TEN STEPS TO PROGRAMMING MASTERY

CHRIS LOMONT

1. The Big Rules

Here are ten ways you can improve your coding. The overriding principle to improving your skill at coding, as well as almost endeavor, is ***open your mind and then fill it with better knowledge.*** Improvement necessarily implies change, yet it is human nature to fear and resist change. But overcoming that fear and embracing change as a way of life will enable you to reach new levels of achievement. Applying this basic precept to software development leads to the following ten rules, designed to challenge you and to give you some concrete steps to take to reach geek guru. The cost is time, humility, effort, and desire. The payback is job security, better pay, happiness, adoring fans, and eternal life. So let's stop screwing around and get started.

Big Rule 1: Break your own habits.

When you began coding, you were much less experienced than you are now, and you probably picked up bad habits out of ignorance. Now is the time to change these habits, but they are difficult to find, and embarrassing to recognize. The best ways to locate weaknesses in your programming ability are to read others code and think about it, discuss your code with others to get new viewpoints, and to read, read, read!

So if you want better habits, your question is "Where do I find habits?". There are many sources: the books listed at the end of this paper, online forums, trade journals, local in house gurus. Ask people questions. Read, test, think. But most of all think carefully through each item and do not throw out ideas out of reflex.

Have some humility. Your brain is limited, so do not assume you can hold everything in it. It is far better to know where to get an answer to 1000 questions than to know the answer to 5. But to know where different answers lie you must read, learn, and be curious.

Ego and pride get in the way of becoming great very often. Do not become stubborn or afraid people will discover you do not know topic X. Because if you do not know topic X, and never ask, you may never learn topic X, which is the original goal: improving your skill. So use ego and pride to drive you, not blind you.

Your own habits are the biggest barrier to change in a better direction. So form habits right the first time when possible, and go back and reshape them later when they are holding you back.

Big Rule 2: Apply defensive programming.

Your code will constantly be under assault, from idiot users, poor hardware, buggy drivers, incompetent maintenance programmers, cosmic rays, and general bit-rot. It will be pushed to the boundaries of its design, passed unholy parameters, and forced to run on systems unforseen at the time of its creation. So design your

Date: July 22, 2003.

functions and interfaces appropriately, expecting Murphy's Law to be elevated to a Law of Thermodynamics. Now, for some specifics:

The single best way to catch such errors is to place `assert(..)`s liberally throughout your code. Anytime there is some fact that should be true at a point in your code, whether it is a pointer being non `NULL` or an integer being in a certain range, `assert` that this is true. You should `assert` any assumptions a function needs in order to operate, and `assert` any things that should be true at the function end. People passing your function bad values while debugging will trigger the first `assert`, and people making errors while modifying function internals will trigger the latter `assert`. Now that you've validated items coming into your function, and items leaving your function, all you have to focus on is the 5-20 lines in your function, which is much easier than worrying about all the external code. This simple habit will catch an awful lot of bugs and coding errors by those using your code, speeding up development and your life as a programmer much better.

Also, on any interface that could conceivably be used by others, if you can spare the performance (and you can, admit it), check all parameters coming in for validity, even in release builds. Those cryptic comments you placed in a header, warning of that one weird case, will be subsequently ignored by developers using your code. Make your code as idiot proof as possible, since idiots will eventually use all the code you write. The Arianne 5 rocket, a half BILLION dollar rocket, blew up from a simple programming error [1] Many more such costly errors have and will continue to happen. Don't cost your employer a half a billion dollars, since that will require a lot of overtime to pay back.

Make comments and error messages useful to others besides you. In `assert`, it is useful to have it print out the function name and filename and useful text explaining what happened, or even that gives a stack trace and diagnostic data file. One way to do this is to use a more powerful assert than the standard `#include<cassert>` method. Check online articles for finding such an assert, there are many versions and ways to approach a "smart assert".

Another defensive method is to make every `switch` statement have a `default:` case, that `assert(0);` in debug mode, to ensure that when someone adds another type to the `enum` that the `switch` switches on, that it gets updated in the many places where it is used. Also, you will catch unexpected cases you may have missed, etc. So if you think there is no way for this `switch` to have more then the 7 cases you coded, then there probably is, and the default `default:` will help you catch it.

Check return types from API's religiously. When you are in the zone it is fun and easy to ignore this. However, tracking down hard to find bugs due to this will break your zone later anyways, so it is better to engineer well upfront than to patch shoddy workmanship. At the very least, in debug version, log any API failing, and perhaps even in release mode, so then when something crashes you can view the log and see what sequence of unexpected items failed. And most importantly is to add this code every time you program, otherwise it is way too hard and boring to go back and add all these tests.

[1] On June 4, 1996 the unmanned Ariane 5 rocket, launched by the European Space Agency, exploded just 40 seconds after lift-off from Kourou, French Guiana, after a $7 billion, ten year development. The explosion was a $500 million loss. The failure was a *software error* in the inertial reference system. A 64 bit floating point dealing with velocity of the rocket was converted to a 16 bit signed integer. Type safety has its value.... see http://www.ima.umn.edu/~arnold/disasters/disasters.html

Learn the power and pitfalls of C++ exception handling. This is a difficult and deep subject, but will result in much more robust code. It is very important to learn, since many functions in the standard library throw exceptions instead of using return codes, so to write code that doesn't `terminate()`, you must handle these exceptions. See the section on exception handling 11 for more details and references.

Finally, understand an error when it occurs. Do NOT patch someone else's code unless you are sure you understand why an error is happening. Do not "special case" the error out because you cannot understand why it is happening. The computer is (mostly) deterministic, and you can isolate exactly what causes the bug, at which point you can fix it. Anything less is like a doctor taking your temperature to diagnose a cut on your foot, and then amputating your arm to see if it stops your foot from bleeding.

Now that you are prepared to make your programming as defensive as possible (as opposed to your old, offensive coding methods), let's move on.

Big Rule 3: Don't be so macho.

Macho went out with the Village People. Just because you know all the inline tricks, routinely use the intricacies of operator precedence, prefer function pointers to functions, and have mastered language subtleties, does not mean you should use them in a masochistic mental ritual. Sure it looks neat, impresses colleagues, and is a fun intellectual exercise, but the end result is code that is so hard to understand that people will either break it during maintenance, avoid it from confusion, or just scrap it and waste time and money rewriting it. Program as if your code was written to be read by programmers several levels beneath your level, and you will be praised for the beauty of your code. You will also find your code is easier for you to read and understand next year, and that it is easier to debug and make flawless. For example, what does the following do?

```
a + b | c || d % 3 && e == f
```

Let's put in nice parentheses:

```
((a + b) | c) || ((d % 3) && (e == f))
```

It still looks like a nonsense operation, but now it is much easier to parse visually, perfectly illustrating the need to avoid showing off knowledge when there is a clear solution. "Everything should be as simple as possible, but no simpler" - Albert Einstein. Save your hard earned tricks for the *Obfuscated C Contest*[2], where you can write code like the following[3] (notice the nice recursive use of calling main):

```
#include <stdio.h>
int O,o,i;char*I="";main(l){O&=l&1?
*I:~*I,*I++||(l=2*getchar(),i+=O>8
?o:O?0:o+1,o=O>9,O=-1,I="t8B~pq‘",l
>0)?main(l/2):printf("%d\n",--i);}
```

[2] www.ioccc.org/

[3] Best Small program, 1998, bas2.c

Don't continue to work when your brain is too tired. The time spent making poor code will be wasted in fixing it later. Some people think it is cool to code way past the point they are making useful, correct code. At this point stop working on production code, and if you really need something to code, play with some ideas and learn some things. But a tired brain trying to make good production code will lead to errors.

Also, realize your brain is small. Do not try to hold the entire design in your brain at once - partition it, and make each piece work properly. It is easier to focus on the one piece you need at the moment and get it right, than to try to write many unrelated functions at once. Since programming is a purely mental exercise, work in a place you can be free from distractions for an hour or more at a time, preferably in a place you will never get distracted. Take short breaks sometimes to sit back, stretch, read a page from a book, or walk a lap around the area, whenever you are losing "the zone."

Now that you are open minded, and learned to program defensively, and stopped being such a tough guy (or girl!), ...

Big Rule 4: Use your tools wisely.

Use all the tools that are available to you. Spend time to learn them. Smart people spent a lot of effort designing and building your tools, and they are designed to speed up your development, so use them. At least be sure to find out what is there, and how to use them - if they turn out to be crap, you can ignore them. But if they have nifty feature Z, and your current programming would really, really be better with a feature Z, and you do not realize it is already there, then you will either reinvent feature Z or, more likely, you will do without, plugging away in tears in your dark cubicle. So occasionally click on the option you do not understand, and learn a new feature. In particular, the following tools are indispensable for modern C++ development:

Debugger - this is your best friend (after your dog if you are male, and after diamonds if you are female...). Debugging during the Jurassic Period consisted of using print statements littered throughout code to try to find bugs. Debugging during Creation consisted of guessing and rewriting. Many current people still use these techniques, even though most toolsets have much more advanced tools. Be sure to know how to use your debugger well. For example does it support source level debugging? Basic and advanced types of breakpoints? Conditional and counting breakpoints? Break on memory changes? Advanced structure viewing? Data logging? Forwards and backwards stepping? Function jumping? Edit and compile? Memory views? Disassembly? What debugging services does your OS provide? Your C Runtime Library? Your Memory manager? Does it catch stack corruption? Can you debug multiple threads? Does it catch using uninitialized variables? Know these things, and how to apply them, and your debugging time will be much more fun, productive, and less full of black magic.

Profiler - *use this before trying to optimize your code!* Be sure to profile your code well with a profiler that gives function and line timing at the very least. Many hours have been spent on optimizing the wrong section of code, when a few minutes time spent up front would have had the same result. Good profiling, followed by intense thought, followed by minimal coding, has resulted in some very, very large program speedups. However, neanderthals still consistently do this as: large time spent coding, followed by minimal time spent thinking, followed by reading a profiler

manual, followed by the original suggested method. So save the extra time and do it right the first time. Even if you do not have a profiler, insert some timing functions (that you remember to remove if necessary!) into your code, and get hard data before spending effort optimizing.

Web resources - look up stuff on the web that you need to use or understand. Never before has there been such an easy way to find programming information of excellent quality (along with the so-so quality stuff, the borderline insane stuff, and the tabloid stuff). Be sure what you are reading is correct, since everyone and his pet gimp have posted coding ideas on the web (which is where I will post this!).

Automatic checkers - if you have them (and many compiler tools and libraries have them buried in the documentation). There are often built in tools to find memory leaks, there is often a C Runtime debug mode, an OS debug mode, and library debug modes, which validate their inputs and provide information on errors you make. There are automatic buffer overrun checking runtime libraries to catch your errors on array sizes. Basically vendors shipping the items you build upon use the `assert` tricks in a bit more sophisticated manner to catch inconsistencies in your code as well as theirs. Use these tools while debugging.

If the only tool you have is a tack hammer, all problems look and taste like carpet tacks. So expand your toolset every week, and soon you will build more sophisticated items.

Big Rule 5: Learn your language.

