C++ Coding Standards for EECS 381

Revised 3/24/09

Introduction

Each software organization will have its own coding standards or "style guide" for how code should be written for ease of reading and maintenance. You should expect to have to learn and follow the coding standards for whichever organization you find yourself in. For this course, you must follow this set of coding standards.

The Concept of Single Point of Maintenance

Programs get modified as they are developed, debugged, corrected, and revised to add new features. High-quality code makes modifications easier by having single points of maintenance instead of multiple places in the code that have to be changed.

Why program constants or parameters are named as symbols or const variables: Change the single definition, recompile, and the change takes place everywhere that name is used.

Why functions are used instead of duplicated code. Change the single function, rebuild, and that aspect of the program's behavior changes everywhere that function is used.

Why classes, inheritance, and virtual functions are used in Object-oriented patterns: Everywhere possible interface and implementation is shared, giving large-scale single points of maintenance. Unique code appears only for unique features.

Many of these coding standards support single point of maintenance.

The Concept of Coding Reliability

Many bugs can be prevented by coding in a simple, clear, and consistent style that follows idioms and patterns that experienced programmers have developed.

Many of these coding standards improve coding reliability.

C++ compiler options for this course

Set compiler options to require ANSI/ISO Standard, and pedantic warnings (for gcc).

Numeric types

Avoid declaring or using unsigned integers; they are seriously error prone and have no compensating advantage.

While they can never be negative, nothing prevents computing a value that is mathematically negative, but gets represented as a large positive value instead. Common error: subtracting a larger unsigned value from a smaller unsigned value. Only the absolute value of the result gives any clue that something is wrong.

E.g `for(unsigned int i = 0; i < n; ++i)` is pointless and does nothing helpful and just makes a subtraction error possible if you take the difference between `i` and something else.

If a number should never be negative, either test it explicitly, or document as an invariant with an assertion.

Major exception (not relevant to this course): if bitwise manipulations need to be done (e.g. for hardware control) using unsigned ints for the bit patterns may be more consistent across platforms than signed ints.

To interface with Standard Library functions, declare size_t or string::size_type variables, or cast to/from int; never explicitly declare an unsigned integer type.

The only case for using unsigned integers is to interface with Standard Library functions that return size_t or string::size_type values that traditionally are defined as unsigned to allow a larger possible size. But never declare such variables as "unsigned int"; instead:

Declare and use size_t or string::size_type variables to hold the values. Preferred if many values and variables of the type need to used, as in code using std::string to do lots of string searching and manipulating. Examples:

```
size_t len = strlen(s);
```

```
string::size_type loc = s.find("boo");
```

Or cast between size_t or string::size_type values and int values. Preferred if only a few variables are involved, conceptually the data is really about simple integers, or arithmetic will be done, especially if subtraction will be done. Example:

```
int len = static_cast<int>(strlen(s));

int len = static_cast<int>(s.size());
```

or using a function-style cast:

```
int len = int(strlen(s));
```

Prefer double to float.

Only use of float: if memory space needs to be saved, or required by an API.

Do not assume that two float or double values will compare equal even if mathematically they should.

Only case where they will: small to moderately large integer values have been assigned (as opposed to computed and then assigned).

Otherwise, code should test for a range of values rather than strict equality.

String literal constants

Prefer to declare and define as pointers to constants rather than initialized arrays of char or const std::string variables:

Bad: `const char message[] = "Goodbye, cruel world!";`

Requires extra storage for the array, plus time to copy the literal into the array.

message is an array sized big enough to hold the string which is copied in at initialization, even though the string has already been stored in memory.

Bad: `const std::string msg("Goodbye, cruel world!");`

Requires extra storage for the string's internal array, plus time to allocate the internal array and copy the literal into it. Run-time memory footprint is at least twice as large as the string literal.

Good: `const char * const message = "Goodbye, cruel world!";`

Simply sets a pointer to the string literal already stored in memory.

message is a constant pointer to constant characters - neither the pointer nor the characters can be changed.

Enum types

Prefer to use an enumerated type instead of arbitrary numeric code values.

The names in the enumeration express the meaning directly and clearly.

Do not use an enumeration if the result is greater program complexity.

