ITK Style Guide

Insight Consortium

June 2001

1 Purpose

The following document is a description of the accepted coding style for the NLM Insight Segmentation and Registration Toolkit (ITK). Developers who wish to contribute code to ITK should read and adhere to the standards described here.

2 Document Overview

This document is organized into the following sections.

- System Overview & Philosophy — coding methodologies and motivation for the resulting style.
- Copyright — the copyright header to be included in all files and other copyright issues.
- File organization — how to organize source code; a guide to the ITK directory structure
- Naming conventions — patterns used to name classes, variables, template parameters, and instance variables.
- Namespaces — the use of namespaces.
- Code Layout and Indentation — accepted standards for arranging code including indentation style.
- Doxygen Documentation System — how to insert the appropriate commands to control the Doxygen documentation system.
- Using Standard Macros (itkMacro.h) — use of standard macros in header files.
- Exception Handling — how to add exception handling to the system.
- Documentation Style — a brief section describing the documentation philosophy inherited by the Insight consortium.

This style guide is an evolving document. Please dialog with the ITK developers if you wish to add, modify, or delete these guidelines. See `http://public.kitware.com/Insight/Web/HTML/-MailingLists.htm` for more information about joining the ITK developers mailing list. This forum is one of the best venues in which to propose changes to these style guidelines.

3 Style Guidelines

The following coding-style guidelines have been adopted by the Insight consortium. To a large extent these guidelines are a result of the fundamental architectural and implementation decisions made early in the project. For example, the decision was made to implement ITK with a C++ core using principles of generic programming. Some guidelines are relatively arbitrary, such as indentation levels and style. However, an attempt was made to find coding styles consistent with accepted practices. The point is to adhere to a common style to assist developers and users of the future learn, use, maintain, and extend ITK. (See the ITK Requirements document for more information.)

Please do your best to be a upstanding member of the ITK community. The rules described here have been developed with the community as a whole in mind. If you consistently violate these rules you will likely be harassed mercilessly, first privately and then publically. If this does not result in correct code layout, your right to CVS write access (if you are developer and wish to contribute code) may be removed. Similarly, if you are not yet currently a developer, your code will not be considered until the style is corrected.

3.1 System Overview & Philosophy

The following implementation strategies have been adopted by the Insight consortium. These directly and indirectly affect the resulting code style.

3.1.1 Implementation Language

The core implementation language is C++. C++ was chosen for its flexibility, performance, and familiarity to consortium members. (Auxiliary, interpreted language bindings such as Tcl, Python, and/or Java are also planned. These are aimply (run-time interpreted) layers on the C++ code.) The ITK consortium uses the full spectrum of C++ features including const and volatile correctness, namespaces, partial template specialization, operator overloading, traits, and iterators.

3.1.2 Generic Programming and the STL

Compile-time binding using methods of generic programming and template instantiation is the preferred implementation style. This approach has demonstrated its ability to create efficient, flexible code. Use of the STL (Standard Template Library) is encouraged. STL is typically *used* by a class, rather than as serving as a base class for derivation of ITK clases. Other STL influences are iterators and traits. ITK defines a large set of iterators; however, the ITK iterator style differs in many cases from STL because STL iterators follow a linear traversal model; ITK iterators are often designed for 2D, 3D, and even n-D traversal. Traits are used heavily be ITK. ITK naming

conventions supersede STL naming conventions; this difference is useful in that it indicates to the developer something of the boundary between ITK and STL.

3.1.3 Portability

ITK is designed to compile on a set of target operating system/compiler combinations. These targets are

- Windows 9x/NT/2000 running Microsoft Visual C++ version 6.0
- Solaris with SunPro compiler
- Irix running SGI's C++ compiler
- Linux running gcc

Additionally, gcc on cygwin (and on other Unix platforms) was supported. This decision has forced the ITK developer community to back off some important C++ features (such as partial specilialization) because of limitations in compilers (MSVC).