C++ has a lot of areas, some of which truly are esoteric, but there are many useful features programmers often do not understand. For example, know that `vector<T>` should be used instead of an array of `T`, almost without exception. The `vector<T>` is type safe, grows when you need it, can be passed to C style functions needing `T* param` parameters, and requires no `delete[]` when you are done with it. Also, instead of using `char * str` to store a string, use the template `string`, which has a lot of useful member functions for searching, pattern replacing, and more. Plus, adding the Boost library [5] gives a regular expression engine for free. For example, you should never use

```
int * array = new int[100];
\\ your code ....
Func(array);
delete[] array; // don't forget to free me!
```

when you can use the much safer and cleaner

```
vector<int> array(100);
\\ your code ....
Func(&array[0]); // note the usage to pass to C-style functions
                 // no deleting necessary - exception safe!
```

Other built in containers are `deque`, `list`, `map`, `queue`, `set`, `stack`, `vector`. Most compilers also have a *nonstandard* `hash_set` and `hash_map` for very fast lookups, but these are expected to be added to the next standard, C++0X. So if you spent (=wasted) time recently implementing a linked-list, or a stack, or a hash table, it was time wasted, since these are already built into C++.

The `algorithms` header contains algorithms to operate on these containers (rap that to a funky beat!), standard C arrays, and any container you can construct. There are `binary_search`es, `count_if` to count items matching a predicate, `for_each` to loop nicely and quickly, `max` and `min` to find max and min items in a container, partial sorts, permutations, `sort`s, `reverse`, unions, `unique` to create a container containg a unique copy of each item, intersections, complements, and many, many more. This is the Swiss Army Knife of algorithms, so before you write a basic algorithm, see if it exists in header `algorithm`. Using them will save time, and are likely to be less buggy and have faster execution than if you develop them from scratch. So learn the STL - it is well worth it!

For example, which is a faster sort for integers:

```
vector<int> vals; // an array of integers...
// fill in some values here ....
sort(vals.begin(),vals.end()); // sort them
```

or the old C standby `qsort`

```
// function to compare integers, needed for qsort...
int IntCompare(const void * e1, const void * e2)
    {
    if (*(int*)e1 < *(int*)e2)
        return -1;
    if (*(int*)e1 > *(int*)e2)
        return 1;
    return 0;
    } // IntCompare

// then somewhere you need...
int * data;
// allocate memory....
// fill in some values here....
qsort(data,size of data...,sizeof(data[0]),intcomp);
// DO NOT FORGET TO RELEASE THE MEMORY!
```

Well, on surprisingly, on a simple test (VC++.NET 2003), the qsort method takes 1.4 times as long to execute as the STL `sort` algorithm on the STL container `vector<int>`. The reason is that `sort` is a template, and expands to an optimized sort for whatever type it operates on, unlike the `qsort` routine, which requires weird function pointers and `sizeof` operators. Plus, the first method is a lot shorter to type and clearer to read, hence is much more likely to be error free.

The moral is: the STL has many features and functions that will make you more productive and your code more nearly correct. For many more features that are worth learning, see item 14.48. There is a lot to learn to really get the most out of the language.

Big Rule 6: Learn common errors and avoid them.

There are a few errors so common, that they account for over 90% of all errors at the programming level, so it is wise to learn them. When they are about to rear an ugly pointer, a little extra diligence will prevent the problem. “A gram of

prevention is worth a kilogram of cure." Everyone nowadays is taught that GOTOs are poor programming practice, since they lead to "spaghetti" code. This should be expanded to the following for future generations:

GOTOs are evil Pointers are evil Macros are evil Arrays are evil

Burn this into your brain with a soldering iron. Why? They all lead to hard to use, read, debug, and maintain code. They add bugs. They are each replaceable with a better construct 99% of the time. Avoid all of them as much as possible. Of course, you currently love using pointers and unsafe arrays, but programmers in the 1960's loved GOTOs, and you do fine without them (the GOTOs, not the programmers). So be a man (in this case, even if you are already a woman), buckle up, and break yourself of these bad habits.

As another example, if you mistype `atoi(str)`, the compiler will catch it most of the time. If, by a misunderstanding, you type `for(int i = 0; i < 10; i++)` when you should have typed `i < 11`, you will not be warned by the compiler and the program will jack your hard drive. If you realize this type of error is common, you will be more careful on the `for` statement than the `atoi` statement. Since your brain energy is finite, you should choose wisely where to spend it. Here is the list of the most common errors in order of occurrences:

(6a) **Avoid pointer errors** - pointers are inherently bad bad bad! It is likely future generations will be taught pointers are bad, like gotos. Almost all uses of pointers in C++ code are leftover from the good ol' C days, and some mildy successful modern languages like Java and C# get along fine without them. In most cases, places you are currently using a pointer, you should use a reference, which is much safer. So take a hint, don't be a dinosaur, and try to avoid pointer altogether.

In the cases you cannot avoid one, be extra careful in how you use it. Pay attention to `delete` data that you `new`ed. AND `delete []` arrays that you allocated. Pay special attention to pointer arithmetic, since it will let you trash your memory, data, and executable code in many cases. Use `const` all over your pointer code, as much as possible - see section 9. Use the debug builds of your C runtime, your OS, and your libraries, while developing, to help catch pointer errors. Finally, learn to use `vector<T>` instead.

(6b) **Avoid "off by one" errors** - the next most common error is the "off by one" error, when you misunderstand, mistype, or generally screw up on the bounds of a loop or array reference. Remember many errors occur here, so when trying to decide if you need `i < 10` or `i < 11`, pay special attention, fire up an extra neuron or two, and get those bounds right the first time. You can rest that neuron later when typing an `#include<...>`, which is pretty hard to screw up.

(6c) **Avoid complexity** - your brain is finite. You can probably only hold five to twenty important items in it at once while making coding decisions, so avoid complex functions where you need to remember twenty-one items simultaneously. The phone rings. You overlook item seventeen. You introduce a bug that takes you six hours to fix three months from now. Welcome to software development.....

So to avoid this scenario (or repeat again), break down functions to smaller functions, each function doing a much smaller task, and glue them

together to replace original 95 line function you had planned. You will save a lot of headaches if you can decompose problems to such fine granularity, and always remember to keep the current task as simple as possible.

(6d) **Avoid interruptions** - to get in the zone, you must have time to concentrate. Programming is a purely mental exercise, and if you are in an environment or pattern where you are interrupted every 5 or 10 minutes, you will never get the level of concentration necessary to write good, clean code. So try to isolate yourself from interruption for sufficient blocks of time to write decent pieces of code.

If you can avoid these few mistakes by noting they are common, you will reduce the number of bugs in your code, thus reducing your debugging time, thus having more time to code, thus..... You get the idea.

Big Rule 7: Pull each other up.

Everyone in the group, from neophyte programmer, to tenderfoot, to journeyman, to adept, to unkempt old-timer, has knowledge that others do not. Learn from each other. Ask questions, have people look over pieces of your code, ask for suggestions, and stimulate and push each other to improve in your mastery of the language, tools, and environment. You are not smart enough to invent all the useful techniques a good programmer should know. No one is that smart, but the sum total of all the knowledge of all the smart people that came before you is available to learn in a fraction of the time it took to invent, so learn some. Find good books on others bookshelves. Read newsgroups. But doing it as a group helps all, since the fewer bugs any one member of the group introduces into the code base, the less time the entire group has to spend fixing things.

The bottom line is that is is a lot quicker to learn something from someone who knows it already than to read it yourself, and it is exponentially quicker to learn from others mistakes than to reinvent all those errors yourself.

Big Rule 8: Do not reinvent the wheel.

It is fun and instructive to write that quicksort, that red-black tree, and that MD5 hashing function for yourself, but you should not be reimplementing algorithms and code that is built into your tools, or is easily obtainable. In the same manner you do not build a hacksaw each time you need to remove handcuffs (well,... sorta like that), you should not recreate others work. One reason is that well tested, banged upon pieces of code are much less likely to hide subtle errors than your 20 minute afternoon attempt at a binary search. Another reason is that often libraries have been built by people that have invested significant mental energy and time to optimize them, and you should build upon that effort.

For example, sit down right now and write detailed pseudo-code for a binary search routine that takes in a list of nondecreasing integers, a length, a value to find, and returns the index of the value if it is in the list, or a "does not exist" if the number is not in the list. How long does it take? And most importantly, is it correct? Does it handle endpoints correctly?, lists of length 1? length 0? So you see how long it would take each time to code it, when you can use the STL `binary_search` routine and save a lot of time and headaches.

On the other hand, it is wise to rebuild items sometimes, since the knowledge gained helps you grow and be able to solve more difficult problems. But this is best done only in play code, not in production code. Save the well designed, off the shelf routines for production use. Or, if you want to try your hand at routine X,

code yours and use the off-the-shelf one, and in the debug version ONLY of your project, run both algorithms and compare the time used and the output. It will quite often be instructive.

Finally, using reason to use pieces of code is it saves time! The whole point of code reuse is to design well written code so it can be dropped into future work, without having to rewrite every project from scratch. Battle Tested^{TM} code, when inserted correctly into new projects, have a much lower defect rate than new code. Hence the adjective "Battle Tested".

If you want to reinvent the wheel, do it on your own dime, mister.

Big Rule 9: Code clearly.

The main purpose of code is not for telling machines what to do, which sounds ridiculous on the surface. Source code is a language for communicating with humans what you *want* a machine to do. Your source code is a book, a novel, a piece of art, that others must read and understand, work on, extend, and utilize. Although you do not have the poetic license that a poet does, slinging symbols all over a blank page, you are creating literature, since people will read your code and understand it. So program accordingly. Unfortunately, since the very literal computer must also understand what you write, you must cross your t's and dot your i's (and close your braces, and free your resources, and a million other "grammatical" rules). The bottom line is that you must find a delicate balance between making your code correct, and making it very readable and understandable.

So we agree that it is important to make your code easy to read. Clearly show the intent of routines, objects, files, libraries, parameters, data flow, assumptions, preconditions, et cetera ad infinitum. Polishing off your masterpiece with consistent spacing and concise precise commenting make your code a gem.

Big Rule 10: Continue learning.

The most important, lasting area of knowledge you can improve is your knowledge of algorithms. Algorithms are the building blocks, and dictate how you will design code, how you will solve problems, and how effectively you can change an approach to a problem. You know bubble sort, and quick sort, and perhaps radix sorts, and perhaps a dozen more sorts, and thus are armed to approach many sorting problems. It is not necessary to master the details of every algorithm on the first pass, but it is useful to know what types of algorithms exist, limits to algorithms (you cannot do linear time comparison sorts, for example), so that you can look up the details when necessary. What is a good string matching algorithm? What is a the fastest way to multiply matrices? Read trade mags. Get a good introductory textbook, such as [35, 36]; after mastering an introductory text, master an advanced text like [10], or the algorithm bible "The Art Of Computer Programming," by Knuth, [21, 22, 23]. All of these books are very good, and are good books to have on your shelf while programming.

The second most important learning area is how to design medium and large scale systems [26]. A 100 line program is easy to do - place the lines in almost any order it runs. 1000 lines is also not too hard - you can still understand the whole program, and probably stomp out all the bugs. 10,000 lines takes a little longer to write, takes time to debug, and is still probably not flawless. But if you want to handle 100,000 line programs, or perhaps 1,000,000 line programs, a very very large part of the success will of the system be in the design that was done before any code was written. It is not unlike building a jigsaw puzzle. You can

probably complete a 30 piece puzzle in a minute, but it is doubtful you can do a 3000 piece puzzle in 100 minutes. The complexity of such tasks is rarely linear in input size. If you want to maintain such large programs, you should learn design principles with mysterious phrases like "refactoring," "patterns," [16], and "Unified Modelling Language." But, as you progress to being responsible for large systems, these are things you should learn.

