A Few Coding Standards

Introduction

This document attempts to describe a few coding standards that are being used in the LLVM source tree. Although no coding standards should be regarded as absolute requirements to be followed in all instances, coding standards can be useful.

This document intentionally does not prescribe fixed standards for religious issues such as brace placement and space usage. For issues like this, follow the golden rule:

> **If you are adding a significant body of source to a project, feel free to use whatever style you are most comfortable with. If you are extending, enhancing, or bug fixing already implemented code, use the style that is already being used so that the source is uniform and easy to follow.**

The ultimate goal of these guidelines is the increase readability and maintainability of our common source base. If you have suggestions for topics to be included, please mail them to Chris.

Mechanical Source Issues

Source Code Formatting

Commenting

Comments are one critical part of readability and maintainability. Everyone knows they should comment, so should you. :) Although we all should probably comment our code more than we do, there are a few very critical places that documentation is very useful:

1. File Headers

Every source file should have a header on it that describes the basic purpose of the file. If a file does not have a header, it should not be checked into CVS. Most source trees will probably have a standard file header format. The standard format for the LLVM source tree looks like this:

```
//===-- llvm/Instruction.h - Instruction class definition -------*- C++ -*-===//
//
// This file contains the declaration of the Instruction class, which is the
// base class for all of the VM instructions.
//
//===----------------------------------------------------------------------===//
```

A few things to note about this particular format. The "`-*- C++ -*-`" string on the first line is there to tell Emacs that the source file is a C++ file, not a C file (Emacs assumes .h files are C files by default [Note that tag this is not necessary in .cpp files]). The name of the file is also on the first line, along with a very short description of the purpose of the file. This is important when printing out code and flipping though lots of pages.

The main body of the description does not have to be very long in most cases. Here it's only two lines. If an algorithm is being implemented or something tricky is going on, a reference to the paper where it is published should be included, as well as any notes or "gotchas" in the code to watch out for.

2. Class overviews

Classes are one fundemental part of a good object oriented design. As such, a class definition should have a comment block that explains what the class is used for... if it's not obvious. If it's so completely obvious your grandma could figure it out, it's probably safe to leave it out. Naming classes something sane goes a long ways towards avoiding writing documentation. :)

3. Method information

Methods defined in a class (as well as any global functions) should also be documented properly. A quick note about what it does any a description of the borderline behaviour is all that is necessary here (unless something particularly tricky or insideous is going on). The hope is that people can figure out how to use your interfaces without reading the code itself... that is the goal metric.

Good things to talk about here are what happens when something unexpected happens: does the method return null? Abort? Format your hard disk?

Comment Formatting

In general, prefer C++ style (`//`) comments. They take less space, require less typing, don't have nesting problems, etc. There are a few cases when it is useful to use C style (`/* */`) comments however:

1. When writing a C code: Obviously if you are writing C code, use C style comments. :)
2. When writing a header file that may be #included by a C source file.

3. When writing a source file that is used by a tool that only accepts C style comments.

To comment out a large block of code, use `#if 0` and `#endif`. These nest properly and are better behaved in general than C style comments.

#include Style

Immediately after the header file comment (and include guards if working on a header file), the minimal list of #includes required by the file should be listed. We prefer these #includes to be listed in this order:

1. Main Module header
2. Local/Private Headers
3. llvm/*
4. llvm/Analysis/*
5. llvm/Assembly/*
6. llvm/Bytecode/*
7. llvm/CodeGen/*
8. ...
9. Support/*
10. Config/*
11. System #includes

... and each catagory should be sorted by name.

The "Main Module Header" file applies to .cpp file which implement an interface defined by a .h file. This #include should always be included **first** regardless of where it lives on the file system. By including a header file first in the .cpp files that implement the interfaces, we ensure that the header does not have any hidden dependencies which are not explicitly #included in the header, but should be. It is also a form of documentation in the .cpp file to indicate where the interfaces it implements are defined.

Source Code Width

Write your code to fit within 80 columns of text. This helps those of us who like to print out code and look at your code in an xterm without resizing it.

Use Spaces Instead of Tabs

In all cases, prefer spaces to tabs in source files. People have different prefered indentation levels, and different styles of indentation that they like... this is fine. What isn't is that different editors/viewers expand tabs out to different tab stops. This can cause your code to look completely unreadable, and it is not worth dealing with.

