Some Programming Style Suggestions

Peter Schröder

Writing Large Programs

Writing small programs that will be thrown away when you are done with them do not require you to be particularly careful or establish particular styles. Writing large programs that are supposed to last beyond the next homework deadline are a different matter altogether! This note is intended to give you some simple to follow suggestions that will save you **many** headaches. Unless you have already written industrial strength programs you will find that many of these rules sound overly anal. Trust me, they were born out of many tears and much blood! I don't want to sound like your mother, but much of this is simply a matter of building good habits to avoid a mess later on.

Before going into very specific suggestions we'll start with some general ideas.

Basic Rules of the Game

Think: Think about your algorithm first, spec it out, don't just start hacking right away. Sometimes it is useful to write a small program to try out an idea. This is good. But then, please, throw that code away. Don't make a quick hack the basis for a larger code. You'll regret it later. On the other hand don't try to overdo it. Attempting to come up with the optimal class design considering all possible cases before you have the first running program generally doesn't work. Some amount of figuring out the right way of doing things only occurs when starting to write code.

Length: As a general rule functions and methods should fit on one screen. This is not always possible, but you should ask yourself why you are getting such a long function if you go beyond a screen's worth. Files, too, shouldn't be too long. Group functions which build a conceptual group together in a file. Don't put everything into a single file. It hurts compiler performance and makes it too hard to find your way around. Now it becomes important that you give files a name which advertises what's inside. Once you have 40 files in a given project you'll appreciate this.

Writing optimal code: Get it right first, then worry about making it fast.

Conventions: Establish and maintain conventions throughout your coding. E.g., the order of source and destination arguments in function calls:

```
void* copy( void* dst, const void* src, const int nbytes );
int* copy( const int* src, int* dst, const int nints );
```

Whichever way you prefer, stick to it! Another example of a useful convention is to spell macros and preprocessor constants in all CAPS.

Whitespace: I trust everyone already indents their code... Equally important is the use of spaces:

```
for(i=obj->begin();i<obj->l()&&obj->M(i)!=-1;i++){
  obj->M(i)=-1;
}
```

isalothardertoreadthanthefollowingequivalentstatement

```
for( i = obj->begin(); i < obj->l() && obj->f(i) != -1; i++ ){
  obj->f(i) = -1;
}
```

Similarly add empty lines between conceptual groups of statements just like you would add paragraphs to a story.

Comments: Yes, the code says what it does, but in order to understand what it does I have to reverse engineer the meaning. Just state it:

```
// initialize all faces in obj to empty, starting with the
// beginning of the segment up to the maximum length unless
// the tail is already initialized to empty
for( i = obj->begin(); i < obj->l() && obj->f(i) != -1; i++ ){
  obj->f(i) = -1;
}
// update the begin pointer for the next segment
obj->begin() = i;
```

Trust me, it makes large complicated codes much easier to understand to state the "obvious."

Specifics

Assertions: `assert()` is by far the programmer's best friend. The macro `assert()`, which lives in `<assert.h>` checks its argument and if it evaluates to zero causes a core dump with a message printed to stdout showing the argument to `assert()` and the line number and file name of the location of the `assert()`. You can then directly bring up dbx and examine the state of your program to help you understand why something went wrong. The general rule about `assert()`s is that you should use them often and in particular for things that you know "can never go wrong." Here are a few examples:

```
Obj *obj = new Obj; assert( obj );

void
dada( Obj *obj, const int begin, const int end )
{
  int i = 0;
  assert( begin >= 0 );
  assert( end < obj->length() );

  for( i = begin; i < end; i++ ){
    obj->f(i) = -1;
  }
}

float
Angle( const Vec *a, const Vec *b )
{
  float dot = a[X] * b[X] + a[Y] * b[Y] + a[Z] * b[Z];
  float alen = Length( a ), blen = Length( b );
  float cosangle = 0;

  assert( alen > 0 ); assert( blen > 0 );
  cosangle = dot / ( alen * blen );

  assert( fabsf( cosangle ) <= 1 );
  return acosf( cosangle );
}
```

Sometimes it makes sense to use fixed length storage for something that is actually dynamic. For example, if you "know" that there'll never be more than 50 vertices that need to change. Be prepared though for the rare input that has more than these:

```
const int MAXVERT = 50;

...

void
LoadVertexIndices( int map[MAXVERT], Obj *obj )
{
  int i = 0;
  while( obj->NextIndex() != -1 ){
    map[i++] = obj->NextIndex();
    obj->IncrIndex();
    assert( i < MAXVERT );
  }
}
```

If you don't put an `assert()` here to alarm you later that you went beyond the limit of something you will trash memory and the symptoms often don't show up until much later in the execution path with extremely hard to pinpoint reasons. These kinds of memory bugs are the nastiest to find!

