Lecture 2
A brief overview of simple Python and more advanced C++

Methods in Medical Image Analysis - Spring 2017
16-725 (CMU Ri) : BioE 2630 (Pitt)
Dr. John Galeotti
Based in part on Damion Shelton's slides from 2006

Recap

Today's lecture is online
- I will usually place lectures online before 4 AM the day of the class.

Goals for this lecture

• C++ vs. Python
• Brief Python Introduction
• Overview of object oriented programming
  • Inheritance & polymorphism
  • Public/private/protected derivation
• Overview of generic programming
  • templates
  • templated classes
  • specialization
• typedef & typename keywords

Disclaimer

• Some of you will definitely know more about Python than I do.
• Some of you may know more about object oriented programming than what I will present (or what I remember)
• We will not discuss the more esoteric inheritance methods, such as friend classes

Reference & Review Material

• Books
  • C++ How to Program - Deitel & Deitel
  • Teach Yourself C++ In 21 Days - Liberty
  • Using the STL: The C++ Standard Template Library - Robison
  • Design Patterns; Elements of Reusable Object-Oriented Software - Gamma et al.
• Websites
  • http://docs.python.org/tutorial/
  • http://docs.python.org/reference/index.html
  • http://www.cppreference.com/
  • I use this one more than the rest.
  • http://www.cplusplus.com/doc/tutorial/

C++ vs. Python

• C++
  • Compile and Link
  • Low-level language (but standardized higher-level libraries available)
  • Writing code takes longer
  • Code runs very fast
• Python
  • Interpreted
  • Very high level language
  • Writing code is quick and easy
  • Python code runs more slowly, but...
  • Python can call precompiled C/C++ Libraries
• Best of both worlds
• So ITK could should execute at full compiled speed, even when called from Python.
In general, I will try to format code in a fixed-width font as follows:

```plaintext
this->IsSome(code);
```

However, not all code that I present could actually be executed (the above, for instance)

---

### Python Example Code
(Take notes as needed!)

```python
# Everything on a line after a # is a comment
# Warning: Indentation matters in Python!
import SimpleITK as sitk
input = sitk.ReadImage( "images/cthead1.jpg" )
output = sitk.SmoothingRecursiveGaussian ( input , 2.0 )
sitk.Show( output )

image = sitk.ReadImage( "images/cthead1.jpg" )
output = sitk.SmoothingRecursiveGaussian ( input , 2.0 )
sitk.Show( output )

image = sitk.Image( 256,256, sitk.sitkFloat32 )
image[160,160] = 99.9   # [] allows direct pixel access
sitk.Show( sitk.Add( output, image) )
```

---

### List of SimpleITK Pixel Types

- The definitive list of SimpleITK pixel types is in its source code
- SimpleITK's source code must be downloaded separately
- Look at the bottom of this file:
  ```
  #include "sitkPixelIDValues.h"
  ```
- Warning: Not every compilation of SimpleITK supports all of these pixel types.
- The source code has recommendations for how to check that a given type is available, etc.

---

### Object-oriented programming

- Identify functional units in your design
- Write classes to implement these functional units
  - Preferably as “black boxes”
- Separate functionality as much as possible to promote code re-use

---

### Class membership

- Classes have member variables and methods
- ITK names class member variables with the “m_” prefix, as in “m_VariableName”
- Class members are 1 of 3 types
  - Public
  - Private
  - Protected
Public membership

- Everyone can access the member
  - The rest of the world
  - The class itself
  - Child classes
- You should avoid making member variables public, in order to prevent undesired modification.
  - A black box shouldn’t have openings!

Private membership

- Only the class itself can access the member
- It’s not visible to the rest of the world
- Child classes can’t access it either

Protected membership

- The middle ground between public and private
- The outside world can’t access it... but derived classes can

ITK and membership

- In ITK, member variables are almost always private
- There are public accessor functions that allow the rest of the world to get and set the value of the private member
- This ensures that the class knows when the value of a variable changes

Why do it this way?

- Consider a filter class—if someone changes a variable in the filter, it should re-run itself the next time the user asks for output
- If nothing has changed, it doesn’t waste time running again
- Accessor functions set a “modified flag” to notify the framework when things have changed
- More on this in another lecture

Inheritance in a nutshell

- Pull common functionality into a base class
- Implement specific/unique functionality in derived classes
- Don’t re-invent the wheel!
- Base classes = parents
- Derived classes = children
Overloading

- If a child class re-implements a function from the base class, it "overloads" the function.
- You can use this to change the behavior of a function in the child class, while preserving the global interface.