The third most important thing to learn is your language and tools. These are the blocks you will use to build your castle. If it comes time to build a wall, but the only block you know is a window, your castle might be pretty but it won't withstand attack. So learn all the types of blocks available and how to use them effectively. This area is covered in so many places in this paper, I will only mention to know your language, your compiler, and your debugger, as well as how they interact. Continue to learn about these items as you use them - make it a point to learn a new thing each day when you start, and at least once a week after you become comfortable with them.

Last most important: sleeping. Just kidding. Sleep is necessary for your brain to form memories. Sleep enough, and drink your milk.

Finally, if you want more information about where to learn development in general and C++ development specifically, here are some resources:

(10a) Read the Usenet newsgroups `comp.lang.c++`, `comp.lang.c++.moderated`, `comp.std.c++`, and also `microsoft.public.vc.stl`, as well as others you can find.

(10b) Browse PC-Lint "Bug of the Month" ads are an excellent test of your C/C++ skills, and are archived at www.gimpel.com/html/bugs.htm.

(10c) Learn exception handling, starting with section 11, and the references there.

(10d) Practice auto documenting code while making comments. Use Doxygen for example, www.doxygen.org, and see section 8 on commenting.

(10e) Read books. Some that are are excellent, conveniently placed in the order you should read them (or at least buy and skim them) are:

- Scott Meyers trilogy "Effective C++," "More Effective C++," and "Effective STL" [33, 32, 34] are a brilliant exposition of using C++ properly.
- Steve McConnell's "Code Complete" [31] is probably the best book written on programming. Read it or wither and die.
- The Stroustrup C/C++ user manual is essential [38] as a reference book.
- The Gang of Four, as they are called, is Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. They wrote "Design Patterns" [16], a good and famous book on, you guessed it, patterns, those recurring problems and solutions in software development, and ways to abstract them to improve reusibility.
- Andrei Alexandrescu's "Modern C++ Design" [2] is a difficult book, but is the cutting edge of what you can do with templates to make very powerful code. Once you master this book, you will be a top notch C++ programmer.

A final note, if you have not read Steve McConnell's book [31], you should. It will cover a lot of good advice, backed up by lots of industry experience, data, and studies, on more topics than you can imagine. It is very good.

And with that, gentle reader, we have reached our goal of covering the ***Ten Steps to Programming Mastery***! Follow them diligently, and you too can be the envy of your peers :)

And now it is time for a.....

Pop quiz: You have functions `f`, `g`, and `h`, and expressions `expr1` and `expr2`. In what order do these 5 items get evaluated in the following line of code:

```
f( g( expr1 ), h( expr2 ) );
```

I think this is from a Herb Sutter article, but could not find it. The answer appears at the end of this paper.

The rest of this paper contains a lot of sections organized around various principles, and contains guidelines for programming. Although this paper is mainly for C++ development, many of these techniques apply to any language. Since there is not space to explain each point in depth, take them as truth (or a grain of salt if you prefer, but ponder the reason for or against, and you will find them to be good). There are ample references explaining reasons for most, if not all, of these points.

Section 2 is about pointers, their weaknesses, and rules for good usage. Section 3 is about namespaces, a useful way to manage complexity. Section 4 is about naming of functions, variables, classes, etc. Section 5 is about usage of variables. Section 6 is about good function design. Section 7 has formatting guidelines to make your code readable and easily usable. Section 8 contains tips on commenting. Section 9 explains how to use `const` correctly, profusely, and wisely. Section 10 explains good class layout and rules. There is not enough space to write all there is to know about class design, sadly. Section 11 is a much needed list of guidelines for exception handling. Section 12 has a few general guidelines on improving code speed. There are 10^{15} more tricks of the trade not listed here, but most are so specific as to be inapplicable to many situations. Section 13 contains portability tips, that dark and overlooked area of programming. After all, "if my compiler compiles the code, it must be correct." Section 14 has general guidelines. Section 15 has some recommended practices. Finally, section 16 has resources of where to get more information, and more in depth explanation of the many items listed in this paper.

2. Resources and pointers

The ***biggest*** source of errors in C++ programming is pointer errors. Using smart pointers (search Google, and read relevant items) is the best way to avoid such errors. Smart pointers manage deleting an object when it is no longer needed, can keep track of how many items are using a resource, and can ensure type safety, among other things. The C++ standard has a basic smart pointer, `auto_ptr`, built in, but they can introduce hard to find errors as explained below. Very good references to understand this crucial area are Scott Meyers' books [32, 33]. To understand `auto_ptr`s, start with [32, item 28], then look at the smart pointer implementations at Boost [5] to see a better smart pointer. For using `auto_ptr` effectively, see Herb Sutter's [40], and see [43] for a comparison of smart pointers.

Finally, the very best, but hardest to fully understand, explanation and implementation of smart pointers is in the book [2] which describes the library Loki [30].

A quick introduction to using `auto_ptr` follows. To begin with, the following common practice is unsafe:

```
// Example 1 : Exception unsafe
void Func(void)
    {
    T * pt( new T );

    /*...your code...*/

    delete pt;
    } // Func
```

If an exception is thrown in your code, the `delete` is never called, leaking memory. The correct exception safe way to do this is

```
// Example 2: Exception safe
void Func(void)
    {
    auto_ptr<T> pt( new T );

    /*...more code...*/

    } // Func
```

Note that you do not need to remember to free the data, since the `auto_ptr` going out of scope calls the destructor automagically. Thus exceptions release the data correctly.

To avoid errors, the rule to remember when using `auto_ptr` is: **COPIES ARE NOT EQUIVALENT**. When you make a copy of an `auto_ptr`, the copy *owns* the object, and the original is set to `NULL`. Otherwise both `auto_ptr` will call the destructor, an error. Do not use `auto_ptr`s in standard containers, since they *will* get corrupted.

Another good use is in classes that have a pointer to an object. Replace

```
// Example 3 :
class Foo
    {
private:
    T * ptr_;

public:

    /*...your code...*/
    } // Foo
```

with

```
// Example 3 :
class Foo
    {
private:
```

```
    auto_ptr<T> ptr_;

public:

    /*...your code...*/
    } // Foo
```

Then you do not have to release it in your destructor, although you still need a correct copy constructor, since **COPIES ARE NOT EQUIVALENT**. Understanding this, here are the general guidelines:

(2.1) Prefer `new` and `delete` to `malloc` and `free` [33, item 3]. Some even say "Do not use `malloc`, `realloc` or `free`".

(2.2) Use proper form of `new` and `delete` [33, item 5]. *Always* use `delete[] ...` when deallocating arrays.

(2.3) "Do not allocate memory and expect that someone else will deallocate it later." [15, Rec. 58]. Use an object handle or `auto_ptr` that deallocates itself if the user forgets. This does not apply if creating a replacement `new`.

(2.4) Memory allocation need not return `NULL`, but may throw an exception [33, item 7]. According to the 1997 ANSI C++ standard, `new` should throw an exception when out of memory; it should not return `NULL` or zero (but note that many compilers still return NULL or 0, promoting bad programming). See section 11 for more information.

(2.5) "Always assign a new value to a pointer that points to deallocated memory." [15, Rec. 59]. Preferably set this pointer to `NULL`. This will prevent many errors, and will also make the program more secure.

(2.6) Use destructors to prevent resource leaks [32, item 9]. Use `auto_ptr` or smart pointers in the template libraries Loki [30] and Boost [5].

(2.7) Pointers to pointers should be avoided whenever possible. If pointers are dangerous, pointers to pointers are sure to kill you.

(2.8) Avoid pointer arithmetic. This leads to lockjaw and night-vision. Besides, it is error prone.

(2.9) Pointers in classes cause copy problems. STL containers (usually) do not handle this correctly, since they will not call destructors properly.

(2.10) Never convert objects of a derived class to objects of a virtual base class with a C style cast. This causes a *slicing error*. Again, search Google for "C++ slicing error" to understand better. See item 14.28.

(2.11) Do not write logical expressions of the type `if(test)` or `if(!test)` when test is a pointer. Compare pointers to `NULL`, since some compilers may not define `NULL` pointers as `0`. Also, address 0 may be a valid address in some architectures, so `if (test)` could fail even though `test` is a valid address.

3. Namespaces

Namespaces are a great way to separate libraries and different areas for a project to avoid naming conflicts. They allow you to avoid the ugly prefixes like "gr" before all graphic functions, etc.

To create a namespace, simply

```
namespace FunkyLib
    {
    /*  stuff you want in the namespace */
```

```
    }
```

Now all items defined between the `{` and `}` are prefixed by the compiler with a `FunkyLib::`. This is useful in header files to compartmentalize code. Here are some ways they can be used:

```
using namespace std; // make all items in std
                     // namespace available to file

namespace Short = A::B::C; // can set equivalent names

void Function1(void)
    {
    using namespace Graph; // makes all symbols in Graph available
    Short::Func();         // This is the same as....
    A::B::C::Func();
    // ... your code ...
    }

void Function2(void)
    {
    using namespace Graph::Bitmap; // only Graph::Bitmap available
                                   // and only in this function
    // ... your code ...

    Different::Bitmap bmp;         // use another namespace version
                                   // of Bitmap
    }
```

Thus `namespace`s give you a decent way to partition the global namespace, yet allows your code locally to avoid having an ugly prefix on every symbol from library "xx". Let's dance:

(3.1) Partition the global `namespace` [33, item 28].

(3.2) Do not put `using namespace...` in header files. If you do, any file including the header will bring all items in the namespace, which is undesirable. To refer to namespace items, explicitly identify them. For example:

In header foo.h:

```
#include <vector>

// to access the vector, avoid the "using"
// and name it directly
float AverageValue(const std::vector<float> & data);
```

In file foo.cpp:

```
#include <vector>

#include "foo.h"

// use this if there are otherwise a lot
// of std::'s floating around your file
```

```
// Good porgramming prevents std's
using namespace std;

float AverageValue(const vector<float> & data)
    {
    /* code here */
    }
```

(3.3) `namespace`s can be nested, but don't abuse it.
(3.4) Place functions and variables local to a file in an unnamed `namespace`. See 5.11 for details.

4. Naming Rules

The overriding rule is to make code that is readable, so choose names that explain the purpose of what is going on. Clarity supersedes brevity. See "Code Complete" [31, Chap. 9, esp. Sec. 9.2] for a very good discussion of naming.

