Siena Computational Crystallography School 2005

Modern approaches to programming

Ralf W. Grosse-Kunstleve

Computational Crystallography Initiative
Lawrence Berkeley National Laboratory

Disclosure

- Experience
  - Basic
  - 6502 machine language
  - Pascal
  - Fortran 77
  - csh, sh
  - C
  - Perl
  - Python
  - C++
- Last five years
  - Python & C++ -> cctbx, phenix
- Development focus
  - phenix.refine, phenix.hyss
- No experience
  - TCL/TK
  - Java

Computational Crystallography Toolbox

- Open-source component of phenix
  - Automation of macromolecular crystallography
- mmtbx – macromolecular toolbox
- cctbx – general crystallography
- scitbx – general scientific computing
- libtbx – self-contained cross-platform build system
- SCons – make replacement
- Python scripting layer (written in C)
- Boost C++ libraries
- Exactly two external dependencies:
  - OS & C/C++ compiler

Object-oriented programming

The whole is more than the sum of its parts.

Syntax is secondary.

Purpose of modern concepts

- Consider
  - You could write everything yourself
  - You could write everything in machine language
- Design of Modern Languages
  - Support large-scale projects <-> Support collaboration
  - Maximize code reuse <-> Minimize redundancy
  - Software miracle: improves the more it is shared

Main concepts behind modern languages

- Namespaces
- A special namespace: class
- Polymorphism
- Automatic memory management
- Exception handling
- Concurrent development
  - Developer communication
- Secondary details
  - friend, public, protected, private

Evolution of programming languages

Namespaces

- **Emulation**

  `MtzSomething` (CCP4 CMTZ library)
  http://www.ccp4.ac.uk/dist/html/C_library/cmtzlib_8h.html
  `QSomething` (Qt GUI toolkit)
  http://doc.trolltech.com/4.0/classes.html
  `PySomething` (Python)
  http://docs.python.org/api/genindex.html
  `glSomething` (OpenGL library)
  http://www.rush3d.com/reference/opengl-bluebook-1.0/
  `A00, A01, C02, C05, C06` (NAG library)
  http://www.nag.co.uk/numeric/fl/manual/html/FLlibrarymanual.asp
- **Advantages**
  - Does not require support from the language
- **Disadvantages**
  - Have to write `XXXSomething` all the time
  - Nesting is impractical

Evolution of programming languages

Namespaces

- **Formalization**

  similar to:

  transition from flat file systems to files and directories

```
namespace MTZ {
  Something
}
```

- **Disadvantages**
  - Does require support from the language
- **Advantages**
  - Inside a namespace it is sufficient to write `Something`
    - as opposed to `XXXSomething`
  - Nesting "just works"
    - If you know how to work with a directories you know how to work with namespaces

Evolution of programming languages

A special namespace: class

- **Emulation**
  - COMMON block with associated functions

```
double precision a, b, c, alpha, beta, gamma
COMMON /unit_cell/ a, b, c, alpha, beta, gamma
subroutine ucinit(a, b, c, alpha, beta, gamma)
double precision function ucvol()
double precision function stol(h, k, l)
```

- **Disadvantage**
  - The associations are implicit
    - difficult for others to see the connections

Evolution of programming languages

A special namespace: class

- **Formalization**

```
class unit_cell:
  def __init__(self, a, b, c, alpha, beta, gamma)
  def vol(self)
  def stol(self, h, k, l)
```

- **What's in the name?**
  - class, struct, type, user-defined type
- **Advantage**
  - The associations are explicit
    - easier for others to see the connections

Evolution of programming languages

A special namespace: class

- **Formalization**

```
class unit_cell:
  def __init__(self, a, b, c, alpha, beta, gamma)
  def vol(self)
  def stol(self, h, k, l)
```

- **What's in the name?**
  - class, struct, type, user-defined type
- **Advantage**
  - The associations are explicit
    - easier for others to see the connections

Evolution of programming languages

A namespace with life-time: self, this

- COMMON block = only **one** instance
- class = blueprint for creating arbitrarily **many** instances
- Example

```
hex = unit_cell(10, 10, 15, 90, 90, 120)
rho = unit_cell(7.64, 7.64, 7.64, 81.79, 81.79, 81.79)
```

- `hex` is one *instance*, `rho` another of the same class
- Inside the class definition `hex` and `rho` are both called `self`
- What's in the name?
  - self, this, instance, object
- `hex` and `rho` live at the same time
- the memory for `hex` and `rho` is allocated when the object is *constructed*

Life time: a true story

A true story about my cars, told in the Python language:

```
class car:
  def __init__(self, name, color, year):
    self.name = name
    self.color = color
    self.year = year

car1 = car(name="Toby", color="gold", year=1988)
car2 = car(name="Emma", color="blue", year=1986)
car3 = car(name="Jamson", color="gray", year=1990)
del car1 # donated to charity
del car2 # it was stolen!
car4 = car(name="Jessica", color="red", year=1995)
```

Alternative view of class

- Function returning only one value

```
real function stol(x)
...
s = stol(x)
```

- Function returning multiple values

```
class wilson_scaling:
  def __init__(self, f_obs):
    self.k = ...
    self.b = ...
wilson = wilson_scaling(f_obs)
print wilson.k
print wilson.b
```

- Class is a generalization of a function

Evolution of programming languages

A special namespace: class

- Summary
  - A class is a namespace
  - A class is a blueprint for object construction and deletion
  - In the blueprint the object is called self or this
  - Outside the object is just another variable
- When to use classes?
  - Only for “big things”?
  - Is it expensive?
- Advice
  - If you think about a group of data as one entity -> use a class to formalize the grouping
  - If you have an algorithm with 2 or more result values -> implement as class