An example of inheritance in a graphical drawing program

Shape
  - Polygon
  - Triangle
  - Quadrilateral
  - Rectangle
  - Trapezoid
  - Rhombus
  - Pentagon
ConicSection
  - Ellipse
  - Circle
  - Parabola

An example of ITK inheritance

```cpp
#include <itk::DataObject>
#include <itk::ImageBase<typename ImageDimension>
#include <itk::Image<TPixel, typename ImageDimension> >
```

C++ Namespaces

- Namespaces solve the problem of classes that have the same name.
- E.g., ITK contains an Array class, perhaps your favorite add-on toolkit does too.
- You can avoid conflicts by creating your own namespace around code.
  ```cpp
  namespace itk {
    code
  }
  ```

C++ Namespaces, cont.

- Within a given namespace, you refer to other classes in the same namespace by their name only, e.g. inside the itk namespace Array means "use the ITK array".
- Outside of the namespace, you use the itk:: prefix, e.g. itk::Array.
- Only code which is part of ITK itself should be inside the itk namespace.
- At minimum, you’re always in the global namespace.

C++ Namespaces, cont.

- Note that code within the itk namespace should refer to code outside of the namespace explicitly.
- E.g. use `std::cout` instead of `cout`.

```cpp
```
C++ Virtual functions

• Want to enforce a consistent interface across a set of child classes?
• Virtual functions allow a base class to declare functions that “might” or “must” be in its child classes
• The “=0” declaration means that the function must be implemented in a child class
  ▪ Because it is not implemented in the base class
• Virtual functions that are implemented in the base class can still be overridden by child classes

C++ Virtual functions, cont.

• You can specify (and use) a virtual function without knowing how it will be implemented in child classes
• This allows for polymorphism
• For example:
  ```cpp
  virtual void DrawSelf() = 0;
  ```

C++ Example of polymorphism in a graphical drawing program

Shape: DrawSelf() = 0;
Polygon: int vertices; DrawSelf() connects vertices with line segments
  ▪ Triangle: vertices=3
  ▪ Quadrilateral: vertices=4
Rectangle
Rhombus
Pentagon: vertices=5
ConicSection
  ▪ Ellipse: DrawSelf() uses semimajor and semiminor axes
  ▪ Circle: fonas length semimajor axis = length semimajor
  ▪ Parabola

Generic programming

• Generic programming encourages:
  ▪ Writing code without reference to a specific data type (float, int, etc.)
  ▪ Designing code in the most “abstract” manner possible
• Why?
  ▪ Trades a little extra design time for greatly improved re-usability

Image example

• Images are usually stored as arrays of a particular data type
  ▪ e.g. `unsigned char[256*256]`
• It’s convenient to wrap this array inside an image class (good object oriented design)
• Allowing the user to change the image size is easy with dynamically allocated arrays

Image example, cont.

• Unfortunately, changing the data type is not so easy
• Typically you make a design choice and live with it (most common)
• Or, you’re forced to implement a double class, a float class, an int class, and so on (less common, can be complicated)
  ▪ This is the interface used by SimpleITK, but…
  ▪ SimpleITK usually automates type selection to make your life easier
Templates to the rescue

- Templates provide a way out of the data type quandary
- ITK uses templates extensively
- SimpleITK relies on ITK, and SimpleITK’s automated type functionality depends on ITK’s templated nature
- If you’re familiar with macros, you can think of templates as macros on steroids
- With templates, you design classes to handle an arbitrary “type”

Anatomy of a templated class

```
template <class TPixel, unsigned int VImageDimension=2>
    class ITK_EXPORT Image : public ImageBase<VImageDimension>
```

TPixel is a class (of some sort)

VImageDimension is an unsigned int, with a default value of 2

```
template <class TPixel, unsigned int VImageDimension=2>
    class ITK_EXPORT Image : public ImageBase<VImageDimension>
```

Image is the name of this class

Image is derived from ImageBase in a public manner
Specialization

- When you specify all of the template parameters, you “fully specialize” the class
- In the previous example, `ImageBase<VImageDimension>` specializes the base class by specifying its template parameter.
- Note that the `VImageDimension` parameter is actually “passed through” from `Image`’s template parameters.