E.g. translating command strings into enums which are then used to select the relevant code for the command simply doubles the complexity of command-selection code.

Give the type a name starting with an upper-case letter terminated with "_e" or similar.

The names for the values should be all upper case.

Always store and pass enum values in the enum type, even though they convert freely to ints.

Keep it in the enum type to maintain the clear and direct meaning.

Prefer to use the default for how enum values are assigned by the compiler

Bad: `enum Fruit_e {APPLE = 0, ORANGE, PEAR, BANANA};// Why? This is the default!`

Not relying on the default when it is suitable indicates either ignorance or confusion.

Bad: `enum Fruit_e {APPLE = 3, ORANGE, PEAR, BANANA};// Potential fatal confusion!`

There needs to be a VERY GOOD reason to override the compiler-assigned values.

Good: `enum Fruit_e {APPLE, ORANGE, PEAR, BANANA};// Let the compiler keep track!`

Understand and use how I/O works with enum values

Enums are written as integer values

To read an enum value, read an int, check if for the maximum and minimum valid values, and then assign with a cast to the enumerated type.

Names

Take names seriously - they are a major way to communicate your design intent to the future human reader (either yourself or somebody else).

Poor names are a major obstacle to understanding code.

Do not use cute or humorous names, especially if they don't help communicate the purpose of the code.

Bad: `delete victim; // there is no "victim" here`

Better: `delete node_ptr;  // there is a node that we are deleting`

Bad: `zap();    // sure, it's cute, but what does it do?`

Better: `clear_pointers(); // ok - this function clears some pointers.`

Don't start variable or function names or #define symbols with underscores.

Leading underscores are reserved for the implementation - break this rule, and you risk name collisions leading to confusing errors.

Use an initial upper-case name for your own types (enums, classes, structs, typedef names).

e.g. `class Thing`, not `class thing`.

Standard Library symbols are almost all initial lower-case, so this is an easy way to distinguish your types from Standard types.

Distinguish enum names with a final "_e", as in Fruits_e;

The values for an enum type must be all upper case.

Distinguish typedef names with a final "_t", as in Thing_list_t;

Preprocessor symbols defined with #define must be all upper case.

Bad: `#define pi 3.14159265`

Good: `#define PI 3.14159265`

Distinguish names for constants that are declared variables. Choose and maintain a style such as a final "_c" or a leading lower-case 'k' followed by an upper-case letter .

Bad: `ymax = screen_h_size;   // no clue that right-hand-size is a constant`

Good: `const int kScreen_h_size = 1024;`

Good: `const int screen_h_size_c = 1024;`

Good: `const char * const error_msg_c = "Error encountered!";`

Use typedef to provide a more meaningful, shorter, or detail-hiding name for a type.

Little value if the typedef name is as verbose as the type.

Bad: `typedef struct Thing * Thing_struct_ptr_t;`

Good: `typedef struct Thing * Thing_ptr_t;`

Use variable names that do not have to be documented or explained - longer is usually better.

Worst: `x;`

Bad: `bsl;`

Good: `binding_set_list;`

Single letter conventional variable names are OK for very local, temporary purposes.

OK:

```
for(int i = 0; i < n_elements; i++)
     sum = x[i];

y = m * x + b;
```

Don't ever use easily confused single-letter variable names - don't rely on the font!

Lower-case L (l), upper-case i (I) are too easily confused with each other and the digit one.

Similarly with upper-case O and digit zero.

Use upper/lower mixed case or underscores to improve readability of explanatory names.

Bad: `void processallnonzerodata();`

Good: `void ProcessAllNonZeroData();`

Good: `void process_all_non_zero_data();`

Don't include implementation details such as variable type information in variable names - prefer to emphasize purpose instead.

Bad: `int count_int;`

Bad: `const char * ptr_to_const_chars;`

Better: `int count;`

Better: `const char * input_ptr;`

How to tell: What if I need to change the variable type to a similar but different type? E.g. long ints, wide characters. Would it be important to change the variable names to match? If so, implementation details are exposed in the variable names.