For other examples see the `switch()` statement below and the comment on checking return values.

Global variables: Avoid them! The problem with global variables is that it is very hard to later understand who is messing with a given variable at what time. This is particularly so when writing event driven programs which do not have a linear flow anymore. Sometimes it makes sense to create a struct for a group of such variables and pass at least a pointer to that struct to any function which manipulates these variables.

Name space pollution: Functions or variables which are only needed by a single source file should be made static so that their name never appears in the global name space. Sooner or later you'll have a clash. Once you start linking against other systems or libraries it's essential to be as tight as possible on these globally visible names.

Memory management: You allocate it, you free it. Simple. A good code is memory clean. You may say "why the effort, when I exit, everything goes back to the system." True, it does. But it's a good idea to get used to doing memory clean programming, because sooner or later one of those functions will migrate into a library and be called over and over again, and you don't want programs that depend on it to slowly run out of memory. Don't say you'll fix it then. You won't. You won't even remember that there is this problem. Once things get complicated enough, you won't even be able to find that damn memory leak anymore. Again, this is a bug in the "extremely nasty" category. Get used to avoiding it right from the start.

Initialize variables: **Always** initialize any variable as part of its definition. The nastiest thing to track down is a bug due to an uninitialized variable:

```
int
CallBack( Event *e )
{
  int result;

  if( e ){
    ...
    result = 1;
    ...
  }

  return result;
}
```

Good luck finding this bug. Only under certain circumstances will this bug be triggered and then only if there happens to be the right kind of garbage on the stack. Those bugs can be reaaaaaal fun.

Let the compiler help you: **Always** compile with all warning levels that the compiler provides, turned on. You won't believe how often the compiler can tell you about something stupid you've just done if only you give it a chance. Oh, yes, read the complaints that the compiler gives you, don't ignore them.

Prototyping: Always prototype all functions including those that do not take an argument:

```
extern void fish( void );
```

is quite different from

```
extern void fish();
```

The latter being a non-prototyped function which takes an unspecified set of arguments. Baaaaaad.

Include files: Be explicit about the include files that a given source file needs. I.e., only list those actually needed. The alternative, having a source include everything makes it very hard to understand the mutual dependencies of a code.

Fixing it later: Don't tell yourself "oh, I know this is a mess, but I'll fix it later." Trust me, later will never come. Fix it now. Taking shortcuts in an initial implementation is ok. However, use assertions to guard yourself against accidentally tripping over that shortcut, so that at least later on when you need the real version of something you just put together quickly, you will find out:

```
switch( i ){
case 'a':
  ...
  break;
case 'b':
  ...
  break;
default:
  assert( 0 );
}
```

Keyword const: Use it early and often. In many cases you give the compiler more opportunities for optimization and you more clearly advertise to the reader of your code what a given function will, or will not do. You'll also get help from the compiler in case you inadvertently try to change something you really weren't supposed to mess with:

```
extern int scanf( const char*, ... );
```

clearly states to any user that `scanf()` will not mess with it's first argument. Nice to know. Similarly the following will be caught:

```
void
dada( const Vec *v )
{
  ...
  v->x() = 5; // compiler error
}
```

This latter example is trivial, but once your code gets more complicated and you have many levels of indirection you will run into this in places where it is not obvious at all...

Version control: Have you ever hacked on a code for several days only to discover on Wednesday that something that worked on Monday doesn't seem to work anymore? Wish you still had the code from Monday to see what the hell you changed that caused the breakage? It's called "version control." Either do it by hand, saving previous versions of your code in a safe place or, much better, use a real version control system. We recommend CVS. It's very easy and it will allow you to get the version of "midnight, last Monday." If you use version control you of course have to check in your code often since you can only recover to within the nearest checkpoint.