Derivation from templated classes

- You must specify all template parameters of the base class
- The template parameters of the base class may or may not be linked to template parameters of the derived class
- You can derive a non-templated class from a templated one if you want to (by hard coding all of the template parameters)

Partial specialization

- C++ also allows partial specialization
- For example, you write an Image class that must be 3D, but still templates the pixel type (or vice-versa)
- Starting with v4, ITK uses partial specialization
- All modern compilers support it
  - But Visual Studio 6 does not

Templated class instances

- To create an instance of a templated class, you must fully specialize it
- E.g.
  ```cpp
  itk::Image<int, 3> myImage;
  ```
  Creates a 3D image of integers
  (not quite true, but we can pretend it does until we cover smart pointers)

Typedefs

- One consequence of templates is that the names of a fully defined type may be quite long
- E.g., this might be a legal type:
  ```cpp
  itk::Image<itk::MyObject<3, double>, 3>
  ```

Typedefs cont.

- You can create a short-hand name for our user-defined type by using the typedef keyword
  ```cpp
  typedef itk::Image<int, 3> 3DIntImageType;
  3DIntImageType myImage;
  3DIntImageType anotherImage;
  ```
Fun with typedefs

- Typedefs can be global members of classes and accessed as such:
  ```cpp
typedef itk::Image<double, 3> OutputType;
  OutputType::Pointer im = filter1.GetOutput();
```
- In template classes, member typedefs are often defined in terms of template parameters—no problem! This is actually quite handy:
  ```cpp
typedef itk::Image<TPixel, 3> InputType;
```

Naming of templates and typedefs

- ITK uses the following conventions:
  - Template parameters are indicated by T (for type) or V (for value). E.g. TPixel means “the type of the pixel” and VImageDimension means “value template parameter image dimension”
  - Defined types are named as FooType. E.g. CharImage5DType

Be careful

- If you’re careless in naming classes, template arguments, typedefs, and member variables (with the “m_” prefix), then it can be quite difficult to tell them apart!
- Don’t write a new language using typedefs.
- Remember to comment well and don’t use obscure names
  - e.g. BPType is bad, BoundaryPointType is good

Typenames

- `typename` is a keyword you will learn to dislike
  - Think of it as existing to optionally help the compiler
  - Different compilers handle it differently
  - In general, you can take it to mean that its target is “some sort of type, but you’re not sure what kind”

For example:
```
typename SomeType typeInstance;
```

“typename” tells the compiler that SomeType is the name of a valid type, and not just a nonsense word

Typenames, cont.

- Windows and older Mac compilers seem to largely ignore typenames—in fact, some old Mac compilers insist they’re “deprecated”
- On Mac and Linux, you may need to preface template parameter types with typename
- My advice: try adding typename if something looks correct and won’t compile
For more on “typename”


.hxx, .cxx, .h

- ITK uses three standard file extensions, and so should you:
  - .h files indicate a class header file
  - .cxx indicates either
    - executable code (an example, test, demo, etc.)
    - a non-templated class implementation
  - .hxx indicates a templated class implementation
    - Like a .cxx file, but it can’t be compiled by itself because it does not specify its template parameter values
    - FYI, previous versions of ITK used .txx instead of .hxx

Did this all make sense?

- It’s ok if you’re a little rusty on the details, etc.
- It’s helpful if you have seen and used some of this stuff before.
- If this is mostly new to you:
  - Understand that neither I nor the TA will teach you how to do basic programming in Python or C++
  - You should probably use mostly SimpleITK
  - Beware that SimpleITK lacks some features of ITK, including several types of registration
  - If you don’t know how to write and compile C++ programs, then I recommend using Python!
- CMU 15-112: https://www.cs.cmu.edu/15-112/
- You could also take a class on C++

Final advice

- If you run across something in ITK you don’t understand, don’t panic
  - Be careful not to confuse typedefs with classes
  - Error messages can be quite long with templates and will take time to get used to
  - Email for help sooner rather than later
- Learning the style of C++ used by native ITK is at least half the battle to writing ITK Code
- Remember, if you just need to use common ITK functionality, then SimpleITK is usually the way to go!