A name or symbol for a constant that is a simple synonym for the constant's value is stupid. The purpose of naming constants is to convey their role or meaning independently of their value; often, the same concept might have a different value at some point in the future.

See discussion of Single Point of Maintenance

Bad: `#define TWO 2  // what else would it be? 3? This is stupid!`

Bad: `#define X 4    // what is this? Can't tell from this name!`

Good: `#define MAX_INPUT_SIZE 255  // the maximum input size, currently this value`

Good: `const double aspect_ratio = 16./ 9.; // We can tell what this is!`

Global variables

In this course, global variables can be used only where specifically authorized.

Global variables should never be used simply to avoid defining function parameters.

Experience shows that passing information through parameters and returned values actually simplifies program design and debugging - global variables used for this purpose are a common source of difficult-to-find bugs.

Global variables are acceptable only when they substantially simplify the information handling in a program. Specifically, they are acceptable only when:

Conceptually, only one instance of the variable makes sense - it is holding information that is unique and applicable to the entire program.

E.g. the standard I/O streams are global variables.

They have distinctive and meaningful names.

They are modified only in one or two conceptually obvious places, and are read-only elsewhere.

They are used at widely different points in a function call hierarchy, making passing the values via arguments or returned values extremely cumbersome.

Their linkage is carefully handled to avoid ambiguity and restrict access if possible.

i.e. C++ methodology is followed.

Internal linkage if possible.

Global constants defined as `const` variables at file-scope or externally linked (program-scope) are not global variables - these restrictions do not apply.

Read-only, with a single definition, does not present any maintenance or debugging problem, and helps ensure consistency.

Be sure they are fully non-modifiable - e.g. `const char * const` for pointers to string literals.

If file-scope, best to give them internal linkage.

Declare `static` in C.

Automatically internally linked by default in C++.

If program-scope, follow global variable guidelines:

Put `extern` declarations only in a header file.

Put definition and initialization in a .c or .cpp file that `#includes` the header file.

Macros (Preprocessor)

Do not use macros for anything except include guards and conditional compilation for platform dependences.

Use #if to select code suitable for different platforms - not common, especially in this course.

One case: selecting different includes for TR1 libraries depending on the platform.

All symbols defined with #define must be ALL_UPPER_CASE.

A critical reminder that a macro is involved.

Use variables defined as const instead of #define for program constants.

Note that a file-scope global const variable gets internal linkage by default in C++, so no need to declare them "static".

Use the assert macro liberally to help document invariants and help catch programming errors.

ONLY for programming errors - not for run-time errors to be reported to the user.

One of the few macros useful in C++ programming.

#include `<cassert>` to access it.

Idiomatic C++

Do not use NULL in C++, use plain zero

Work is underway to create a language construct corresponding exactly to the concept of a null pointer. NULL doesn't work any better than just plain zero, and is thus considered an unnecessary use of a macro.

Use the bool type with true/false instead of int with non-zero/zero when you need a true/false variable.

The code expresses your intent much better, and the compiler ensures that anything that could play a true/false role will be correctly and implicitly converted to the bool type.

Bad:: A hangover from C:

```
while(1) { ... }
```

```
int check()
{
      . . .
      if(whatever)
            return 1;
      else
            return 0;
}
```

Good:: We can say what we really mean:

```
while(true) { ... }

bool check()
{
      . . .
      if(whatever)
            return true;
      else
            return false;
}
```

in C++, do not use the "struct" keyword except in the declaration of the struct type itself.

You have to use it in C everywhere you refer to the struct type, but not in C++.

Prefer "typename" instead of "class" in template parameter declarations.

A template parameter doesn't have to be a "class" type, so "typename" is more clear.

Don't use C facilities instead of better C++ facilities:

Do not use scanf/printf family functions in C++.

Do not use malloc/free family functions in C++.

Do not use memset, memmove, memcpy in C++.

Too easy to cause serious problems in construction/destruction processing.

Avoid using the `exit` function in C++.

Unlike the case in C, `exit` in C++ is not equivalent in effect to a return from `main`.

Calling `exit` does not ensure that all relevant destructors are called.

Only valid use: As an emergency exit where leaving a mess is the only alternative.

Preferable: Error conditions should result in exceptions thrown and then caught at the top level of `main` which then returns.