Check return values: Many functions will (should) return a status. Take advantage of this and check for it:

```
i = fscanf( file, "%f %f %f", &x, &y, &z );
assert( i == 3 );
```

Meaningful names: Give variables meaningful names. There is a subtle tradeoff here between verbosity which cuts down on readability and terseness which cuts down on readability as well:

```
void
InitializeFaces( Obj* obj )
{
  Face *f = obj->faces();
  const int nof = obj->length();
  const int empty = -1;
  int i = 0;

  // set all faces to empty
  for( i = 0; i < nof; i++ ){
    f[i] = empty;
  }
}
```

Note that the initialization of i to zero wasn't necessary here, but it pays to build habits.

Good names for arguments in extern declarations are particularly useful:

```
extern void VertexMovedCB( Mesh *m,
                           const float newcoord[3],
                           const int meshindex,
                           SoCoordinate3* destcoords );
```

is much easier to understand than

```
extern void VertexMovedCB( Mesh*, const float*, const int, SoCoordinate3* );
```

Files and file names: Every source file should have a header file of the same name

```
vec3d.h
vec3d.c
```

The header file advertises externally what's user callable in the source file. Having the source file include the header file itself (even though generally this is not necessary) ensures that the source and header never diverge. E.g., you change the type of an argument in the source file, but forget to change it in the header file. This would typically lead to hard to find disasters unless you enlist the compiler's help in pointing out the mismatch.

Makefiles: Makefiles are your friends. Take the time to once carefully create a template Makefile that you can reuse for all your projects. In particular automate the dependency management by creating the target "depend:" which makes all the dependencies automatically. Maintaining them by hand is a mess and will simply not work for large projects. (We will give you an example Makefile.)

C++ Style

Once your start programming in OpenInventor you will have to use C++. Aside from the comments on writing style given above there are a few new things to keep in mind.

What every `class` should look like: C++ will generate default members for certain class methods if you don't do so explicitly. This is generally a bad idea since it can lead to hard to find surprises. We suggest the following basic class layout (we'll use a fictitious `class Color` as an example):

```
// forward reference to avoid having to include <iostream.h> here
class ostream;
class istream;

class Color{
public:
  Color( void );            // default constructor
  Color( const Color& );    // so-called `X( const X& )' constructor
  ~Color( void );           // *always* have a destructor

  Color operator=( const Color& );

  friend ostream& operator<<( ostream&, const Color& );
  friend istream& operator>>( istream&, Color& );

  // section for data accessors (example);
  float  r( void ) const { return p_r; }
  float& r( void )       { return p_r; }

protected:
private:
  // ALL data members should ALWAYS be at least protected if not private
  float p_r, p_g, p_b;
};
```

The default constructor (one that takes no arguments) is needed if you want to be able to declare arrays of a class type. The constructor `X( const X& )` is used when calling functions (among other places). Especially for large classes it is profitable to define this constructor explicitly since you can save yourself copy on and off the stack.

Having an explicit destructor for a class, even if it is trivial, really helps building a habit of memory clean programming. Especially in C++ this is really made very simple since all you have to do is make sure that the constructor and destructor do the right thing when it comes to dynamic memory.

The assignment operator is defined by default to do bitwise copy. If you have pointer members of a class this is rarely what you want. Thus it's a good idea to be explicit about it right from the start.

Almost always you'll want a way to print an object to a stream. By defining `operator<<()`, you can print objects of class type just like any other object with the usual stream semantics. Similarly it is useful to have a read function, `operator>>()` which should be designed to read whatever the output function generates. For some class types the read function makes little sense, e.g., if pointers are an integral member of a given class.

As a matter of principle all data members should be hidden and any access to them should be mediated through accessors. If the accessors are inline this will not cost you any performance overhead, but if you ever change your mind about the internal implementation of a class, having accessors can often make such a change trivial, rather than having to go through all your code and changing some statements. Note also the use of `const` in the two accessors above. The first one cannot be used as an Lvalue, while the second one can. You'll typically want both.

I/O: For I/O in C++ you should use `<iostream.h>` exclusively. It gives you type safe input and output and is considerably more elegant. Note that it doesn't mix well at all with `<stdio.h>` since the two use different buffer areas.

Inheritance: There is only a rare justified use of single inheritance. Multiple inheritance should definitely be avoided. It is Byzantine to say the least and will only get you into trouble.

Use of references: C++ introduces a new semantic concept to C, that of a reference. They are extremely handy since they give you the semantics of dealing with the object (rather than just a pointer to the object), yet have the storage efficiency of a pointer. Especially for large class objects this can make a big difference.