As always, follow the Golden Rule above: follow the style of existing code if your are modifying and extending it. If you like four spaces of indentation, **DO NOT** do that in the middle of a chunk of code with two spaces of indentation. Also, do not reindent a whole source file: it makes for incredible diffs that are absolutely worthless.

Indent Code Consistently

Okay, your first year of programming you were told that indentation is important. If you didn't believe and internalize this then, now is the time. Just do it.

Treat Compiler Warnings Like Errors

If your code has compiler warnings in it, something is wrong: you aren't casting values correctly, your have "questionable" constructs in your code, or you are doing something legitimately wrong. Compiler warnings can cover up legitimate errors in output and make dealing with a translation unit difficult.

It is not possible to prevent all warnings from all compilers, nor is it desirable. Instead, pick a standard compiler (like `gcc`) that provides a good thorough set of warnings, and stick to them. At least in the case of `gcc`, it is possible to work around any spurious errors by changing the syntax of the code slightly. For example, an warning that annoys me occurs when I write code like this:

```
if (V = getValue()) {
  ..
}
```

`gcc` will warn me that I probably want to use the `==` operator, and that I probably mistyped it. In most cases, I haven't, and I really don't want the spurious errors. To fix this particular problem, I rewrite the code like this:

```
if ((V = getValue())) {
  ..
}
```

...which shuts `gcc` up. Any `gcc` warning that annoys you can be fixed by massaging the code appropriately.

These are the `gcc` warnings that I prefer to enable: `-Wall -Winline -W -Wwrite-strings -Wno-unused`

Which C++ features can I use?

Compilers are finally catching up to the C++ standard. Most compilers implement most features, so you can use just about any features that you would like. In the LLVM source tree, I have chosen to not use these features:

1. Exceptions: Exceptions are very useful for error reporting and handling exceptional conditions. I do not use them in LLVM because they do have an associated performance impact (by restricting restructuring of code), and parts of LLVM are designed for performance critical purposes.

   Just like most of the rules in this document, this isn't a hard and fast requirement. Exceptions are used in the Parser, because it simplifies error reporting **significantly**, and the LLVM parser is not at all in the critical path.

2. RTTI: RTTI has a large cost in terms of executable size, and compilers are not yet very good at stomping out "dead" class information blocks. Because of this, typeinfo and dynamic cast are not used.

Other features, such as templates (without partial specialization) can be used freely. The general goal is to have clear, consise, performant code... if a technique assists with that then use it.

Write Portable Code

In almost all cases, it is possible and within reason to write completely portable code. If there are cases where it isn't possible to write portable code, isolate it behind a well defined (and well documented) interface.

In practice, this means that you shouldn't assume much about the host compiler, including its support for "high tech" features like partial specialization of templates. In fact, Visual C++ 6 could be an important target for our work in the future, and we don't want to have to rewrite all of our code to support it.

Style Issues

The High Level Issues

A Public Header File is a Module

C++ doesn't do too well in the modularity department. There is no real encapsulation or data hiding (unless you use expensive protocol classes), but it is what we have to work with. When you write a public header file (in the LLVM source tree, they live in the top level "include" directory), you are defining a module of functionality.

Ideally, modules should be completely independent of each other, and their header files should only include the absolute minimum number of headers possible. A module is not just a class, a function, or a namespace: it's a collection of these that defines an interface. This interface may be several functions, classes or data structures, but the important issue is how they work together.

In general, a module should be implemented with one or more `.cpp` files. Each of these `.cpp` files should include the header that defines their interface first. This ensure that all of the dependences of the module header have been properly added to the module header itself, and are not implicit. System headers should be included after user headers for a translation unit.

#include as Little as Possible

`#include` hurts compile time performance. Don't do it unless you have to, especially in header files.

But wait, sometimes you need to have the definition of a class to use it, or to inherit from it. In these cases go ahead and #include that header file. Be aware however that there are many cases where you don't need to have the full definition of a class. If you are using a pointer or reference to a class, you don't need the header file. If you are simply returning a class instance from a prototyped function or method, you don't need it. In fact, for most cases, you simply don't need the definition of a class... and not `#include`'ing speeds up compilation.

It is easy to try to go too overboard on this recommendation, however. You **must** include all of the header files that you are using, either directly or indirectly (through another header file). To make sure that you don't accidently forget to include a header file in your module header, make sure to include your module header **first** in the implementation file (as mentioned above). This way there won't be any hidden dependencies that you'll find out about later...

