## Guidelines, Usage, Documentation

This file describes (a) design and (b) coding *guidelines* for L++ projects.

Hints about *usage* are included in comments in main.cpp, where examples of various usages are also given. *Documentation*, e.g. operator precedence lists and member functions for selected parts of the STL are in separate appropriately named Rich Text files.

Overarching principle, as only recently realized:

### With the introduction of container classes, references and move semantics, pointers are considered harmful.

### Design Guidelines

1. For a class that allocates memory (or other resource) dynamically, you should typically have:
   1. A default constructor.
   2. A copy constructor, which calls
   3. An operator=(T const&) constructor.
   4. A constructor “For the masses” which allocates the resources and gives them nontrivial values.
   5. A destructor that frees up the resources.

If the class does not allocate such resources, the first 3 are sufficient.

Here is what the methods look like, for the class Lderived, which has Lbase as a base class:

Lderived::Lderived() : Data(**nullptr**), nDim(0) {}

Lderived::Lderived(Lderived1 **const**& inDerived)

: Data(**nullptr**) {

\***this** = inPlot;

}

Lderived& Lderived::**operator**=(Lderived **const**& rhs) {

**if** (**this** != &rhs) {

**delete** Data;

**if** (rhs.Data) Data = **new** Lhist1(\*rhs.Data);

nDim = rhs.nDim;

Lbase::**operator**=(rhs);

}

return \*this;

}

Lfit1::~Lfit1() {

**delete** Data;

}

1. Some things to note:
   1. The copy constructor’s initializer list just touches the pointer variables. While you might think to allocate memory here, the subsequent call to operator= makes that pointless.
   2. The operator= prevents self-assignment. Code that is hosed when self-assigning is an easy mistake, and that is why there is an explicit check against it.
   3. Use delete (if the new operator was used in the construction of the instance), delete[] (for new[]) or free (all others). This is called the RAII (Resource allocation is initialization) design rule; "ownership" of the resource then lies with the created object.
   4. Use the convention that pointers are either nullptr or point to valid data. One of the quirks of C++ is that delete et. al. frees up the memory but does not set the pointer to nullptr. In many cases of course, this is just fine. There are a pair of routines in Ldelete.h for when it is not.
   5. The assignment operator returns \*this as a reference; this allows you chain assignment operations
2. Spent some time thinking about how the equality operator == will work. By default it only exists for built-in types and enumerations; the standard library has it for many things including strings, iterators, complex numbers, valarrays, etc. The compiler will not produce a default version for a class, but it can be overridden for a class or enumeration.
3. If you have both an operator+ and an operator+=, make the former out of the latter and have the code in only one place. Example:

Lclass Lclass::operator+(cmplx const& rhs) const {

Lclass retval(\*this);

return (retval += rhs);

}

Lclass& class::operator+=(cmplx const& rhs) {

// whatever is needed for += operation

return \*this;

}

Similarly for operator- and operator-= etc.

1. In C++ we learn to overload operators. Then we learn to not overload operators. Unless you have an explicitly mathematical object for which things like multiplication or addition are well defined, or you need an operator to implement shallow copying (aka move semantics or address copy) rather than deep copy (aka copy semantics), don't overload an operator other than the assignment operator, which should always be a deep copy, or the << operator to get I/O. If you do need to overload to define a shallow copy operator, try using = operator with move semantics. Really really really do not overload &&, ||, ? or the comma operator, as these have evaluation order semantics; eg, if the left hand side of && evaluates to false, then the right hand side will never be evaluated.
2. Do not define an operation which is addition or multiplication by an intrinsic numeric type. If you

Lbigthing::operator\*(double const& rhs);

then

Lbigthing whonkin;

whonkin \* 3.14159265;

will compile, but

Lbigthing whonkin;

3.14159265 \* whonkin;

will not, because an operator\* isn't left-right symmetric.