For example: if every time you did arithmetic, you had to pick new symbols for the digits, you could still do arithmetic, but the effort would be much larger, and checking your work would be error prone. By agreeing on what symbols mean, it becomes second nature how to use them, as in arithmetic. So pick names in programming in a consistent manner, to assist understand code quickly and accurately. Some useful, tested naming guidelines:

(4.1) In names which consist of more than one word, the words are written together and each word that follows the first is begun with an uppercase letter.
(4.2) All names are of the form `ThisIsAnIdentifier` and not of the lowercase form `this_is_an_identifier`. However, the C++ standard libraries are tending to lean to the latter, so choose one method and stick with it. The first method has the advantage that it fits well with the rest of the conventions in this section.
(4.3) Do not use typenames that differ only by the use of uppercase and lowercase letters. This leads to confusion and chaos.
(4.4) Names should not include abbreviations that are not generally accepted. Your dogs initials are not a valid class name.
(4.5) When making a library, enclose all the identifiers of every globally visible class, enumeration type, type definition, function, constant, and variable in a `namespace` (see section 3). If this is not possible, begin each identifier with a 2 letter prefix that is unique for the library. For example, `GraphicLib::NewSegment()` could be the name of a function in the graphics library with `namespace GraphicLib`, or `grNewSegment()` without namespaces.
(4.6) Write code in a way that makes it easy to change the prefix for global identifiers. `namespace`s make this very easy.
(4.7) Names of abstract data types, `class`es, `struct`ures, `typedef`s, `enum`erated types, and `namespace`s are to begin with an uppercase letter. Example `ThisIsAClass`, `BigNumber`.
(4.8) Functions begin with a capital letter. `SomeFunc();`
(4.9) Variables start with a lowercase letter: `largeValue`

(4.10) `class` and `struct` member variables end with a trailing underscore. Example: `memberVariable_`.
(4.11) Local variables do not have the trailing underscore: `tempVariable`.
(4.12) It is ok (and preferred) to use local variables like `i`, `j`, `k` for indexing tasks
(4.13) Use `UPPERCASE_IDENTIFIERS` for `#define`d constant and macro identifiers. Avoid `#define`s!. See 5.8
(4.14) `#undef` local `#define`s whenever possible to clean up the namespace.
(4.15) If using suffixes and prefixes on labels:
- A name that begins with an uppercase letter is to appear directly after its prefix.
- A name that begins with a lowercase letter is to be separated from its prefix using an underscore ('_').
- A name is to be separated from its suffix using an underscore ('_').

5. Variables

One key idea for variables is to avoid polluting the global namespace (see section 3). Also, choose good variable names (see the naming section 4). In general avoid global variables since they break modularity, prefer variables with short scope over variables with large scope, and make your names clear and related to the *function* of the variable versus the *type* of the variable. Use `const` as much as possible to enable compiler optimizations and prevent linker conflicts. To wit:

(5.1) Avoid using global variables. They tend to make code brittle and hard to maintain.
(5.2) Choose variable names that suggest the usage. Clearly `houseCount` is a much better name than `aPositiveInteger` for the same variable.
(5.3) Encapsulate in a `class` or a `namespace` all global variables and constants, `typedef`s, and `enum`erated types.
(5.4) Make variables as local as possible. That is, declared them with the smallest possible scope.
(5.5) A variable with a large scope should have a long name.
(5.6) Postpone variable definitions until necessary [33, item 32]. This often speeds up code, and makes it clearer.
(5.7) Always use `unsigned` for variables which cannot reasonably have negative values. This gives the a higher top value.
(5.8) Prefer `const` and `inline` to `#define` [33, item 1]. Do not use the preprocessor directive `#define` to obtain more efficient code; instead, use `inline` functions. Make variables `const` whenever possible, and place constants in unnamed `namespace`s instead of using `#define` (see item 5.11). `#define`s pollute the global namespace, are not typesafe, and lead to errors. For example, what are the values of `a`, `b`, and `c` after the following?

```
int a=5,b=1,c;
c=MAX(a++,b);
```

Answer: it depends on how `MAX` is implemented! If `MAX` is the C++ standard template based `max` then the answer is as you'd expect: `a=6`, `b=1`, and `c=5`. However if max is the common `#define` based max

```
#define MAX(a,b) ((a)>(b) ? (a) : (b)) // Errors start here!!
```

Then the answer will be `a=7`, `b=1`, and `c=6`, so both `a` and `c` are incorrect.

This is a side effect of the `(a)` appearing twice in the `#define`, and shows one case where `#define`s lead to errors, so avoid them at all costs!

(5.9) Initialize a variable at the place it is declared, to avoid uninitialized variable errors. Compilers can give warnings about using uninitialized variables, sometimes even at runtime, but do not rely on this to catch errors. If you need a variable, most likely you know a value to place in it.

(5.10) Avoid the use of numeric values in code; use symbolic values instead such as `PI` or `M_PI` instead of `3.14159265358`. Do not `#define` them; make them of type `const...` and in an unnamed `namespace` (see item 5.11 if possible, otherwise `const...` Most compilers are smart enough to place `const` variables inline as if they were `#define`s, so there is no speed hit, yet you get added type safety.

(5.11) Using `static` to denote local linkage has been deprecated [38, p. 819]. Use unnamed `namespace`s instead. Thus,

```
static int LocalFunc1(void) {...} // BAD - DO NOT USE
static const int localVal1 = 10;  // BAD - DO NOT USE

namespace { // this unnamed namespace keeps things local
   int LocalFunc2(void) {...} // GOOD - NOTE NO static
   const int localVal2 = 10;  // GOOD - NOTE NO static
   }; // end of local namespace
```

6. Functions

Naming functions is often more important than naming variables, so refer to the naming section 4.

(6.1) Clearly document any assumptions made by functions in the comment block that precedes the function body, and any post conditions.

(6.2) Prefer pass by reference to pass by value [33, item 22]. This increases speed.

(6.3) Prefer pass by const reference `const &` to pass by pointer since it is safer. It's easier to check your references than to apply pointers correctly.

(6.4) Do not return a reference when you need an object [33, item 23]. This will lead to hard to find bugs.

(6.5) "Avoid functions with many arguments." [15, Rec. 41.] Three or fewer arguments is good; five or more is questionable. Arguments are the leading cause of divorce and war.

(6.6) Use `assert(...)` to test pre and post conditions wherever possible.

(6.7) A function should do only one thing and be fairly short.

(6.8) Write cohesive functions.

(6.9) Avoid long and complex functions. Functions over 100 lines should be relatively rare, but there are cases where this is unavoidable (such as a `switch` statement with numerous branches, each containing only a line or two).

(6.10) Make functions local to a file using unnamed `namespace`s (see item 5.11) to avoid link conflicts. No longer use `static` to make functions local; this has been deprecated [38, p. 819].

(6.11) If a C function has no parameters, use the keyword `void` to ensure consistency with C++. This helps the name mangling system link them For example, `int Doodad(void);`

(6.12) Use an explicit status parameter for a function that returns a status. Don't return an `int` with different values denoting different status codes.

(6.13) If a function will return a value via one or more function parameters, those parameters should appear after all input parameters for which there are not default values.

(6.14) If a function returns a success-or-failure status, success will be represented by the `bool` value `true` and failure by the `bool` value `false` whenever appropriate. This is not to be confused with the exceptions 11 definition of function failure.

(6.15) When overloading functions, all variations should be used for the same purpose.

(6.16) Minimize the number of temporary objects that are created as return values from functions or as arguments to functions.

7. Formatting Rules

The purpose of formatting is to make code readable and increase understandability. Liberal use of whitespace is good, since it gives space to make notes on printouts, and makes the code easier to grasp. See the sections on naming 4 and commenting 8 and reference [31, Chapter 18].

(7.1) Use indentation to denote subordinate control.

(7.2) Braces ("{}") which enclose a block are to be indented, on separate lines directly before and after the block.

(7.3) Spaces are used in the following:

- between comparisons, e.g., `i < 3`, `NULL != ptr`, etc.
- after commas in parameter lists: vb+int Func(int a, int b);+
- between binary operators: `a * b - 18` and `b1 && b2`
- in `for` loops as: `for (int i = 0; i < 10; i++)`

(7.4) Use tabs for initial indenting, and spaces for indenting after the first printable character. The initial tabs allow different porgrammers to set the spacing well. The following spaces help keep things aligned when tab sizes are changed in different editors.

(7.5) Always provide the return type of a function explicitly.

(7.6) In a function definition, the return type of the function should be written on the same line as the function name.

(7.7) When declaring functions, the leading parenthesis and the first argument (if any) are to be written on the same line as the function name.

(7.8) If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name.

(7.9) Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument).

(7.10) Always write the left parenthesis directly after a function name.

(7.11) The flow control primitives `if`, `else`, `while`, `for` and `do` should be followed by a block, even if it is an empty block, except in the case of a one line `if`.

(7.12) The dereference operator '*' and the address-of operator '&' should be spaced from the types and type names in declarations and definitions.

(7.13) Do not use spaces around '.' or '->', nor between unary operators and operands.

Thus functions look like

```
int FunctionName(int & val)
  { // comment
  if (0 == val)
     {
     val = 10;
     for (int i = 0; i < 5; i++)
         val *= val;
     }
  else
     val *= val;

  return val;
  } // FunctionName
```

8. Commenting Rules

Commenting is an important part of your code. Poor comments waste people's time trying to understand them. Missing comments fail to point out important issues when using code. Wrong comments mislead, leading to maintenance errors. Good comments are concise, point out important issues, explain reasons for design choices, and are local to the issue at hand.

A consistent method for commenting is important so you know where to go with questions about functions. In C++, header files are the de facto place to comment interfaces, reasons for classes existing, parameters and return types of member functions, and (God forbid) global variable explanations.

Each non trivial function should have a comment block preceding it explaining anything not obvious, usually this includes how to use it, the parameters allowed, the return codes, and if necessary, how it relates to the file as a whole. Is it local to the file? Is it something called only as a helper for a bigger task?

At the implementation file level (*.cpp file), it is good to explain the purpose of the file, perhaps include copyright info, perhaps tell when it was written and by whom and how it is compiled. If you use source control that automatically updates comments, here is a good place to have those updates occur.

At the function level, each local variable should have a comment explaining its use, unless it is very obvious. You should comment on parameters, return values, algorithms used, warnings, and gotchas.

This section has two main areas: 1) general commenting tips, and 2) a section on commenting in such a way that an auto document tool such as JavaDoc or Doxygen can automatically create documentation for your programs, libraries, and interfaces.

See the chapters "Layout and Style" and "Self-Documenting Code" in [31, Chapters 18, 19].

8.1. General commenting.

Here are recommendations to commenting, including file level comments, function level, class level, etc.

(8.2) Comments tell *why*, so you and others recall the reason for doing something.

(8.3) Comment the beginning a file: state the reason for its existence.
(8.4) Comment the end of file, so on printouts, etc., it is clear that there is no more, with a an `// end - FileName.cpp` line at the end of every file, to clarify on printouts and to verify that nothing was chopped off somewhere during some transmission (which has come in useful).
(8.5) Place a comment before each function and class member explaining its purpose, all input parameters, return values, and assumptions about input and output.
(8.6) Comment each `typedef`, `struct`, `class`, and `namespace` for usage and caveats.
(8.7) Comment the end of large scopes, such ar the end of a large loop, `if`, `#ifdef`, function, `class`, and `namespace`. This is a general comment explaining what is in the preceding block, i.e., `} // loop on customer`, or `#endif // DEFINED_LABEL`, etc. This makes code easier to navigate.
(8.8) Comment the reason for a variable where it is defined.
(8.9) Comment design decisions at the appropriate place.
(8.10) Place a "todo" comment at a place where you need to do more work, since this makes it easy to find later.

```
// todo - this section could be optimized
```

(8.11) Keep comments local - only one copy, and by source of problems. Comment interfaces in header files, and perhaps keep a copy of those comments near the definitions of those functions, classes, etc. But the header file is the definitive comment.
(8.12) If you use source control, and it has the ability to update files based on keywords in comments, place these appropriately. For example, most source control programs allow placing comments in the file denoting changes and who made them, which allows quick checking on why something changed, etc.

8.2. **Auto Documentation.** Doxygen, from www.doyxgen.org is an open-source, free, versatile documentation generator. It reads tags from comments in `*.h` header files (and even `*.cpp` source files), and generates documentation in many formats. If you have used JavaDoc for Java code, doxygen is similar yet more sophisticated, as the following table shows.