Keep "internal" Headers Private

Many modules have a complex implementation that causes them to use more than one implementation (`.cpp`) file. It is often tempting to put the internal communication interface (helper classes, extra functions, etc) in the public module header file. Don't do this. :)

If you really need to do something like this, put a private header file in the same directory as the source files, and include it locally. This ensures that your private interface remains private and undisturbed by outsiders.

Note however, that it's okay to put extra implementation methods a public class itself... just make them private (or protected), and all is well.

The Low Level Issues

Assert Liberally

Use the "`assert`" function to its fullest. Check all of your preconditions and assumptions, you never know when a bug (not neccesarily even yours) might be caught early by an assertion, which reduces debugging time dramatically. The "`<cassert>`" header file is probably already included by the header files you are using, so it doesn't cost anything to use it.

To further assist with debugging, make sure to put some kind of error message in the assertion statement (which is printed if the assertion is tripped). This helps the poor debugging make sense of why an assertion is being made and enforced, and hopefully what to do about it. Here is one complete example:

```
inline Value *getOperand(unsigned i) {
  assert(i < Operands.size() && "getOperand() out of range!");
  return Operands[i];
}
```

Here are some examples:

```
assert(Ty->isPointerType() && "Can't allocate a non pointer type!");

assert((Opcode == Shl || Opcode == Shr) && "ShiftInst Opcode invalid!");

assert(idx < getNumSuccessors() && "Successor # out of range!");

assert(V1.getType() == V2.getType() && "Constant types must be identical!");

assert(isa<PHINode>(Succ->front()) && "Only works on PHId BBs!");
```

You get the idea...

Prefer Preincrement

Hard fast rule: Preincrement (++X) may be no slower than postincrement (X++) and could very well be a lot faster than it. Use preincrementation whenever possible.

The semantics of postincrement include making a copy of the value being incremented, returning it, and then preincrementing the "work value". For primitive types, this isn't a big deal... but for iterators, it can be a huge issue (for example, some iterators contains stack and set objects in them... copying an iterator could invoke the copy ctor's of these as well). In general, get in the habit of always using preincrement, and you won't have a problem.

Avoid endl

The `endl` modifier, when used with iostreams outputs a newline to the output stream specified. In addition to doing this, however, it also flushes the output stream. In other words, these are equivalent:

```
cout << endl;
cout << "\n" << flush;
```

Most of the time, you probably have no reason to flush the output stream, so it's better to use a literal `"\n"`.

Exploit C++ to its Fullest

C++ is a powerful language. With a firm grasp on its capabilities, you can make write effective, consise, readable and maintainable code all at the same time. By staying consistent, you reduce the amount of special cases that need to be remembered. Reducing the total number of lines of code you write is a good way to avoid documentation, and avoid giving bugs a place to hide.

For these reasons, come to know and love the contents of your local <algorithm> header file. Know about <functional> and what it can do for you. C++ is just a tool that wants you to master it. :)

Writing Iterators

Here's a pretty good summary of how to write your own data structure iterators in a way that is compatible with the STL, and with a lot of other code out there (slightly edited by Chris):

```
From: Ross Smith
Newsgroups: comp.lang.c++.moderated
Subject: Writing iterators (was: Re: Non-template functions that take iterators)
Date: 28 Jun 2001 12:07:10 -0400

Andre Majorel wrote:
> Any pointers handy on "writing STL-compatible iterators for
> dummies ?"

I'll give it a try...

The usual situation requiring user-defined iterators is that you have
a type that bears some resemblance to an STL container, and you want
to provide iterators so it can be used with STL algorithms. You need
to ask three questions:

First, is this simply a wrapper for an underlying collection of
objects that's held somewhere as a real STL container, or is it a
"virtual container" for which iteration is (under the hood) more
complicated than simply incrementing some underlying iterator (or
pointer or index or whatever)? In the former case you can frequently
get away with making your container's iterators simply typedefs for
those of the underlying container; your begin() function would call
member_container.begin(), and so on.

Second, do you only need read-only iterators, or do you need separate
read-only (const) and read-write (non-const) iterators?

Third, which kind of iterator (input, output, forward, bidirectional,
or random access) is appropriate? If you're familiar with the
properties of the iterator types (if not, visit
http://www.sgi.com/tech/stl/), the appropriate choice should be
obvious from the semantics of the container.

I'll start with forward iterators, as the simplest case that's likely
to come up in normal code. Input and output iterators have some odd
properties and rarely need to be implemented in user code; I'll leave
them out of discussion. Bidirectional and random access iterators are
covered below.

The exact behaviour of a forward iterator is spelled out in the
```