3.1.4 Multi-Layer Architecture

ITK is designed with a multi-layer architecture in mind. That is, three layers, a templated layer, a run-time layer, and an application layer. The templated (or generic) layer is written in C++ and requires significant programming skills and domain knowledge. The run-time layer is generated automatically using the CABLE wrapping system to produce language bindings to Tcl, Python, and Java. The interpreted layer is easier to use than the templated layer, and can be used for prototyping and smaller-sized application development. Finally, the application layer is not directly addressed by ITK other than providing simple examples of applications.

3.1.5 CMake Build Environment

The ITK build environment is CMake. CMake is an open-source, advanced cross-platform build system that enables developers to write simple "makefiles" (named CMakeLists.txt) that are processed to generated native build tools for a particular operating system/compiler combinations. See the CMake web pages at http://public.kitware.com/CMake for more information.

3.1.6 Doxygen Documentation System

The Doxygen open-source system is used to generate on-line documentation. Doxygen requires the embedding of simple comments in the code which is in turn extracted and formatted into documentation. See xxx for more information about Doxygen.

3.1.7 vnl Math Library

ITK has adopted the vnl math library. Vnl is a portion of the vxl image understanding environment. See xxx for more information about vxl and vnl.

3.1.8 Reference Counting

ITK has adopted reference counting via so-called "smart pointers" to manage object references. While alternative approaches such as automatic garbage collection were considered, their overhead due to memory requirements, performance, and lack of control as to when to delete memory, precluded these methods. Smart pointers manage the reference to objects, automatically incrementing and deleting an instance's reference count, deleting the object when the count goes to zero.

3.2 Copyright

ITK has adopted a standard copyright. This copyright should be placed at the head of every source code file. The current copyright reads as follows:

```
/*=========================================================================

  Program:   Insight Segmentation & Registration Toolkit
  Module:    $RCSfile: itkObject.h,v $
  Language:  C++
  Date:      $Date: 2001/05/04 20:31:56 $
  Version:   $Revision: 1.21 $

Copyright (c) 2001 Insight Consortium
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

 * The name of the Insight Consortium, nor the names of any consortium members,
   nor of any contributors, may be used to endorse or promote products derived
   from this software without specific prior written permission.

  * Modified source versions must be plainly marked as such, and must not be
    misrepresented as being the original software.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS ``AS IS''
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
```

```
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=========================================================================*/
```

Note the use of embedded CVS commands at the top of the header. These should be used in each file.

3.3 File organization

Classes are created and (usually) organized into a single class per file set. A file set consists of .h header file, .cxx implementation file, and/or a .txx templated implementation file. Helper classes may also be defined in the file set, typically these are not visible to the system at large, or placed into a special namespace.

3.4 Naming conventions

In general, names are constructued by using case change to indicate separate words, as in TimeStamp (versus Time_Stamp). Underscores are not used. Variable names are chosen carefully with the intention to convey the meaning behind the code. Names are generally spelled out; use of abbreviations is discouraged. (Abbreviation are allowable when in common use, and should be in uppercase as in RGB.) While this does result in long names, it self-documents the code. If you learn how to use name completion in your editor (e.g., emacs), this inconvenience can be minimized.

Depending on whether the name is a class, file, variable, or other name, variations on this theme result as explained in the following subsections.

3.4.1 Naming Classes

Classes are named beggining with a capital letter. Classes are placed in the appropriate namespace, typically itk:: (see namespaces below).

3.4.2 Naming Files

Files should have the same name as the class, with an "itk" prepended. Header files are named .h, while implementation files are named either .cxx or .txx, depending on whether they are implementations of templated classes. For example, the class itk::Image is declared and defined in the files itkImage.h and itkImage.txx (because Image is templated). The class itk::Object is declared and defined in the files itkObject.h and itkObject.cxx.

3.4.3 Naming Methods and Functions

Global functions and class methods, either static or class members, are named beginning with a capital letter. The biggest challenge when naming methods and functions is to be consistent

with existing names. For example, given the choice between ComputeBoundingBox() and CalculateBoundingBox (CalcBoundingBox() is not allowed because it is not spelled out), the choice is ComputeBoundingBox() because "Compute" is used elsewhere in the system for in similar settings.