Prefer += for std::string instead of = of + if performance is important.

```
s = s1 + s2;  // requires creating a temporary string to hold s1 + s2;

s1 += s2; // requires only possible expansion of s1 and copying
```

Casts

Never use C-style casts; always use the appropriate C++ -style cast that expresses the intent of the cast.

Casts usually mean the design is bad and using them undermines type-safety; try to correct the design if possible.

E.g. any use of void * type (which always requires casting to be useful) is likely to be a bad design - unless it is in the implementation of a low-level component.

Function style/constructor casts can be used for routine numeric conversions:

OK:

```
int i = static_cast<int>(double_var);
```

also OK:

```
int i = int(double_var);
```

Take advantage of the definition of non-zero as true, zero as false, when testing pointers.

Clumsy:

```
if(ptr != 0) or if(ptr == 0)
```

Better:

```
if(ptr) or if(!ptr)
```

Write for *statements in their conventional form if possible.*

Good - the conventional, most common form:

```
for(i = 0; i < n; i++)  // correct for almost all cases
```

Bad:

```
for(i = 1; i <= n; i++) // confusing - what is this about?
for(i = n; i > 0; i--)  // better be a good reason for this!
for(i = -1; i <= n; i++)    // totally confusing!
```

In `for` loops with iterators, prefer the pre-increment operator, rather than post-increment.

Bad: `for(list<Thing>::iterator it = things.begin(); it != things.end(); it++)`

Good: `for(list<Thing>::iterator it = things.begin(); it != things.end(); ++it)`

Why: Post-increment must copy and hold a value for returning; if unused, the compiler may not be able to optimize it away.

Catch exceptions by reference, not by value.

The throw operator creates a copy of the thrown object into a special area outside of the function call stack so that it remains available while the stack is unwound. No point (and possible performance issues) if it gets copied again.

Bad:

```
catch (Error x) { ... }
```

Good:

```
catch (Error& x) { ... }
```

Don't use `this` in member function code unnecessarily.

In member functions, the compiler automatically converts appearances of member variables to accesses through the `this` pointer, and calls to member functions of the same class to calls through the `this` pointer. Explicitly writing out `this->`or `(*this).` for such purposes is just duplicating the compiler's work and cluttering the code - not to mention looking ignorant of what C++ does automatically.

Reserve use of the `this` pointer for cases where you actually do need to supply a pointer to "this" object, which usually only happens in a call to a function in another class or as a return value. Prefer statements in which the compiler-supplied `this->` is used instead of an explicit reference to `*this`.

Bad:

```
// Why duplicate the compiler's work or complicate the reader's task?
void Thing::foo()
{
      this->x = this->y + this->z;
      this->zap(this->x);
}

Thing& Thing::operator= (const Thing& rhs)
{
      . . .
      temp.swap(*this);// not idiomatic; confusing
      . . .
}
```

Good - `this` explicitly used only where needed:

```
void Thing::foo()
```

```
{
      x = y + z;
      zap(x);
}

Thing& Thing::operator= (const Thing& rhs)
{
      . . .
      swap(temp);
      return *this;    // return a reference to "this" object
}

void Warship::attack(Ship * target)
{
      . . .
      target->receive_hit(this, firepower);  // tell target "this" ship has fired at it
}
```

Const correctness

Declare everything const that is meaningful and correct to be const.

Do so from the beginning of a project to avoid "viral" nuisance in modifying existing code.

If something turns out to be non-const, correct the design rather than patch the error.

E.g. avoid using const_cast to get around having made something const that really isn't.

Don't treat things as const that actually change.

If the design concept is that X changes when Y happens, don't say that X is const!

Don't misuse `mutable` to fake constness. Reserve const member functions that modify a mutable member variable for situations in which you can improve the speed of the function without changing its other visible, logical, or conceptual behavior.

Any other use of `mutable` in this course is almost certainly a serious design failure.

Distinguish between constness of a container, the objects in the container, and objects pointed to.

If the container is modifiable,

and contains const items, you can add or remove items, but not change the value of one of the items that is in the container.

and contains pointers to const objects, you can add, remove, or change the pointers, but not change the objects being pointed to using the contents of the container.