	Javadoc	Doxygen
Input language(s)	Java source code	C, C++, IDL, Java code
Obtain from	the Java Development Kit	www.doxygen.org
Output	HTML	HTML, LaTeX, RTF, PS, PDF, CHM, manpage

TABLE 1. Comparison of JavaDoc and Doxygen

Autodocumenting has the benefit of making the documentation as you code, which can be very useful, since it helps keep the documentation up to date. VS.NET now has the ability to make document websites from comments also, but the flexibility and cross platform availability of dxoygen makes it a better choice. The guidelines below are a good way to comment code, since they are the common

ground between doxygen and JavaDoc, so you can learn one commenting method. Note that doxygen has many other styles of commenting it supports. See the documentation for details.

The benefit of all this hassle: hyperlinked documentation to help learn, manage, and maintain code. You do not have to write separate, hard to keep up-to-date documentation. You can quickly get to the function you want from the documentation. We've all used bad docs and good docs - which really help you? How much did it help? Get in the habit of documenting well and you will never go back.

For C#? and Visual Basic code, you will probably need to learn Microsoft's commenting style.

Some useful links:

- An introduction for beginners is at the doxygen site.
- An easy way to setup doxygen for VC++ : www.codeproject.com/tips/doxysetup.asp
- Setup for VS.NET www.codeproject.com/macro/KingsTools.asp
- A comparison of these tools.

The basic idea is to start comments with `/**` to get doxygen's attention. Note the double `*`. You can place inline variable comments with a `/**< ... */` comment layout, such as

```
int b; /**< description */
```

Here are then the initial rules to using doxygen to convert comments into documentation. See the website tutorials and documentation to learn more advanced abilities.

(8.13) Comment each header file for which you want interface documentation. Commenting sourcecode files is possible, but rarely needed or used.

(8.14) Each doxygen comment starts with `\**`. Note the double star!

(8.15) Doxygen skips initial `*` on a line, thus the following two comments are parsed the same.

```
\** a comment block
  * is here
  */

\** a comment block
    is here
   */
```

(8.16) Very important! In order to have doxygen work on comments in a file, you *must* place a `/** @file filename.h */` line in this file naming the file.

(8.17) Before each function, member function, class, typedef, struct, enum, etc., that you want documented, place a block describing a brief and detailed comment, separated by a blank line as:

```
\** Brief description goes first.
*
* A more detailed
* description.
*
* @return Returns random nonsense.
*/
```

```
int Function(void);

/**
* A pure virtual member.
* @see ThatFunc()
* @param c1 the first argument.
* @param c2 the second argument.
*/

virtual void ThisFunc(char c1,char c2) = 0;
```

(8.18) Inline comments have a `/**< ... */` comment layout, and are useful after variables, `enum`s, simple functions, etc.

```
class Test
  {
  public:
    /** An enum type. (brief description)
     *  The documentation block cannot be put after the enum!
     */
    enum EnumType
      {
      int EVal1,     /**< enum value 1 */
      int EVal2      /**< enum value 2 */
      };

    void member();   /**< a member function. */

  protected:
    int value;       /**< an integer value */
  };
```

(8.19) Here is a list of useful keywords to tell doxygen how to format and place comment items:

- `@file` Name of the file youre documenting.
- `@brief` For specifying a brief description.
- `@param` Parameter name and its description.
- `@return` Description of return values.
- `@author` Used in each file to denote who made it.
- `@version` Version of the interface.
- `@throws` (or `@exception`) What exceptions this function throws.
- `@see` "See also" to link to other functions, classes, etc.
- `@since` Since what version this comment has existed.
- `@deprecated` This item is deprecated.

(8.20) Place a list in the documentation using:

```
/**
 *  A list of events:
 *    - mouse events
 *         -# mouse move event
```

```
*         -# mouse click event\n
*            More info about the click event.
*         -# mouse double click event
*    - keyboard events
*         -# key down event
*         -# key up event
*
*  More text here.
*/
```

The result in the documentation will be:
A list of events:

- mouse events
    1. mouse move
    2. mouse click
       More info about the click event.
    3. mouse double click event
- keyboard events
    1. key down event
    2. key up event

More text here.

(8.21) You can also do the above example by embedding the HTML list commands `<ul>`, `<li>`, and `<ol>`.

(8.22) Doxy can embed HTML directly in comments to make the output very sophisticated.

(8.23) Doxygen can do grouping (modules), LaTeXformulas, complex graphs and diagrams, and much much more.

9. Const correctness

Be `const` correct. That means whenever you have a parameter, member function, pointer, or variable that you do not want to be able to change, declare it as `const`. If a function does not change a parameter passed in by reference or pointer, make the parameter `const`, such as `int Func(const string & data)`. This gives a user of the function guarantee that the function will not modify the parameter passed, and improves code considerably. If a member function does not change the *user observable* state of an object, such as a simple query, declare the member function as `const` with `int GetSize(void) const`. This allows you then to pass the object as a `const` object into other functions, who are then only allowed to call the `const` member functions on the `const` object. Conversely, if you are passes a `const` object, you should not cast it to a non-const object and change it. You just broke a contract, and usually this is a very bad idea. So while designing programs, functions, variables, and objects, use `const` properly, and this will greatly strengthen your design and make your programs better.

Being `const` correct will eliminate a lot of errors in design, find bugs at compile time, prevent you from making a lot of mistakes, and in general protect objects from incorrect manipulation. It is a lot more fun to fix a bug at compile time than production runtime. Consider the differences:

```
char * ptr;                 // non const ptr, non const data
const char * ptr;           // non const ptr, const data
```

```
char const * ptr;           // const ptr, non const data
const char * const ptr;     // const ptr, const data
```

Always use `const` for a function parameter if it is not changed. `const` is a contract between the user of a function and the behavior of the function. Use `const` in function return types and parameters. For example, a class member: `const string & Myname(void) const;` returns a string to the object name by `const` reference is able to keep the member function `const`, avoiding the cost of a copy constructor.

See the brief article at [1, Section 18] for further explanation, or as usual search Google for "const correctness". Now for some rules:

(9.1) `const` all that you can! [33, item 21].

(9.2) Use constant references (`const &`) instead of call-by-value, unless using a pre-defined data type or a pointer. This is faster.

(9.3) Never convert a `const` to a non-const. See item 14.28 for casting information.

(9.4) A member function that does not affect the state of an object (its instance variables) is to be declared `const`.

(9.5) Use `const` or `const` in an unnamed `namespace` (see item 5.11) to declare constants in your code instead of `#define`. See item 5.8. For example:

```
// header MathUtil.h
extern const double M_PI; // Note the extern here

// in file MathUtil.cpp
#include "MathUtil.h"     // Note use of quotes versus <>
const double M_PI = 3.14159265358979323846;
```

10. Classes

The purpose of classes are to implement abstract data types. C++ classes were designed to ease building large systems, and entail a lot of keywords and nuances. The entire situation is far too large to cover here, so consult your local bookstore, psychic, or online programming forum to get more details. But learning how to design, use, and extend classes well is what adds the ++ to C giving C++. That said, here are some general guidelines:

(10.1) Keep class interfaces complete and minimal [33, item 18].

(10.2) Avoid `public` data members [33, item 20]. (Except perhaps for very simple types, like a `Point3D` that holds variables `x_`, `y_`, `z_`, and some members to operate on points.)

(10.3) Classes with pointers to member data need a copy constructor, an assignment operator (see item 10.6!), and a destructor to avoid memory leaks [33, item 11].

(10.4) Prefer initialization to assignment in constructors [33, item 12]. This executes faster. Prefer `Point3D(void) : x_(0), y_(0), z_(0) {};` to the slower `Point3D(void) {x_=y_=z_=0;};`.

(10.5) An assignment operator ought to return a reference to the object using `return *this;`, and take a `const` reference as a parameter [33, item 15]. This prevents unnatural constructs with your objects. Write

```
String & operator=(const String & str)
```

(10.6) An assignment operator which performs a destructive action must be protected from performing this action on the object upon which it is operating. Check for self assignment in `operator=` [33, item 17]. Thus check for self assignment with code like `if (this == &obj)` before assigning pointers to data members; otherwise you will get resource leaks.

(10.7) Class members are initialized in the order they appear in the class, not the initialization order! [33, item 13]. So do not make assumptions about order in member initialization lists.

(10.8) Make sure base classes have `virtual` destructors [33, item 14]. Otherwise you will have a hard to find bug, since derived classes will call the wrong destructor. **Rule**: if the class has any `virtual` members, it must have a `virtual` destructor.

(10.9) Explicitly disallow implicitly generated member functions you don't want [33, item 27]. For example, if you do not want to allow certain type conversions, declare those conversions explicitly as `private` member functions, disallowing users from doing those conversions implicitly, which the compiler would otherwise do.

(10.10) Use the Pointer To Implementation (pimpl) idiom to increase modularity and lower compile times. See "Guru of the Week # 24" [39]. This removes class internals from class interface, making it easier to change internals later without breaking headers. Plus you can ship your headers without information on how classes are implemented. It is described in detail in GOTW issues 7, 15, 24, 25, and 28.

(10.11) A `public` member function must never return a non-const reference or pointer to member data. Only the object itself should manipulate it's internal structure.

(10.12) Friends of a class should be used to provide additional functions that are best kept outside of the class.

(10.13) Avoid the use of global objects in constructors and destructors. Global destruction could result.

(10.14) Use operator overloading sparingly and in a uniform manner.

(10.15) Give derived classes access to class type member with `protected` access functions.

11. Exception Handling

Exception handling is probably the trickiest part of the C++ language to understand and apply well. Why should you understand it? Well, Like a wise man once said: "exceptions happen." And as mentioned in section 1, the standard library throws exceptions, so for your code to work properly, you should handle them. An uncaught exception terminates your program (item 11.16). The difficulty is summed by by C++ luminary Scott Meyers:

"When I give talks on EH, I teach people two things:

"**POINTERS ARE YOUR ENEMIES**, because they lead to the kinds of problems that `auto_ptr` is designed to eliminate.

"**POINTERS ARE YOUR FRIENDS**, because operations on pointers can't `throw`.

"Then I tell them to have a nice day :-)"

For an example of the current state of the `union` :), most compilers currently return `0` or `NULL` when `new` fails to allocate memory, but the C++ standard requires `new` to throw a `bad_alloc` exception on failure, and does not say what the return value of `new` should be. See item 2.4 for more information. And compilers, even Microsoft's, are changing to conform to the Standard, so that code we have all written like

```
Obj * ptr = new Obj;
if (NULL == ptr)
   DoError();
```

will actually not work as we wanted it to work. Instead, you'd need

```
try {
   Obj * ptr = new Obj;
   }
catch(exception::bad_alloc & e);
   {
   DoError();
   }
```

So there is reason, grasshopper, to learn and become wise in the ways of exception handling. It is such a difficult area that the C++ community took many years to understand it well, and to come up with some recommendations that work well in practice. All this wisdom will not fit here, so you should read the following items:

(11.1) "The C++ Programming Language, Special Third Edition" by Stroustrup has an entire chapter on exception handling, and should explain a lot to the novice [38, Ch 14].

(11.2) Herb Sutters Guru of the Week (GOTW) # 8. This is a great great introduction to exception handling.

(11.3) Tom Cargill's excellent article, "Exception Handling: A False Sense of Security" [8]. In it Cargill shows how exceptions can be very, very tricky to get perfect. It is a must read.

(11.4) Herb Sutter's article "A Pragmatic Look at Exception Specifications" at www.gotw.ca/publications/mill22.htm.