Standard in terms of a set of expressions with specified behaviour, rather than a set of member functions, which leaves some leeway in how you actually implement it. Typically it looks something like this (I'll start with the const-iterator-only situation):

```
  #include <iterator>

  class container {
    public:
      typedef something_or_other value_type;
      class const_iterator:
        public std::iterator<std::forward_iterator_tag, value_type> {
          friend class container;
          public:
            const value_type& operator*() const;
            const value_type* operator->() const;
            const_iterator& operator++();
            const_iterator operator++(int);
            friend bool operator==(const_iterator lhs,
                                   const_iterator rhs);
            friend bool operator!=(const_iterator lhs,
                                   const_iterator rhs);
          private:
            //...
        };
      //...
  };
```

An iterator should always be derived from an instantiation of the std::iterator template. The iterator's life cycle functions (constructors, destructor, and assignment operator) aren't declared here; in most cases the compiler-generated ones are sufficient. The container needs to be a friend of the iterator so that the container's begin() and end() functions can fill in the iterator's private members with the appropriate values.

[Chris's Note: I prefer to not make my iterators friends. Instead, two ctor's are provided for the iterator class: one to start at the end of the container, and one at the beginning. Typically this is done by providing two constructors with different signatures.]

There are normally only three member functions that need nontrivial implementations; the rest are just boilerplate.

```
  const container::value_type&
    container::const_iterator::operator*() const {
      // find the element and return a reference to it
    }

  const container::value_type*
    container::const_iterator::operator->() const {
      return &**this;
    }
```

If there's an underlying real container, operator*() can just return a reference to the appropriate element. If there's no actual container and the elements need to be generated on the fly -- what I think of as a "virtual container" -- things get a bit more complicated; you'll probably need to give the iterator a value_type member object, and fill it in when you need to. This might be done as part of the increment operator (below), or if the operation is nontrivial, you might choose the "lazy" approach and only generate the actual value when one of the dereferencing operators is called.

The operator->() function is just boilerplate around a call to

operator*().

```
  container::const_iterator&
    container::const_iterator::operator++() {
      // the incrementing logic goes here
      return *this;
    }

  container::const_iterator
    container::const_iterator::operator++(int) {
      const_iterator old(*this);
      ++*this;
      return old;
    }
```

Again, the incrementing logic will usually be trivial if there's a real container involved, more complicated if you're working with a virtual container. In particular, watch out for what happens when you increment past the last valid item -- this needs to produce an iterator that will compare equal to container.end(), and making this work is often nontrivial for virtual containers.

The post-increment function is just boilerplate again (and incidentally makes it obvious why all the experts recommend using pre-increment wherever possible).

```
  bool operator==(container::const_iterator lhs,
                  container::const_iterator rhs) {
    // equality comparison goes here
  }

  bool operator!=(container::const_iterator lhs,
                  container::const_iterator rhs) {
    return !(lhs == rhs);
  }
```

For a real container, the equality comparison will usually just compare the underlying iterators (or pointers or indices or whatever). The semantics of comparisons for virtual container iterators are often tricky. Remember that iterator comparison only needs to be defined for iterators into the same container, so you can often simplify things by taking for granted that lhs and rhs both point into the same container object. Again, the second function is just boilerplate.

It's a matter of taste whether iterator arguments are passed by value or reference; I've shown tham passed by value to reduce clutter, but if the iterator contains several data members, passing by reference may be better.

That convers the const-iterator-only situation. When we need separate const and mutable iterators, one small complication is added beyond the simple addition of a second class.

```
  class container {
    public:
      typedef something_or_other value_type;
      class const_iterator;
      class iterator:
        public std::iterator<std::forward_iterator_tag, value_type> {
          friend class container;
          friend class container::const_iterator;
          public:
            value_type& operator*() const;
            value_type* operator->() const;
            iterator& operator++();
```

```
        iterator operator++(int);
        friend bool operator==(iterator lhs, iterator rhs);
        friend bool operator!=(iterator lhs, iterator rhs);
      private:
        //...
    };
    class const_iterator:
      public std::iterator<std::forward_iterator_tag, value_type> {
        friend class container;
        public:
          const_iterator();
          const_iterator(const iterator& i);
          const value_type& operator*() const;
          const value_type* operator->() const;
          const_iterator& operator++();
          const_iterator operator++(int);
          friend bool operator==(const_iterator lhs,
                                 const_iterator rhs);
          friend bool operator!=(const_iterator lhs,
                                 const_iterator rhs);
        private:
          //...
      };
    //...
};
```