If the container is constant,

you can't add or remove items in the container, or change the value of one of the items that is in the container.

and contains pointers to non-const objects, you can't add, remove, or change the pointers, but you can change the objects being pointed to using the contents of the container.

Avoid using std::set for objects that must be changed, unless you can implement the change honestly - without const_cast or faking const member functions with mutable.

Copy the object in the set, remove it from the set, change the copy, and put the changed copy into the set.

If a non-key value must be changable, consider use a set of pointers to non-const objects.

Designing and declaring classes

A class should have a limited, clear, and easily stated set of responsibilities.

Corresponding to a single concept.

If you can't explain what a class does in a few coherent sentences, suspect that the basic design is flawed and needs correction.

If a class does several things that aren't closely related, suspect that the class is bloated - it tries to do too much; perhaps additional classes are needed to handle the other responsibilities.

Suspect designs in which a class and its client do similar work. Either the client or the class should be responsible for the work, not both.

As shown by similar code in both - suspect that the responsibilities have not been properly represented in the class.

Example: Both client and class make the same distinctions between possible situations and so use similar code to test or switch on the situation.

Suspect designs in which only one object of the class will exist in the program.

Good design in C++ does NOT require that all the code be in classes!

Usually, the purpose of a class is to describe a kind of object, of which there will be multiple instances, not a single unique object.

Why is there only one object? Two *good* reasons:

Providing a single unique object is the purpose of the class - as in the Singleton or Model-View-Controller patterns, which are easily recognized when properly coded and documented.

The class design provides for extensibility by using virtual functions in a base class to specify an interface that can be implemented with different derived classes in the future - a basic technique for code extensibility. Single objects of derived classes will be used to implement easily changed functionality, often at run time - e.g. in the Strategy, State, or Abstract Factory patterns. Only used when inheritance and virtual functions are appropriate - not for "concrete classes."

Why is there only one object? One *bad* reason that is common among beginning programmers:

The class does too much, so much that only a single object of its type is needed because it tries to does *everything* related to its purpose.

Favor designs in which the class provides limited and general functionality, and let the specific details be handled by the client code using multiple objects of the class.

In a class declaration, list public members first, followed by protected members, followed by private members.

Most readers just need to see the public interface - put it first for them. Rest is implementer's business.

If the class can't be used without understanding its private members, the design is probably defective and should be fixed.

Public member functions are the interface for clients of the class. Make public only those functions that clients can meaningfully use.

All other member functions should be private or protected.

Functions required by the implementation should be private helper functions.

If possible, scope typedefs or enums within a class declaration rather than declare at the top level of a header file.

If used in a class implementation only, declare them in the private part of the class declaration.

If clients must have access to them, declare them in the public part of the class declaration.

If possible, keep "helper" class or struct declarations out of the header file; if not possible, put them inside a class declaration rather than declare at the top level of a header file.

A header file should contain the minimum necessary for a client to use the module - anything that could be put in the .cpp file should be.

If the helper is used in a class implementation only, declare it in the private part of the class declaration.

If clients must have access to helpers, declare them in the public part of the class declaration.

Consider friend functions or top-level friend classes to be part of the public interface for a class.

Provides services to clients that can't be provided by member functions.

Input and output operators are a common example.

Key role of friends is to help maintain encapsulation - e.g. a friend function makes it unnecessary to have public reader/writer functions that would undermine encapsulation.

Avoid granting friendship to functions or classes that are developed or maintained by a different programmer or group.

Because friends have access to the implementation details, and so depend on them, they should be considered part of the class and developed and maintained along with the class. Otherwise, severe communication, maintenance, and debugging problems will result.

Prefer to use "struct" instead of "class" for a simple concrete type all of whose members are conceptually public, and do not use the "public" or "private" keywords in the declaration.

Especially appropriate if the type is going to be used the same way as a struct in C.

Can be appropriate even if the type has constructors, member functions, and operators - as long as all members are conceptually public.

All member variables of a class must be private.

Essential to separating interface from implementation, and making separation and delegation of responsibility possible.

The need to make some member variables public is almost certainly a result of a design error.