(11.5) The C++ FAQ online Exception Handling section

(11.6) Again, Herb Sutter's GOTW numbers 47, 56, 59, 60, and 65 for more cases and coverage.

(11.7) Finally, Herb Sutter's GOTW 82, a very good up to date summary of what the C++ community knows about good exception handling practices.

Exception handling syntax uses the keywords `try`, `catch`, and `throw`. This is best shown with an example:

```
// Example 1: A try-block example
//
try {
   if( test )
      throw string( "Error Text" );
   else if( other_test )
```

```
      throw 42;
   } // end of try block, check for exceptions

catch( const string& )
   {
   // do something if a string was thrown
   if (cannotHandleIt)
       throw; // if we couldn't handle it, re throw it
   }
catch( ... ) // ... catches any exception that gets this far
   {
   // do something if anything else was thrown
   }
```

So you put regular code in a `try` block, `throw`ing objects to denote errors. A `catch` block, *anywhere up the call stack*, will catch the object thrown, cleaning up the stack as necessary by calling destructors properly. It is important to realize that a `catch` does NOT have to be in the same function as in the example above. Notice that if a `catch` block, can only partially handle an exception, or perhaps not handle it at all, the block can always rethrow the caught exception with a simple `throw;` statement.

Another use is to tell the compiler what exceptions a function can `throw` with an exception specification. Basically

```
int Func1(void);            // can throw anything

int Func2(void) throw();    // will throw nothing

int Func3(void) throw(A,B); // can only throw A or B
```

By default, in C++, `Func1()` could indeed `throw` anything. It is possible to tell the compiler and the reader what kinds of things a function may `throw`, but it turns out to be a bad idea.

The compiler enforces *at runtime* that functions will only `throw` listed exceptions (possibly none). If an exception gets `thrown` in a function not allowed to `throw` it, `std::unexpected()` will be called, bringing down the house. So do not put specifications on a function unless you know know subordinate functions will NOT `throw` other objects. This is a nightmare if some low level function `BadFunc` suddenly needs to `throw` a `Foo`, because you now need to go to every function calling `BadFunc` and add a `throw(Foo...)`, and then to every function calling those, and on and on. This leads to:

Moral #1: Never write an exception specification. Even experts don't bother.

To quote Herb Sutter:

"Exception specifications can cause surprising performance hits, for example if the compiler turns off inlining for functions with exception specifications.

"A runtime `unexpected()` error is not always what you want to have happen for the kinds of mistakes that exception specifications are meant to `catch`.

"You generally can't write useful exception specifications for function templates anyway because you generally can't tell what the types they operate on might `throw`."

The Boost development team has found that a throws-nothing specification on a non-inline function is the only place to do it. See item 11.14. Thus

Moral #2: You can place a no `throw` on an atomic function, but it is fine to avoid even that.

Important concepts are the three Abrahams Guarantees for exception safety. Once you understand some exception handling problems and what types of things can throw exceptions, you should wonder what level of safety your functions and objects attain:

(11.8) **Basic Guarantee**: If an exception is `thrown`, no resources are leaked, and objects remain in a destructible and usable – but not necessarily predictable – state. This is the weakest usable level of exception safety, and is appropriate where client code can cope with failed operations that have already made changes to objects' state.

(11.9) **Strong Guarantee**: If an exception is `thrown`, program state remains unchanged. This level always implies global commit-or-rollback semantics, including that no references or iterators into a container be invalidated if an operation fails. Also, the pimpl 10.10 (Pointer to Implementation) idiom turns out to be necessary to achieve this level of safety [39, GOTW #59].

(11.10) **Nothrow Guarantee:** The function will not emit an exception under any circumstances. It turns out that it is sometimes impossible to implement the strong or even the basic guarantee unless certain functions are guaranteed not to `throw` (e.g., destructors, deallocation functions). As the references above explain, an important feature of the standard `auto_ptr` is that no `auto_ptr` operation will vb+throw+. It has been shown that some functions *must* have this level of safety in order to allow other functions to have the *strong guarantee*.

A useful idea is the canonical form of a strongly exception-safe copy assignment. First, provide a nonthrowing Swap() function that swaps the guts (state) of two objects:

```
void T::Swap( T & other ) throw()
   {
   // ...swap internals of *this and other, no throwing allowed!
   }
```

Second, implement `operator=()` using the "create a temporary and swap" idiom:

```
T & T::operator=( const T & other )
   {
   T temp( other ); // do all the work here - this may throw
   Swap( temp );    //   then "commit" the work using
   return *this;    //   nonthrowing operations only
   }
```

As a final note, before the list of suggested practices, here are some conclusions reached by the C++ community, as paraphrased from Herb Sutter:

Conclusion 1: Exception Safety Affects a Class's Design. Exception safety is never "just an implementation detail." Classes need designed with exception safety in mind, similar to designing a class to handle multithreading. The pimpl 10.10 transformation is a minor structural change, but can be required to make a class safe. GOTW #8 shows an example of how exception safety considerations affects the design of member functions.

Conclusion 2: You Can Always Make Your Code (Nearly) Strongly Exception-Safe. This is a commendable goal, but necessary if you are writing code that must not fail (such as medical, military, Playstation 2). But you need to read, read, read on the subject and learn to do it well.

Thus, to summarize exception handling:

(11.11) Learn exception handling by reading relevant sources.

(11.12) `throw` an exception when a constructor fails, since there is no return code.

(11.13) NEVER THROW AN EXCEPTION FROM A DESTRUCTOR. It will wreck stack unwinding, leaking resources all over your new shoes. The C++ language will call `terminate()` at this point, bringing down the house.

(11.14) Don't bother with exception specifications - except possibly `throw()` on non-inline non-virtual non-template functions.

(11.15) A big tree of exceptions rooted in the standard C++ class `exception` is good. There are many predefined exceptions you should use and derive from in item 11.18.

(11.16) Uncaught exceptions call `terminate()`. So `catch` as `catch` can.

(11.17) Always `catch` exceptions by reference.

(11.18) The C++ standard defines exceptions, all rooted under class `exception` as following:

- `exception`
  - `logic_error`
    - `length_error`
    - `domain_error`
    - `out_of_range`
    - `invalid_argument`
  - `bad_alloc`
  - `bad_exception`
  - `ios_base::failure`
  - `bad_typeid`
  - `bad_cast`
  - `runtime_error`
    - `range_error`
    - `overflow_error`
    - `underflow_error`

(11.19) In order to reach the **Strong Guarantee** above, it is sometimes necessary to use the pimpl transformation 10.10.

12. Efficiency

If your programs are inefficient, slow enough to make users moan, then they will not use your program very long. However, when building a program, rough efficiency estimates are good enough. It is useful to understand efficiency issue

during design and coding, but do not go overboard on it. The proper time to speed up code is only when it needs it. Do not waste months writing the hyper-optimized sort only to find out it is not going to be used in the final program, or worse yet, that you spent the entire time budget on the sort, and never got to the main program.

If and when you need to increase efficiency, profile your code first as in section 1, under profiling. Then, once you have hard data on what is causing your program to be slow, consider the following:

(12.1) Find a better algorithm. The best way to get major speedups in code is to find better algorithms for time intensive tasks. A good book on algorithms is the Cormen, Leiserson, Rivest, and Stein text [10], and the bible is the 3 volume set "The Art of Computer Programming" by Knuth [21, 22, 23]. After this you may need domain specific algorithm references, say in graph theory, signal processing, cryptography, compression, etc.

(12.2) Do not optimize without the help of a good profiler - do not guess where the bottlenecks are. (Lomont's 1^{st} law of time wasted). You will spend a lot of time on optimizing the wrong sections of code. Bentley has several chapters devoted to optimizing methods in [4].

(12.3) Use lazy evaluation [32, item 17]. That is, do not compute things you do not need yet.

(12.4) Conversely, amortize costs. If you will need things in the future which will be slow to compute, spread the work to other times when there are more cycles available [32, item 18]

(12.5) Understand temporary objects [32, item 19]. How and when they are generated can cost a lot in efficiency.

(12.6) Use reference counting, write on modify data, and other tricks [32, item 29]. This allows items to share data, speeding performance in many cases.

(12.7) Small object pools - if you find you create lots of small objects, and destroy them, over and over, you might consider a custom allocator for them, that manages a pool of memory more efficiently than the standard runtime. A very good reference on this is [2, Chapter 4].

(12.8) . Prefer `++i` to `i++`. Sound crazy? Well, prefix forms return a reference (fast), and postfix forms return a `const` object, making `++i` faster for many items. So if `i` is an iterator, for example, `++i` should be a lot faster. So get in the habit of preferring prefix rather than postfix increments. [32, item 6].

For fun: Is `a+++++b` legal? Explain why or why not.

13. Portability Recommendations

A good way to test portability of your code is to use several compilers. Since this is often hard to do for platform specific code, keep the following items in mind while coding.

(13.1) Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another.

(13.2) Avoid the direct use of pre-defined data types in declarations in places where size matters. Thus

(13.2.1) Do not assume that an `int` and a `long` have the same size.

(13.2.2) Do not assume that an `int` is 32 bits long (it may be only 16 bits long, or 36 bits, etc.).

(13.2.3) Do not assume that pointers and integers have the same size.

(13.3) Be careful not to make type conversions from a "longer" type to a "shorter" one. Also remember that on your compiler, a `long` and a `int` may be the same size, but on a future platform they may differ, so even if you are casting between supposedly same types today, they may not be the same tomorrow. Avoid casting.

(13.4) Never specify relative UNIX names in `#include` directives. This is not portable.

(13.5) Never include your project header using `#include <...>`, since the `<..>` is reserved for system files. Include using `#include ".."` since this is portable.

(13.6) Do not assume that a `char` is `signed` or `unsigned`.

(13.7) Always set `char` to `unsigned` if 8-bit ASCII is used.

(13.8) Use explicit type conversions for arithmetic using `signed` and `unsigned` values.

(13.9) Do not assume that you know how an instance of a data type is represented in memory.

(13.10) Do not assume that `long`s, `float`s, `double`s or `long double`s may begin at arbitrary addresses. Some architectures require them byte or word aligned.

(13.11) Do not depend on underflow or overflow functioning in any special way.

(13.12) Do not assume that the operands in an expression are evaluated in a definite order (except short circuiting expression evaluation).

(13.13) Do not assume that you know how the invocation mechanism for a function is implemented.

(13.14) Do not assume that objects are initialized in any special order in constructors [2, Chapter 6]. See item 10.7.

(13.15) Do not assume that `static` objects are initialized in any special order. There is *no order* defined across files, so such code becomes compiler dependent. If your `static` objects rely on other `static` objects being initialized, you will get into trouble. See [2, Chapter 6] for details, and possible solutions. Also, see item 5.11.