When referring to class methods in code, an explicit "this-¿" pointer should be used, as in this-¿ComputeBoundingBox(). The use of the explicit this-¿ pointer helps clarify exactly which method, and where it originates, is being invoked. Similarly the "::" global namespace should be used when referring to a global function.

3.4.4 Naming Class Data Members

Class data members are prepended with "m_" as in m_Size. This clearly indicates the origin of data members, and differentiates them from all other variables.

3.4.5 Naming Local Variables

Local variables begin in lowercase. There is more flexibility in the naming of local variables; please remember that others will study, maintain, fix, and extend your code. Any bread crumbs that you can drop in the way of explanatory variable names and comments will go a long way towards helping other developers.

3.4.6 Naming Template Parameters

Template parameters follow the usual rules with naming except that they should start with either the capital letter `T` or `V`. Type parameters begin with the letter `T` while value template parameters begin with the letter `V`.

3.4.7 Naming Typedefs

Typedefs are absolutely essential in generic programming. They significantly improve the readibility of code, and facilitate the declaration of complex syntactic combinations. Unfortuneatly, creation of typedefs is tantamount to creating another programming language. Hence typedefs must be used in a consistent fashion.

The general rule for typedef names is that they end in the word `Type`. For example

```
typedef TPixel PixelType;
```

However, there are many exceptions to this rule that recognize that ITK has several important *concepts* that are expressed partially in the names used to implement the concept. An iterator is a concept, as is a container or pointer. These concepts are used in preference to `Type` at the end of a typedef as appropriate. For example

```
typedef typename ImageTraits::PixelContainer PixelContainer;
```

Here `Container` is a concept used in place of `Type`.

ITK currently identifies the following concepts used when naming typedefs.

- `Self` as in "`typedef Image Self;`". All classes should define this typedef.
- `Superclass` as in `typedef ImageBase<VImageDimension> Superclass;`. All classes should define the `Superclass` typedef.
- `Pointer` as in a smart pointer to an object as in `typedef SmartPointer<Self> Pointer;`
- `Container` is a type of container class.
- `Iterator` an iterator over some container class.
- `Identifier` or id such as a point or cell id.

3.4.8 Using Underscores

Don't use them. The only exception is when defining preprocessor variables and macros (which are discouraged). In this case, underscores are allowed to separate words.

3.4.9 Preprocessor Directives

Some of the worst code contains many preprocessor directives and macros. Do not use them except in a very limited sense (to support minor differences in compilers or operating systems). If a class makes extensive use of preprocessor directives, it is a candidate for separation into its own class.

3.5 Namespaces

All classes should be placed in the itk:: namespace. Additional sub-namespaces are being designed to support special functionality. Please see current documentation to determine if there is a sub-namespace is relevant to your situation. Normally sub-namespaces are used for helper ITK classes.

3.6 Const Correctness

Const correctness is important. Please use it as approriate to your class or method.

3.7 Code Layout and Indentation

The following are the accepted ITK code layout rules and indentation style. After reading this section, you may wish to visit many of the source files found in ITK. This will help crystalize the rules described here.

3.7.1 General Layout

Each line of code should take no more than 79 characters. Break the code across multiple lines as necessary. Use lots of whitespace to separate logical blocks of code, intermixed with comments. To a large extent the structure of code directly expresses its implementation.

The appropriate indentation level is two spaces for each level of indentation. DO NOT USE TABS. Set up your editor to insert spaces. Using tabs may look good in your editor but will wreak havoc in someone elses.

The declaration of variables within classes, methods, and functions should be one declaration per line.

```
int   i;
int   j;
char* stringname;
```

3.7.2 Class Layout

Classes are defined using the following guidelines.

- Begin with `#include` guards.
- Follow with the necessary includes. Include only what is necessary to avoid dependency problems.
- Place the class in the correct namespace.
- Public methods come first.
- Protected methods follow.
- Private members come last.
- Public data members are forbidden.
- Templated classes require a special preprocessor directive to control the manual instantiation of templates. See the example below and look for `ITK_MANUAL_INSTANTIATION`.)