If this seems incorrect, maybe all members should be public - maybe struct would be better instead of class.

Suspect designs in which all (or most) private members have getter/setter functions.

If values must be available or controllable by clients, the design of the class is probably bad - it isn't taking responsibility for the work - clients are doing it instead.

Do not provide functions that return non-const references or pointers to private members.

Breaks encapsulation by making it possible for clients to modify member variables directly.

Returning a reference-to-const can be used to avoid copying a returned value - faster.

If function is a const member function, then compiler will reject certain non-const return values, but not others - depending on the return type and member variable type. So to be sure, declare the return type as a reference or pointer to const and also declare the function as const.

Bad: (compiler may not warn or flag as an error)

```
char * get_ptr() const {return ptrmember;}
int *  get_int() {return &intmember;}
int &  get_int() {return intmember;}
```

Good:

```
const char * get_ptr() const {return ptrmember;}
const int *  get_int() const {return &intmember;}
const int &  get_int() const {return intmember;}
```

Define simple functions like getter/setter functions in the class declaration to enable inlining.

Definition in a class declaration is a request to the compiler to inline the function.

Note that inlined function code must be visible to the compiler, so needs to be in the header file.

If function is complex, define it in the .cpp file, not in the class declaration.

Inlining is not necessarily a good idea due to possible code bloat.

The code will clutter the class declaration unnecessarily.

The constructor for a class should ensure that all member variables have meaningful initial values.

As much as possible, initialize member variables in the constructor initializer list rather than in the constructor body.

Note that class type member variables will get default construction automatically if initial values are not specified.

Complex operations such as allocating memory should be placed in the constructor body.

If the compiler-supplied constructor or constructor call correctly initializes an object, let it do so - do not write code that duplicates what the compiler gives you automatically.

You have to understand what the compiler will and won't do for you.

Less code to write means less code to read, debug, and maintain.

If construction can fail, use an exception to inform the client code.

Avoid designs in which the client has to inspect an object to see if it has been validly constructed. Such "zombie" objects are difficult and unreliable to work with.

Note that the throw will cause the object-under-construction to fall out of scope - any member variables constructed prior to the throw will be destructed, and if the object is being allocated with new, the memory will be automatically deallocated. Thus the object does not exist after the throw takes effect.

If you have a single-argument constructor, prefer to define the default constructor using a default parameter value in the same constructor.

OK:

```
class Thing {
public:
	Thing() : i(0) {}
	Thing(int i_) : i(i_) {}
private:
	int i;
};
```

Better:

```
class Thing {
public:
	Thing(int i_ = 0) : i(i_) {}
private:
	int i;
};
```

Mark single-argument constructors as explicit unless allowing implicit conversion from the argument type is part of the design.

Dubious:

```
class Thing {
public:
	Thing(int i_ = 0) : i(i_) {}
private:
	int i;
};

void foo(Thing t);
...
foo(2);    // implicit conversion from an int to a Thing - do you really mean for this to make
sense?
```

Better:

```
class Thing {
```

```
public:
     explicit Thing(int i_ = 0) : i(i_) {}
private:
     int i;
};

void foo(Thing t);
...
foo(Thing(2));   // no implicit conversion allowed
```

When choosing overloaded operators for a class, only overload those operators whose conventional (built-in) meanings are conceptually similar to the operations to be done. Prefer named functions otherwise.

Good example: std::string overloaded operators.

Bad example: What could thing1%thing2 possibly mean?

Member functions that provide services meaningful only to derived classes should be declared as protected.

Conveys that they aren't part of the public interface.

Declare member functions const if they do not modify the state of the object.

If a member function doesn't modify the logical state of an object, but does modify a member variable to produce better performance (e.g. a cache scheme of some sort), declare the member function `const` and the member variable to be `mutable`. See section on const-correctness.

Any other use of `mutable` in this course is almost certainly a serious design failure.

Understand which member functions the compiler will create for you and what they do. Do not write the default constructor, destructor, copy constructor, or assignment operator if the compiler-supplied one is correct and meaningful.

Unnecessary work is an unnecessary source of bugs.

Explicitly decide whether default construction is meaningful and required and provide it if so.