(13.16) Avoid using shift operations instead of arithmetic operations. These are error prone and hard to debug. Most modern compilers (gcc, VC++) will replace common arithmetic code with faster shifting tricks automatically. The VC++.NET 2003 optimizer even is smart enough to replace

```
void Test1(void)
    {
    unsigned long val1 = 1, val2 = 2, val3 = 3;

    for (int pos = 0; pos < 10; pos++)
        {
        val1 *= 10;
        val2 += 3;
        val3 += val2;
        if (val2&1)
            {
            val1 -= 5*val2;
            val2 += 1;
```

```
            }
        val1 *= val3/1000;
        }
    val1 = val1+val2+val3;

    cout << val1 << endl;
    } // Test1
```

with the equivalent

```
void Test1(void)
    {
    cout << 275 << endl;
    } // Test1
```

14. General Rules

Here we collect a lot of guidelines, rules, and so on, that do not easily fit anywhere else. First of all, a nice and useful trick. When you want to prevent anyone from using your class as a base class (for example, you do not want virtual destructors), the C++ FAQ Lite has a nice section [1, Section 23.8] on **The Final Class Trick**.

```
class Final; // forward declaration necessary

class Base
   { // do nothing class - it is empty
private:
   friend Final; // make the derived class a friend
   Base() { };   // private constructor!
   };

class Final : private virtual Base // note private
   {
 public:
   // usual stuff here
   };
```

How does it work? Well, since a derived class must call the base class constructor, and `Base` has a private destructor, no one can derive from it, except its only friend, class `Final`. Now if you derive a class from `Final`, since the new class will not be a `friend` of `Base`, they will not play well together, and you get a compiler error. Hence you have made a class from which others cannot be derived.

Now, the guidelines:

(14.1) There should be one name per *.h/*.cpp pair, and it should be the name of the class defined within. Include the header in the C++ file to ensure function declaration consistency.

(14.2) Include files in C++ always have the file name extension ".h". Implementation files in C++ always have the file name extension ".cpp".

(14.3) Always `#include` proper headers [34, item 48]. If something is needed in a file, include it explicitly. Don't expect different headers to bring in items, since this is compiler dependent.

(14.4) Use the directive `#include "filename.h"` for user-prepared include files. Use `#include <header>` for include files from libraries.

(14.5) Every file that contains source code must be documented with an introductory comment that provides information on the file name and its contents.

(14.6) All files should include copyright information.

(14.7) Every include file must contain a mechanism that prevents multiple inclusions of the file as follows. Note the use of all caps for defines and the leading underscore.

```
#ifndef _FILENAME_H #define _FILENAME_H

// ....your stuff here

#endif // _FILENAME_H
```

(14.8) Strive for low coupling between cohesive modules.

(14.9) Prefer `0==i` to `i==0` in comparisons. This prevents mistyping `i=0`, since `0=i` does not compile.

(14.10) When the following kinds of definitions are used (in implementation files or in other include files), they must be included as separate include files:
- classes that are used as base classes
- classes that are used as member variables
- classes that appear as return types or as argument types in function/member function prototypes
- function prototypes for functions and/or member functions used in inline member functions that are defined in the file

(14.11) Every implementation file is to include the relevant files that contain:
- declarations of types and functions used in the functions that are implemented in the file
- declarations of variables and member functions used in the functions that are implemented in the file.

(14.12) Select restrictive argument types. This primarily includes the keywords `unsigned` and `const`, and using enumerations in preference to simple integers for return types and restricted parameters.

(14.13) Be sure that all functions have an explicit return type, even if that type is `void`.

(14.14) "A function must never return a reference or a pointer to a local variable." [15, Rule 34] This rule applies to all C functions and to public C++ functions (with a possible exception for `static` member functions).

(14.15) The choice of loop construct (`for`, `while` or `do-while`) should depend on the specific use of the loop.

(14.16) "Always use inclusive lower limits and exclusive upper limits." [15, Rec. 52].

(14.17) Avoid the use of `continue` in loops.

(14.18) Use `break` to exit a loop if this avoids the use of flags.

(14.19) Do not use identifiers which begin with one or two underscores ('_' or '__') (other than header defines) since these may be reserved and compiler dependent.

(14.20) Use `()` to show precedence instead of using the built in precedence. This makes the code much easier to understand, and shows intent clearly.

(14.21) Order comparisons left to right, i.e., `((3<i)&&(i<=10))` as opposed to `((i<=10)&&(i>3))`.

(14.22) If the behavior of an object is dependent on data outside the object, this data is not to be modified by `const` member functions. This is to express intent to outer uses, not as an internal design decision. See `mutable` and the section on `const` 9.

(14.23) Do not use unspecified function arguments (ellipsis notation). It is not typesafe.

(14.24) The names of formal arguments to functions are to be specified and are to be the same both in the function declaration and in the function definition.

(14.25) Always specify the return type of a function explicitly. A public function must never return a reference or a pointer to a local variable.

(14.26) If possible, always use initialization instead of assignment.

(14.27) Do not write code which depends on functions that use implicit type conversions. Prefer C++ style casts [32, item 2]. Any type conversions must be explicit, to show others you really meant it. However, see item 14.28 to see how to correctly do casting in C++.

(14.28) Do not use traditional C style casts, since they are very dangerous (like pointers). C casts let you cast very unrelated things together, almost certainly being non portable and introducing bugs. There are four new casting types:

- `static_cast<T>`: This replaces the traditional C style cast, is safer and easier to find with grep, among other benefits. It is used like:

```
int a=1;
double b;
b = (double)a;             // old style cast. Bad!  :(
b = static_cast<double>a; // new style cast. Good! :)
```

- `const_cast` This is used to remove the `const` or `volatile` from an object. Any other use gives an error. If you are using this, you are violating the `const` guarantee (section 9) you were entrusted with, so use this sparingly and wisely.
- `dynamic_cast` is used for casting up and down an inheritance hierarchy in a type safe manner. Since this requires runtime support, this type of casting can be costly in performance.
- `reinterpret_cast` is almost always compiler dependent, making your code non portable. For an explanation read the recommended literature, since I will not contribute to the delinquency of minors.

The new casting types are explained well in [32, item 2], GOTW #17 and [41, 42].

(14.29) The code following a `case` label must always be terminated by a `break` statement. A `switch` statement must always contain a `default` branch which handles unexpected cases.

(14.30) Never use `goto`. This causes non exception safe code and results in resource leaks. Violators burn in hell.

(14.31) Use `strncpy` instead `strncpy`, etc., whenever possible, to help prevent buffer overruns. It is preferable to use the C++ Standard functions to prevent worrying so much about buffer sizes.

(14.32) Do not call `assert(..)` on a function that needs to be called. The assert will not call the function in release code. For example:

```
assert(true == DoSomethingUseful());
```

will fail to `DoSomethingUseful()` in release mode when the `assert` is disabled. Instead do:

```
bool retval = DoSomethingUseful();
assert(true == retval);
```

(14.33) Constants are to be defined using `const` or `enum`; never using `#define` 5.8. These are good to place in a `namespace`. See also the variables 5 section.

(14.34) Use the non *.h headers. `#include <cmath>` is better than the usual `#include<math.h>` since the former places names in `namespace std`. See section 3.

(14.35) Prefer `iostream` to `stdio.h` [32, item 2].

(14.36) Use inlining judiciously [33, item 33]

(14.37) Minimize compilation dependencies [33, item 34]

(14.38) Prefer compile time and link time errors to runtime errors [33, item 46]

(14.39) Understand compiler warnings [33, item 48]

(14.40) Learn the standard library [33, item 49]

(14.41) Always prefer `inlined` functions to macros. See 5.8 for reasons why.

(14.42) Use inline functions when they are really needed.

(14.43) Access functions are often good to have inline.

(14.44) `inlines` should never have `static` local variables, since these may be scattered across files erroneously.

(14.45) Distinguish between pointers and references [32, item 1].

(14.46) Prevent resource leaks in constructors [32, item 1].

(14.47) Program for change [32, item 32].

(14.48) Learn the language [32, item 35]. The resources section 16 has many good books, links, and other ways to find information. Some areas to read about are

- The Standard Template Library (STL).
- `strings` and `streams`
- RTTI
- `namespaces`. See section 3.
- Keywords like `bool`, `true`, `false`, `mutable`, `explicit`, `volatile`, `typeid`, `typename`, `try`, `catch`, `throw`, `using`, `explicit`, `register`, and `namespace`.
- Initializing `static` class members in the class.
- Templates.
- Exception handling. See section 11.

(14.49) Call `empty()` versus `size()==0` [34, item 4] on STL containers. It is usually quicker to check a container for empty than to compute its size.
(14.50) Never create containers of `auto_ptr` [34, item 8].
(14.51) Prefer `string` and `vector` to dynamically allocated arrays [34, item 13].
(14.52) Use `reserve()` in STL containers to avoid unnecessary allocations[34, item 14].
(14.53) Avoid `vector<bool>` - use `deque<bool>` or `bitset` instead [34, item 18]. For dynamically allocated bitsets use the Boost library [5].
(14.54) Prefer algorithms to hand written loops [34, item 43]. This is often faster, since the STL algorithms may know about the internals of the STL objects beyond the `iterator` interface exposed to the user.
(14.55) Do not change a loop variable inside a `for` block.
(14.56) All `switch` statements should have a `default:` case to catch errors.
(14.57) Do not compare one `float` or `double` to another for equality, since this is rarely correct. Use some tolerance: `fabs(val1-val2)<tolerance`.

15. Recommendations

(15.1) Think twice before you begin. Think three times before you change something you do not understand.
(15.2) Hungarian notation is more trouble than it is worth for evolving systems. More important to a reader of code than type of an object is meaning of an object. Name accordingly.
(15.3) Eliminate all warnings. Ignoring warnings will lead to trouble. More importantly, *understand* the warnings.
(15.4) Use STL containers instead of rolling your own. See section 1 on not reinventing the wheel.
(15.5) Prefer `string` to `char *`.
(15.6) Prefer streams using `#include <iostream>` to `printf` and other functions from `#include <stdio.h>`.
(15.7) Prefer `fstream`s to `FILE *` for file handling.
(15.8) Have a condensed version of the main points as a first pass for development. Fill in details with code.
(15.9) Always undefine local macros. This helps keep the namespace clean. Better yet - avoid `#define` like the plague that is second only to `goto`! See item 5.8.
(15.10) No `#pragma` directive should be used. `#pragma` directives are, by definition, non-standard, and can cause unexpected behavior when compiled on other systems. On another system, a `#pragma` might even have the opposite meaning of the intended one. In some cases `#pragma` is a necessary evil. Some compilers use `#pragma` directives to control template instantiations. In these rare cases the `#pragma` usage should be documented and, if possible, `#ifdef` directives should be to ensure other compilers don't trip over the usage.
(15.11) Optimize code only if you know that you have a performance problem. See section 12
(15.12) An `#include` file should not contain more than one class definition.
(15.13) Always give a file a name that is unique in as large a context as possible.
(15.14) Capitalization of filenames follows classes contained within.

(15.15) An `#include` file for a class should have a file name of the form name + extension. Use uppercase and lowercase letters in the same way as in the source code.
(15.16) Write some descriptive comments before every function.
(15.17) Use `//` for comments.
(15.18) When two operators are opposites (such as `==` and `!=`), it is appropriate to define both.
(15.19) Avoid inheritance for parts-of relations.
(15.20) Take care to avoid multiple definition of overloaded functions in conjunction with the instantiation of a class template.
(15.21) Use a `typedef` to simplify program syntax when declaring function pointers.
(15.22) Use parentheses to clarify the order of evaluation for operators in expressions. See section 1 for an example of how this clarifies code.
(15.23) Avoid global data if at all possible. If not possible, hide it behind access functions to assist portability and maintainability.
(15.24) Use exception handling correctly. This is very hard. See section 11 for information.
(15.25) Check the fault codes which may be received from library functions even if these functions seem foolproof. This will help find errors much more quickly.

16. Resources

Contained within are resources for learning and improving your C++ knowledge, and other resources that assist coding.

(16.1) Herb Sutter's "Guru of the Week" series at www.gotw.ca/gotw [39].
(16.2) The "C/C++ Users Journal" has a lot of online columns.
(16.3) The Boost libraries www.boost.org contain a large number of free, excellent libraries for a lot of useful tasks, from regular expressions, to portable threading, to graph algorithms, to a dynamic bitset, and much much more. It is well worth looking into.