1. Declare, construct and initialize objects as close as possible to their point of use.
2. Set null pointers to nullptr, not 0 or NULL.
3. Use inheritance for “is-a” relationships, i.e. when you are defining a class that is a subset of another class. If an object contains an other object, create the class to have a member which is that other object. For example, do not make picture a derived class from pictureframe; declare a frame as a member of picture. With this rule, complicated inheritance diagrams will be rare.
4. Hesitate to override in a derived class a method that is overloaded in the base class without replacing all the signatures. This is because if there is only one overriding signature, it hides *all* the base class methods.
5. Do not make rich interfaces. You need not define every operator or signature you can think of; this only makes the code harder to document and maintain and does not in the end make the class easier to use.
6. Prefer std::string over the C style char array. Remember that c\_str() will return a temporary object, and that setting a char\* to it is setting a pointer to something that won’t be there.
7. Prefer <vector> over <deque> unless there is a reason otherwise. It’s typically a little faster, I hear.
8. Parenthesize arguments to macros to prevent the expanded macro from having unintended order of evaluation. For example,  
   #define SQUARE x\*x followed by SQUARE(y+1) will give y+1\*y+1, which is 2*y* + 1, not (*y*+1)2.
9. Even parenthesization isn't always enough. For example,   
   #define SQUARE (x)\*(x) followed by SQUARE(y++) will give *y*(*y*+1). Did you really want that?
10. Just use typedef, const and function templates instead of #define when you can.
11. Destructors are called when an exception is thrown, so don’t throw exceptions from destructors. If the destructor calls other methods, catch both known and unanticipated exceptions from that code in your destructor. Because the destructor will not be called if the constructor did not finish, it is OK to throw exceptions from constructors if you have freed up all the resources that the constructor has allocated beforehand.
12. In principle, the throwing of an exception is *needed* only when you wish to return flow control by more than one and less than all levels on the call stack. For conditions that go back all levels, i.e. the execution terminates, the exception is inferior to simple crash-and-burn error handling if it does not provide information about where the problem occurred. For conditions that can be handled without returning at all, an error handler is what you want. In practice, for cases where one wants to go back one level, using the exception mechanism is clean but makes the practice of setting breakpoints at the THROW point to unravel what went wrong useless.
13. In any event, strive for code with "rollback semantics" aka "commit semantics", "strong exception safety" or "no-change guarantee": if the method fails and has to throw an exception, the user of the method knows at least that no side effects occurred. E.g. if memory allocation to expand a binary tree fails, at least the binary tree itself remains intact.
14. Catch exceptions by reference, as this will prevent slicing of exceptions that are derived objects.
15. In fact, throwing exceptions usually isn’t a great idea. Better to have an error handling package (which admittedly might be constructed using exceptions) that does what you want, and then either return, exit or continue apace.
16. Create and use setter and corresponding getter methods when values really need to be checked for validity or to be formatted or unpacked for use. Otherwise, just leave the data member public. Although I’ve never done it, you might also want to use setters/getters to LOTO data in a multithreaded environment.
17. Stick with single public inheritance.
18. Don't create setter/getter functions in a derived class that violate the private/public status of members in the base class; that hack should only be perpetrated if you are not allowed to muck with the base class.
19. Do not have the same information in 2 places; if you must, make on version the definitive one and make sure that the 2 copies are synchronized. Similarly, avoid having the same source code in multiple places; that is what subroutines and templates are for.
20. You *must* initialize the following with an initializer list: base classes with no default constructors, reference data members, non-static const data members, or a class type which contains a constant data member. I figure, initialize everything in the initializer list unless you have some reason to do otherwise. Be certain also to initialize all static data members; simple types can be initialized in the header file but arrays or other more complicated static data members have to be defined in the implementation file. Do not initialize one data member (say, x) from another data member (say, y) which is in turn initialized from a constructor argument. The order of initialization is controlled by the order of declaration of the data members in the declaration of the class, and you might not have y initialized when you go to initialize x.
21. Avoid worst-than-linear algorithms when reasonable. The oft-cited Hoare-Knuth maxim “Premature optimization is the root of all evil” applies to optimization that makes the code more complicated or difficult to read, or that takes a long time to implement. Optimization which simplifies is the root of all good.
22. The newish keyword, auto? Evil! It defeats the whole point of C++ being a strongly typed language. Well there is one case at least, where it is needed, to construct an ::iterator of a container of template types. So, if ya hafta, OK.
23. Use the explicit keyword in 1-argument constructors other than the copy constructor (you only need it in the .h file). This will prevent a certain types of subtle bug:

Widget::Widget(int widInt);

void Display(double);

void Display(const Widget&);

Display(5); // Displays a Widget, not 5!

Arguments with default values don't count, ie.

explicit Lprof1(Lbins inXbins, std::string inInfo="");

is a 1-argument constructor.

