Like vector, but easy to put something at beginning
`map`	Associates keys with elements
`set`	Only keys
`stack`	LIFO
`queue`	FIFO
`...`	

SciNet compute • calcul CANADA

Object Oriented Programming

Encapsulation in objects

- 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.

Object oriented

- 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

What are 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.

Classes: 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);
};
```

Classes: How do we define these member functions?

The scope 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 access
ptrname->name                   // variable access
ptrname->name(arguments);       // member function access
```

Example

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

SciNet compute • calcul CANADA

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:** only this class and its derived classes have access.
- 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

Gotcha:

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 as member functions of 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 as well.

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

Destructor

... is called when an object is destroyed.

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();
      ...
}
```

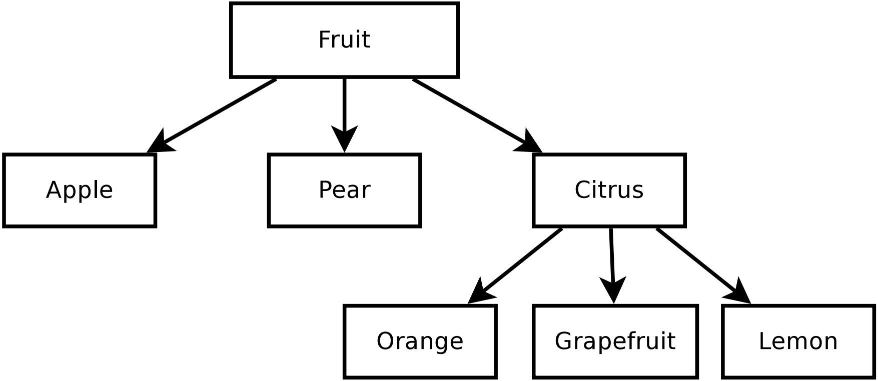
Define a destructor as follows:

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

- A destructor cannot have arguments.

OOP: Inheritance (Derived Classes)

Example (abstract object hierarchy)



Inheritance

Definition

- 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);
};
```

SciNet compute • calcul CANADA

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.

SciNet compute • calcul CANADA

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

Debugging, Python. . .

HW4

Rewrite your coupled tracer particle/diffusion application to C++. In the process of refactoring, you will make sure both the Tracer and Diffusion classes derive from the common class:

```
#ifndef _DYNAMICSYSTEMH_
#define _DYNAMICSYSTEMH_
class Parameters;
class Diffusion;
class Tracer;
class DynamicSystem {
   public:
   virtual void parameters(Parameters& p)  virtual void
   print()
   virtual void compute()
   virtual void evolve()
   virtual void couple(Diffusion&d)
   virtual void couple(Tracer&t)
   virtual  DynamicSystem()
};
#endif
```

HW4

The following code should work:

```
#include "parameters.h"
#include "diffusion.h"
#include "tracer.h"
int main(){
   Parameters param;
   param.read();
   Diffusion part1;
   Tracer part2;
   part1.parameters(param);
   part2.parameters(param);
   part1.print();
   part2.print();
   for (int step = 0; step < param.nsteps; step++) {
      part1.couple(part2); part2.couple(part1);
      part1.compute(); part2.compute();
      part1.evolve(); part2.evolve();
      part1.print(); part2.print();
   } return 0;
}
```