Here are references for coding style. The most in depth one is the Ellemtel [15] 16.7 version. See also [7, 24], and especially [31]. You need hyperlinks to get to them, or search on the web.

Forthcoming is "C++ Coding Standards" by Herb Sutter and Andrei Alexandrescu, Addison-Wesley, 2003. Unless the earth spirals into the sun, this should be a good book, judging from the author's past works.

(16.4) A long standard
(16.5) NFRA guidelines
(16.6) Mozilla coding style
(16.7) The Ellemtel guidelines [15]
(16.8) The 60 page CERN Standard
(16.9) A brief convention at Artic Labs
(16.10) UWYN C+++ Coding Standard
(16.11) A brief government style guideline
(16.12) Tom Ottinger's naming conventions
(16.13) Chris Lott's list of coding guidelines
(16.14) A Coding Convention for C++ Code
(16.15) The C++ FAQ Lite Coding Standards

(16.16) FX-ALPHA C and C++ Coding Conventions
(16.17) Allen Clark's 1995 SES C++ Coding Conventions
(16.18) Programming in C++, Rules and Recommendations
(16.19) Linux Kernel Coding Style: (short yet full of bad ideas)

16.1. **Learning C++.** Here are some good books to learn C++. Also, check the references in the bibliography, since some are listed there that do not appear in this list.

(16.20) "Accelerated C++," Andrew Koenig and Barbara Moo" [25].
(16.21) "Essential C++," Stan Lippman [29].
(16.22) "The C++ Programming Language, Special Third Edition", Bjarne Stroustrup [38].
(16.23) "Design & Evolution of C++," Bjarne Stroustrup [37].
(16.24) "Effective C++, 2nd Edition", Scott Meyers, [32].
(16.25) "More Effective C++", Scott Meyers [33].
(16.26) "Efficient C++", Dov Bulka and David Mayhew [6].
(16.27) "C++ FAQs, 2nd Edition", Marshall P. Cline, Greg A. Lomow, and Mike Girou [9].
(16.28) "Thinking in C++, 2nd Edition, Volume 1," (3rd Edition to appear) Bruce Eckel [13].
(16.29) "Thinking in C++, 2nd Edition, Volume 2," Bruce Eckel and Chuck Allison [14].
(16.30) "C++ Gotchas," Stephen Dewhurst [12].

16.2. **Intermediate C++.**

(16.31) "Effective STL", Scott Meyers [34].
(16.32) "Exceptional C++", Herb Sutter [41].
(16.33) "More Exceptional C++", Herb Sutter [42].
(16.34) "Inside the C++ Object Model", Stan Lippman [28].
(16.35) "Mastering C++ Large-Scale C++ Software Design", John Lakos [26].
(16.36) "Generic Programming and the STL", Matt Austern [3].
(16.37) "STL Tutorial and Reference Guide, 2nd Edition," David R. Musser *et. al.* [17].
(16.38) "C++ Templates, The Complete Guide," David Vandervoorde and Nicolai M. Josuttis [20].
(16.39) "The C++ Standard Library," Nicolai Josuttis [19].
(16.40) "The C++ Standard Template Library," P.J. Plauger *et. al.* [18].
(16.41) "Standard C++ IOStreams and Locales," Angelika Langer and Klaus Kreft [27].

16.3. **Learning Advanced C++ Topics.** Here are some books for more advanced topics.

(16.42) "Modern C++ Design", Andrei Alexandrescu [2].
(16.43) "Generative Programming", Krzysztof Czarnecki & Ulrich Eisencecker [11].
(16.44) Multi-Paradigm Design for C++ James O. Coplien Addison-Wesley, 1998 ISBN: 0201824671
(16.45) Smart pointers in Boost: www.cuj.com/documents/s=8470/cuj0204karlsson/

(16.46) "Exception Safety in Generic Components", David Abrahams www.boost.org/more/error_handling.html

16.4. **Algorithms.** Finally, you need algorithms like a fish needs water. Here are the canonical references, which any serious programmer, and even facetious ones, should have as reference. They are listed in order of depth, but the Knuth books are the bible of algorithms (and he is working on the next volume "Combinatorial Algorithms", planning volumes 5, 6, and 7: "Syntactic Algorithms" (including compression), "Theory of Context-Free Languages", and "Compiler Techniques").

(16.47) "Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching (3rd Edition)", and "Algorithms in C++ Part 5: Graph Algorithms (3rd Edition)", Robert Sedgewick, [35, 36].

(16.48) "Introduction to Algorithms, Second Edition", Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein [10].

(16.49) The three volume series in "The Art Of Computer Programming" : "Volume 1: Fundamental Algorithms (3rd Edition)", "Volume 2: Seminumerical Algorithms (3rd Edition)", and "Volume 3: Sorting and Searching (2nd Edition)", Donald Knuth, [21, 22, 23].

17. The End!

This brings us to the close. I hope you have enjoyed this more than I did while writing it, because it was very painful and took a long time. But I wrote it mostly for myself, to solidify my thoughts on a lot of this material. I do hope that you get some use out of it. Read it, put it in a draw or folder on your hard drive, and read it again and again over time. It grows on you like a mold or fungus.

Please email corrections, suggestions, and criticism to `clomont@cybernet.com` or `clomont@math.purdue.edu`. Versioning of this document will be by the date on the bottom of the first page.

Finally, thanks for your support and patience if you got this far!

Answer to Pop Quiz from beginning: There is no completely specified order a given compiler will evaluate the items. The functions and expressions may be evaluated in any order that respects the following three obvious rules:

- `expr1` must be evaluated before `g()` is called.
- `expr2` must be evaluated before `h()` is called.
- Both `g()` and `h()` must complete before `f()` is called.

References

1.
2. Andrei Alexandrescu, *Modern C++ Design*, Addison-Wesley, Boston, New York, 2001, ISBN: 0201704315.
3. Matt Austern, *Generic Programming and the STL*, Addison-Wesley, Boston, New York, 1998, ISBN: 0201309564.
4. Jon Bentley, *Programming Pearls, 2nd Edition*, Addison-Wesley, 1999, ISBN: 0201657880.
5. *Boost template libraries*, www.boost.org.
6. Dov Bulka and David Mayhew, *Efficient C++*, Addison-Wesley, Boston, New York, 1999, ISBN: 0201379503.
7. Tom Cargill, *C++ programming Style*, Addison-Wesley, Boston, New York, 1992.
8. ———, *Exception Handling: A False Sense of Security*, C++ Report **9** (1994), no. 6.

9. Marshall P. Cline, Greg A. Lomow, and Mike Girou, *C++ FAQs, 2nd Edition*, Addison-Wesley, Boston, New York, 1998, ISBN: 0201309831.
10. Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition*, MIT Press, 2001, ISBN: 0262032937.
11. Krzysztof Czarnecki and Ulrich Eisencecker, *Generative Programming*, Addison-Wesley Professional Computing Series, Boston, New York, 2000, ISBN: 0201309777.
12. Stephen Dewhurst, *C++ Gotchas*, Addison-Wesley, 2002, ISBN: 0321125185.
13. Bruce Eckel, *Thinking in C++, 2nd Edition, Volume 1, (3rd Edition to appear)*, Prentice Hall, 2000, ISBN: 0139798099.
14. Bruce Eckel and Chuck Allison, *Thinking in C++, 2nd Edition, Volume 2*, Prentice Hall, 2003, ISBN: 0130353132.
15. *Programming in C++, Rules and Recommendations*, Telecommunications Systems Laboratories, 1992.
16. Erich Gamma *et. al.*, *Design Patterns*, Addison-Wesley Professional Computing Series, Boston, New York, 1995.
17. David R. Musser *et. al.,* STL Tutorial and Reference Guide, 2nd Edition*, Addison-Wesley Professional Computing Series, 2001, ISBN: 0201379236.*
18. *P.J. Plauger et. al.,* The C++ Standard Template Library*, Prentice Hall, 2000, ISBN: 0134376331.*
19. *Nicolai Josuttis,* The C++ Standard Library*, Addison-Wesley, 1999, ISBN: 0201379260.*
20. *Nicolai Josuttis and David Vandervoorde,* C++ Templates, The Complete guide*, Addison-Wesley, 2002, ISBN: 0201734842.*
21. *Donald Knuth,* The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*, Addison-Wesley, 1997, ISBN: 0201896834.*
22. _____, The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*, Addison-Wesley, 1997, ISBN: 0201896842.*
23. _____, The Art of Computer Programming, Volume 3: Searching and Sorting (2nd Edition)*, Addison-Wesley, 1999, ISBN: 0201896850.*
24. *Andrew Koenig,* C traps and Pitfalls*, Addison-Wesley, Boston, New York, 1988.*
25. *Andrew Koenig and Barbara Moo,* Accelerated C++*, Addison-Wesley, 2000, ISBN: 020170353X.*
26. *John Lakos,* Mastering C++ Large-Scale C++ Software Design*, Addison-Wesley, 1996, ISBN: 0201633620.*
27. *Angelika Langer and Klaus Kreft,* Standard C++ IOStreams and Locales*, Addison-Wesley, 2000, ISBN: 0201183951.*
28. *Stan Lippman,* Inside the C++ Object Model*, Addison-Wesley, 1996, ISBN: 0201834545.*
29. _____, Essential C++*, Addison-Wesley, 1999, ISBN: 0201485184.*
30. Loki template library*, sourceforge.net/projects/loki-lib/.*
31. *Steve McConnell,* Code Complete*, Microsoft Press, 1993.*
32. *Scott Meyers,* More Effective C++, 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley Professional Computing Series, Boston, New York, 1996, ISBN: 020163371X.*
33. _____, Effective C++, 50 Ways to Improve Your Program Design, Second Edition*, Addison-Wesley Professional Computing Series, Boston, New York, 1998, ISBN: 0201924889.*
34. _____, Effective STL, 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley Professional Computing Series, Boston, New York, 2001, ISBN: 0201749629.*
35. *Robert Sedgewick,* Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching (3rd Edition)*, Addison-Wesley Professional Computing Series, 1998, ISBN: 0201350882.*
36. _____, Algorithms in C++ Part 5: Graph Algorithms (3rd Edition)*, Addison-Wesley Professional Computing Series, 2001, ISBN: 0201361183.*
37. *Bjarne Stroustrup,* Design and Evolution of C++*, Addison-Wesley Professional Computing Series, 1994, ISBN: 0201543303.*
38. _____, The C++ Programming Language, Special Third Edition*, Addison-Wesley Professional Computing Series, Boston, New York, 2000, ISBN: 0201700735.*
39. *Herb Sutter,* Guru of the Week*, www.gotw.ca/gotw.*

40. _____, Using auto_ptr effectively, *www.gotw.ca/publications/using_auto_ptr_effectively.htm, 1999.*

41. _____, Exceptional C++, *Addison-Wesley Professional Computing Series, Boston, New York, 2000, ISBN: 0201615622.*

42. _____, More Exceptional C++, *Addison-Wesley Professional Computing Series, Boston, New York, 2001, ISBN: 020170434X.*

43. _____, The New C++: Smart(er) Pointers, *www.cuj.com/documents/s=7980/cujcexp2008sutter/, 2002.*

E-mail address: clomont@cybernet.com, clomont@math.purdue.edu
URL: www.math.purdue.edu/∼clomont