Operator overloading: One of the principle advantages of C++, especially for graphics, is operator overloading. For example, for a 3-vector class you can define all manner of arithmetic operators allowing you to have your code read almost exactly like the mathematical expressions underlying a particular computation:

```
class Vector3{
public:
  Vector3( const float x, const float y, const float z )
    : p_x( x ), p_y( y ), p_z( z ) {}

  Vector3 operator+( const Vector3& v )
    { return Vector3( p_x + v.p_x, p_y + v.p_y, p_z + v.p_z; }

protected:
private:
  float p_x, p_y, p_z;
};
```

Together with a `Matrix` class your code can be something like:

```
// create a rotation around the x-axis (with a suitable constructor)
Matrix r( 'x', m_pi_f / 6 );
// translation constructor (we assume 4x4 matrices here)
Matrix t( Vector3( 1, 1, 1 ) );
// scaling
Matrix s( .5f );

// concatenate transforms
Matrix m = r * s * t;

// transform object
for( int i = 0; i < obj->length(); i++ ){
  obj->pts( i ) *= m;
}
```

Especially when such computations become more involved, i.e., the expressions are more complicated, the C-style of calling a bunch of functions or macros in the right order with manual management of temp variables turns into a mess while the C++ version will (can!) be perfectly readable, *and* just as efficient!

When first playing with operator overloading many people succumb to overdoing it. Obviously things such as addition and scalar multiplication of vectors make sense and have an obvious operator which should do this. But what do you do about things such as dot products? For example

```
class Vector3{
public:
  ...

  float operator|( const Vector3& v )
    { return p_x * v.p_x + p_y * v.p_y + p_z * v.p_z; }
protected:
```

```
private:
  ...
};
```

is very tempting since you can then write `angle = a | b`. However this is a particularly bad idea. While you can redefine just about every operator in C++ (yes, even the comma operator...), the precedence of operators cannot be changed. Thus your dot product just got the precedence of bitwise OR. This is definitely not what you want (why?).

Information hiding: C++ classes allow you to define an object with particular semantics and move all the code associated with implementing this semantics effectively into a module. This really helps with big codes since everything is neatly separated. One should always be as conservative as possible with this to not loose these advantages. Thus our recommendation to hide all data members. It's nobody's business just how a rotation matrix (for example) is represented internally, and no part of your code external to the matrix class should make assumptions about this. For example, a rotation could be stored as a 3x3 matrix, or as a quaternion, or as a set of Euler angles.

Hide unimplementable operators: Sometimes a class is such that certain standard operators don't make semantic sense or are too much of a pain to implement correctly. Consider a `class Tree` with members which are pointers to children of a given node. Building a copy constructor for such an object which does the right thing is extremely tricky. Instead of simply not defining `operator=` and the `X( const X& )` constructor you should create such an operator, put it in the private section, and guard it with an assert:

```
class Tree{
public:
  ...
protected:
private:
  Tree( const Tree& ) { assert( 0 ); }
  Tree operator=( const Tree& ) { assert( 0 ); return *this; }
  ...
};
```

this way nobody external to the class can accidentally call these members and if you do it accidentally inside member function code, you'll get an assertion failure. This is a lot better than having the compiler generate a default assignment operator (bitwise copy) and quietly use it...

Inline and const: Almost every use of preprocessor defined constants or macros is better done in C++ through the use of `inline` and `const`:

```
#define SQR(a) (a)*(a)
// versus
inline float sqr( const float a ) { return a * a; }
// or even better
template <class T>
inline T sqr( const T& t ) { return t * t; }

#define PI 3.14159265358979323846
const double pi = 3.14159265358979323846;
```

The macro `SQR` simply does text substitution. That has the advantage that it works for any data type which has multiplication defined for it. But, you don't get function call semantics. For example, consider `SQR(a*b-c*d)`. The resulting evaluation will compute the expression `a*b-c*d` twice, while the function call will first evaluate the expression and assign the result to the actual parameter. Since `sqr()` is an inline function it will also do the equivalent of text substitution, but with function call semantics, avoiding many of the usual problems with macros. If you want it to work for any suitable data type (like a macro would), you can use the template version of this function.

Similarly, the advantage of replacing preprocessor constants with `const` imbues the constant with semantics such as scope and type that the preprocessor knows nothing about. For example, you can easily do this:

```
const double pi = 3.14159265358979323846;
...
{
  const double pi = 3.14;

  // some computation with the new pi
}
// original definition back in scope
```

(this example is a bit sick, but you get the idea.)