There needs to be a conversion from iterator to const_iterator (so that mixed-type operations, such as comparison between an iterator and a const_iterator, will work). This is done here by giving const_iterator a conversion constructor from iterator (equivalently, we could have given iterator an operator const_iterator()), which requires const_iterator to be a friend of iterator, so it can copy its data members. (It also requires the addition of an explicit default constructor to const_iterator, since the existence of another user-defined constructor inhibits the compiler-defined one.)

Bidirectional iterators add just two member functions to forward iterators:

```
class iterator:
  public std::iterator<std::bidirectional_iterator_tag, value_type> {
    public:
      //...
      iterator& operator--();
      iterator operator--(int);
      //...
};
```

I won't detail the implementations, they're obvious variations on operator++().

Random access iterators add several more member and friend functions:

```
class iterator:
  public std::iterator<std::random_access_iterator_tag, value_type> {
    public:
      //...
      iterator& operator+=(difference_type rhs);
      iterator& operator-=(difference_type rhs);
      friend iterator operator+(iterator lhs, difference_type rhs);
      friend iterator operator+(difference_type lhs, iterator rhs);
      friend iterator operator-(iterator lhs, difference_type rhs);
      friend difference_type operator-(iterator lhs, iterator rhs);
      friend bool operator<(iterator lhs, iterator rhs);
      friend bool operator>(iterator lhs, iterator rhs);
```

```
        friend bool operator<=(iterator lhs, iterator rhs);
        friend bool operator>=(iterator lhs, iterator rhs);
        //...
    };

  container::iterator&
    container::iterator::operator+=(container::difference_type rhs) {
      // add rhs to iterator position
      return *this;
    }

  container::iterator&
    container::iterator::operator-=(container::difference_type rhs) {
      // subtract rhs from iterator position
      return *this;
    }

  container::iterator operator+(container::iterator lhs,
                                container::difference_type rhs) {
    return iterator(lhs) += rhs;
  }

  container::iterator operator+(container::difference_type lhs,
                                container::iterator rhs) {
    return iterator(rhs) += lhs;
  }

  container::iterator operator-(container::iterator lhs,
                                container::difference_type rhs) {
    return iterator(lhs) -= rhs;
  }

  container::difference_type operator-(container::iterator lhs,
                                       container::iterator rhs) {
    // calculate distance between iterators
  }

  bool operator<(container::iterator lhs, container::iterator rhs) {
    // perform less-than comparison
  }

  bool operator>(container::iterator lhs, container::iterator rhs) {
    return rhs < lhs;
  }

  bool operator<=(container::iterator lhs, container::iterator rhs) {
    return !(rhs < lhs);
  }

  bool operator>=(container::iterator lhs, container::iterator rhs) {
    return !(lhs < rhs);
  }
```

Four of the functions (operator+=(), operator-=(), the second operator-(), and operator<()) are nontrivial; the rest are boilerplate.

One feature of the above code that some experts may disapprove of is the declaration of all the free functions as friends, when in fact only a few of them need direct access to the iterator's private data. I originally got into the habit of doing this simply to keep the declarations together; declaring some functions inside the class and some outside seemed awkward. Since then, though, I've been told that there's a subtle difference in the way name lookup works for functions declared inside a class (as friends) and outside, so keeping them

```
together in the class is probably a good idea for practical as well as
aesthetic reasons.

I hope all this is some help to anyone who needs to write their own
STL-like containers and iterators.

--
Ross Smith <ross.s@ihug.co.nz> The Internet Group, Auckland, New Zealand
```

See Also

A lot of these comments and recommendations have been culled for other sources. Two particularly important books for our work are:

1. Effective C++ by Scott Meyers. There is an online version of the book (only some chapters though) available as well.
2. Large-Scale C++ Software Design by John Lakos

If you get some free time, and you haven't read them: do so, you might learn something. :)

Chris Lattner
The LLVM Compiler Infrastructure
Last modified: Sun Oct 12 22:12:43 CDT 2003