Evolution of programming languages

Polymorphism

- The same source code works for different types
- Runtime polymorphism
  - “Default” in dynamically typed languages (scripting languages)
  - Very complex in statically typed languages (C++)
- Compile-time polymorphism
  - C++ templates

Evolution of programming languages

Compile-time polymorphism

- Emulation
  - General idea

```
S      subroutine seigensystem(matrix, values, vectors)
D      subroutine deigensystem(matrix, values, vectors)
S      real             matrix(...)
D      double precision matrix(...)
S      real             values(...)
D      double precision values(...)
S      real             vectors(...)
D      double precision vectors(...)
```

    Use grep or some other command to generate the single and double precision versions

  - Real example
    - http://www.netlib.org/lapack/individualroutines.html

Evolution of programming languages

Compile-time polymorphism

- Formalization

```
template <typename FloatType>
class eigensystem
{
  eigensystem(FloatType* matrix)
  {
    // ...
  }
};

eigensystem<float> es(matrix);

eigensystem<double> es(matrix);
```

- The C++ template machinery automatically generates the type-specific code as needed

Automatic memory management

- Context
  - Fortran: no dynamic memory management
    - Common symptom
      - Please increase MAXA and recompile
  - C: manual dynamic memory management via malloc & free
    - Common symptoms
      - Memory leaks
      - Segmentation faults
      - Buffer overruns (vector for virus attacks)
      - Industry for debugging tools (e.g. purify)

Automatic memory management

- Emulation: Axel Brunger's ingenious approach
  - Insight: stack does automatic memory management!

```
subroutine action(args)
  allocate resources
  call action2(args, resources)
  deallocate resources

subroutine action2(args, resources)
  do work
```

  - Disadvantage
    - Cumbersome (boiler plate)

Automatic memory management

- Formalization
  - Combination
    - Formalization of object construction and deletion (class)
    - Polymorphism
  - Result = fully automatic memory management
  - "Default" in scripting languages
    - garbage collection, reference counting
  - C++ Standard Template Library (STL) container types
    - std::vector<T>
    - std::set<T>
    - std::list<T>
- Advice
  - Use the STL container types
  - Never use new and delete
    - Except in combination with smart pointers
      - std::auto_ptr<T>, boost::shared_ptr<T>

Evolution of programming languages

Exception handling

- Emulation

```
subroutine matrix_inversion(a, ierr)
...
matrix_inversion(a, ierr)
if (ierr .ne. 0) stop 'matrix not invertible'
```

- Disadvantage
  - `ierr` has to be propagated and checked throughout the call hierarchy -> serious clutter
  - to side-step the clutter: `stop`
    - not suitable as library

Emulation of exception handling

```
program top
call high_level(args, ierr)
if (ierr .ne. 0) then
  write(6, *) 'there was an error', ierr
endif
end
subroutine high_level(args, ierr)
call medium_level(args, ierr)
if (ierr .ne. 0) return
do something useful
end
subroutine medium_level(args, ierr)
call low_level(args, ierr)
if (ierr .ne. 0) return
do something useful
end
subroutine low_level(args, ierr)
if (args are not good) then
  ierr = 1
  return
endif
do something useful
end
```

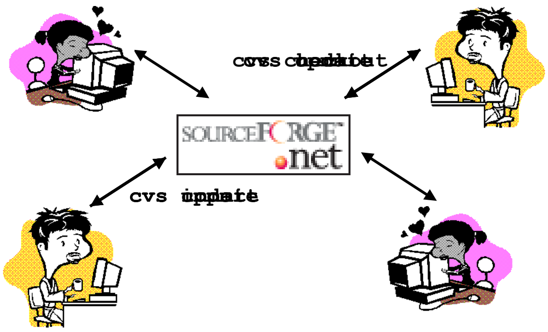
Evolution of programming languages

Exception handling

- Formalization

```
def top():
  try:
    high_level(args)
  except RuntimeError, details:
    print details
def high_level(args):
  medium_level(args)
  # do something useful
def medium_level(args):
  low_level(args)
  # do something useful
def low_level(args):
  if (args are not good):
    raise RuntimeError("useful error message")
  # do something useful
```

Collaboration via SourceForge



Conclusion concepts

- Advantages
  - Modern languages are the result of an evolution
    - Superset of more traditional languages
    - A real programmer can write Fortran in any language
  - Designed to support large collaborative development
    - However, once the concepts are familiar even small projects are easier
  - Solve common problems of the past
    - memory leaks
    - error propagation from deep call hierarchies
  - Designed to reduce redundancy (boiler plate)
  - If the modern facilities are used carefully the boundary between "code" and documentation begins to blur
    - Especially if runtime introspection is used as a learning tool
  - Readily available and mature
    - C and C++ compilers are at least as accessible as Fortran compilers
  - Rapidly growing body of object-oriented libraries

Conclusion concepts

- Disadvantages
  - It can be difficult to predict runtime behavior
    - Tempting to use high-level constructs as black boxes
  - You have to absorb the concepts
    - syntax is secondary!
  - However: Python is a fantastic learning tool that embodies all concepts outlined in this talk
    - except for compile-time polymorphism

Acknowledgements

- Organizers of this meeting
- Paul Adams
- Pavel Afonine
- Peter Zwart
- Nick Sauter
- Nigel Moriarty
- Erik McKee
- Kevin Cowtan
- David Abrahams
- Open source community

http://www.phenix-online.org/ http://cctbx.sourceforge.net/