1. Prefer the C++ static\_cast over the traditional C style casts. If the cast is from a base to a derived class, the traditional C style cast will not create the desired compile-time error if the compiler only has a forward declaration of the derived class on hand when it hits the cast statement. Then the cast will be to an ill-defined class.
2. Of course, whenever you need to have a cast of any type, it is time to re-think things.
3. Don't make static storage objects e.g. global variables that need other static storage objects to complete initialization. There is no way to control which of the 2 statics the compiler will have be created first and so the initialization may not be completable. Google "static initialization order fiasco" to learn more.
4. If you are going to define a parameter in a header file, remember that the definition will appear at global scope in LppMain.h. So instead of something like

#define Nsig 3.5;

write something like

double const Nsig = 3.5;

in the class definition, preferably in private scope.

1. Be cautious in passing parameters to threaded functions by reference. You might want to do this because there is a large data structure and you want different threads to work on it. However, the different threads must either be carefully synchronized, or the threads must LOTO the data structure before changing it, or you will not know anything about the order in which the different threads have modified the structure. If that is too hard, or the data structure is not very big, you might want to pass parameters by value and use a async / future arrangement.

### Coding Guidelines

1. Comments in header files should describe the inputs, outputs, preconditions, postconditions, machine states, general performance goals and functionality of methods and the meaning of data fields. More detailed information, e.g. which specific version of an algorithm was used in the implementation, citations of specific papers etc. go in the comments for the implementation file. Feel free to send the reader to the header file of the base class.
2. Use #pragma once to implement your include guards.
3. As a general rule, follow the comments and styles of earlier code.
4. The sequence for including headers is:
   1. LppGlobal.h etc.
   2. stdlib headers that are not already in the above
   3. Headers from other L++ classes that are needed in this one.

These go in the header file; the implementation file *usually* only contains a single include, which is the name of the corresponding header file.

1. Do not place using namespace statements in header files. In source files, place them after all #include statements.
2. Explicitly include public:, protected: and private: in the class declaration; put them in that order and leave out any of the three that do not have any corresponding members.
3. Do not use struct when creating a class with method members, or class for structures without method members.
4. int i = 1; is better than int i; i=1;
5. Use 4 spaces for tabs, and try to stay within 80 columns (but do not feel very tightly bound to that rule). You need not write the full curly bracket structure e.g.

if (i==1) {

(\*RA)(i) = (\*inLvec.RA)(i);

}

if in fact it will all fit on the same line:

if (i==1) (\*RA)(i) = (\*inLvec.RA)(i);

1. C and C++ have the convention that variable names beginning with 2 underscores, or an underscore followed by a capital letter are reserved for the compiler / standard library writers. Avoid the use of underscores at the front of a variable name; you may put it at the end of the name of a private variable.
2. Use int rather than addr = size\_t unless the variable is clearly an address sort of a thing rather than an integer; do not go for addr over int just because the integer would never meaningfully be negative. For example, a number of degrees of freedom in a fit is an int, not an addr. Indexing in an ra<T> is with ints because sometimes the index is negative or zero. size() operators, as in stdlib containers, return addrs.
3. Use ++i rather than i++. In some compilers you save making another object. Not that this really matters to run time.
4. Use explicit return statements, except for methods where there is only a *very* short stretch of statements with trivial flow control.
5. Don’t call by value-const. Func(const int i) and Func(int i) are one & the same. I think that Func(const int& i) and Func(int& i) are not?
6. Place a break after each fork in a switch statement, or a comment to point out that you have not done so for some reason.
7. Do not omit the argument names in method declarations. You can get double stirling(int); to compile, but you shouldn't.
8. The standard C headers (e.g. stdio.h) also appear in C++ with 'c' as a prefix and the .h filetype gone, (e.g. cstdio). Include the latter form rather than the former; the former create definitions that appear in the global namespace and the latter create definitions only in the std namespace.
9. Use container.empty() rather than container.size()==0 – for lists and some other containers it can be a lot faster.
10. Send error messages to cout rather than cerr. Because you ain't bothering to look at cerr, right? Right.
11. Use std::endl rather than "\n". Because, god forbid, maybe someday you will need to use a Windows machine. But be lax on this rule when using C-style methods like printf.
12. If you need to use a namespace qualifier, use the shortest one possible. So if you are in e.g. gar::cheat and you need something in gar::reco write reco::something, not gar::reco::something.

***23)***