The class layout looks something like this:

```
#ifndef __itkImage_h
#define __itkImage_h

#include "itkImageBase.h"
#include "itkPixelTraits.h"
#include "itkDefaultImageTraits.h"
#include "itkDefaultDataAccessor.h"

namespace itk
{
template <class TPixel, unsigned int VImageDimension=2,
          class TImageTraits=DefaultImageTraits< TPixel, VImageDimension > >
class ITK_EXPORT Image : public ImageBase<VImageDimension>
{
public:
```

```
  ....stuff...
protected:
  ....stuff...
private:
  ....stuff...
};

}//end of namespace

#ifndef ITK_MANUAL_INSTANTIATION
#include "itkImage.txx"
#endif

#endif //end include guard
```

3.7.3 Method Definition

Methods are defined across multiple lines. This is to accomodate the extremely long definitions possible when using templates. The starting and ending brace should be in column one. For example:

```
template<class TPixel, unsigned int VImageDimension, class TImageTraits>
const double *
Image<TPixel, VImageDimension, TImageTraits>
::GetSpacing() const
{
...
}
```

The first line is the template declaration. The second line is the method return type. The third line is the class qualifier. And the fourth line in the example above is the name of the method.

3.7.4 Use of Braces { }

Braces must be used to delimit the scope of an `if`, `for`, `while`, `switch`, or other control structure. Braces are placed on a line by themselves:

```
for (i=0; i<3; i++)
  {
  ...
  }
```

or when using an `if`:

```
if ( condition )
```

```
  {
  ...
  }
else if ( other condition )
  {
  ...
  }
else
  {
  ....
  }
```

You can choose to use braces on a line with a code block when the block consists of a single line:

```
if ( condition ) {foo=1;}
else if ( condition2 ) {foo=3;}
else {return;}
```

or

```
for (i=0; i<3; ++i) {x[i]=0.0;}
```

3.8 Doxygen Documentation System

Doxygen is an open-source, powerful system for automatically generating documentation from source code. To use Doxygen effectively, the developer must insert comments, delimited in a special way, that Doxygen extracts to produce the documentation. While there are a large number of options to Doxygen, developers at a minimum should insert the following Doxygen commands. (See more at http://www.stack.nl/ dimitri/doxygen/index.html.)

3.8.1 Documenting a Class

Classes should be documented using the
`class` and
`brief` Doxygen commands, followed by the detailed class description:

```
/** \class Object
 * \brief Base class for most itk classes.
 *
 * Object is the second-highest level base class for most itk objects.
 * It extends the base object functionality of LightObject by
 * implementing debug flags/methods and modification time tracking.
 */
```

3.8.2 Documenting a Method

Methods should be documented using the following comment block style as shown in the following example.

```
/**
 * Access a pixel. This version can be an lvalue.
 */
TPixel & operator[](const IndexType &index)
   { return this->GetPixel(index); }
```

The key here is that the comment starts with `/**`, each subsequent line has an aligned `*`, and the comment block terminates with a `*/`.

3.9 Using Standard Macros (itkMacro.h)

There are several macros defined in the file `itkMacro.h`. These macros should be used. The reason for this is that they perform several important operations, that if not done, can cause serious, but hard to debug problems, in the system. These operations are:

- Object modified time is properly managed.
- Debug information is printed.
- Reference counting is handled properly.

Please review this file and become familiar with these macros.

3.10 Exception Handling

Indicate that methods throw exceptions in the method declaration as in:

```
const float* foo() const throws itk
```

4 Documentation Style

The Insight consortium has adopted the following guidelines for producing supplemental documentation (documentation not produced by Doxygen).

- The common denominator for documentation is either PDF or HTML. All documents in the system should be available in these formats, even if they are mastered by another system.
- Presentations are acceptable in Microsoft PowerPoint format.
- Administrative and planning documents are acceptable in Microsoft Word format (either .doc or .rtf).
- Larger documents, such as the user's or developer's guide, are written in LaTeX.