Certain containers require a default constructor for their content objects.

If you have declared a constructor with a parameter, the compiler will not create a default constructor; if it is needed, you have to explicitly declare and define it.

A constructor with a single parameter that has a default value will be used as a default constructor.

Can be called with no arguments!

Explicitly decide whether copying and assignment is meaningful and required and rule them out if not.

Certain containers require copy and assignment of their content objects.

Declare copy and assignment functions to be private if not needed.

Normal declarations:

```
Classname(const Classname&);                // copy constructor

Classname& operator= (const Classname&);   // assignment operator
```

Explicitly decide whether you need to write a copy constructor and assignment operator and implement them if so.

Rule of the "big three" - if you had to write a destructor, then you almost certainly need to write copy or assignment, unless they are not meaningful and can be ruled out.

In writing a copy constructor, remember to initialize all member variables - a common error.

Designing functions

Use functions freely to improve the clarity and organization of the code.

Modern machines are very efficient for function calls, so avoiding function calls is rarely required for performance.

If it is, prefer inline functions to get both performance and clarity.

Define functions that correspond to the conceptual pieces of work to be done, even if only called once or from one place.

Clarify the code structure, making coding, debugging, maintenance, easier.

E.g. in a spell-checking program, create a function that processes a document by calling a function that processes a line of the document that in turn calls a function that finds each word in the line.

Use functions to avoid duplicating code.

Copy-paste coding means copy-pasting bugs and multiplying debugging and modification effort.

Concept: *Single point of maintenance.* If you have to debug or modify, you want one place to do it.

How do you tell whether duplicated code should be turned into a function? Move duplicated code into a function if:

The code is non-trivial - getting a single point of maintenance is likely to be worthwhile.

What the code does can be separated from the context - you can write a function with simple parameters and return value that does the work for each place the duplicated code appears.

The result is less total code with the complexity appearing only in a single place - the function - giving a single point of maintenance.

If functions are used (or should be used) only within a module, give them internal linkage and keep them out of the header file if possible.

Header file is reserved for public interface; non-member functions should not be declared in the header file unless they are part of the public interface for the module.

Can use unnamed namespace instead of static declaration in the .cpp file.

To tell the compiler you aren't using a function parameter in a definition, leave out its name.

```
void foo(int i, double x) { code that doesn't use x} - gets a warning about unused x
```

```
void foo(int i, double) { code } - NO warning about unused second parameter
```

Prefer to use overloaded functions instead of different function names to designate arguments of different types.

Bad:

```
set_Thing_with_int(int i); set_Thing_with_string(const string& s);
```

Good:

```
set_Thing(int i); set_Thing(const string& s);
```

Avoid Swiss-Army functions that do different things based on a switching parameter.

Bad:

```
void use_tool(/* parameters */, int operation)
{
      if(operation == 1)
            /* act like a corkscrew */
      else if(operation == 2)
            /* act like a screwdriver */
      else if(operation == 3)
             /* act like a big knife */
      else if(operation == 4)
             /* act like a small knife */
etc
}
```

The problem is that you can't tell by reading the call what is going on - you have to know how the switching parameter works, and what the other parameters mean depending on the switching parameter, etc. Separate functions for separate operations are usually better.

- especially bad if the resulting code is almost as long or longer than separate functions with good names would be.
- using an enum for the switching parameter helps only slightly because it clutters the rest of the program.

Could be justified if:

- the switch parameter is very simple (like true/false only)
- the behavior controlled by the switching parameter is conceptually very simple (like turning output on or off)
- the switched-function is considerably smaller, simpler, and re-uses code much better than separate functions would do.
- the function call is always commented with an explanation of the switching parameter value

For "input arguments" whose values are not supposed to be changed by a function:

If the argument is a built-in type, simply use the built-in type (not pointer or reference to a built-in type, or even const built-in type).

Bad:

```
void foo(const int& i); void foo (const int * const ip);
```

Good:

```
void foo(int i);
```

Rationale:

- Call-by-value for built-in types is simpler and faster than call-by -reference or -pointer.
- const is redundant because function can't change caller's value anyway.

If the argument type involves complex construction (e.g. std::string), use a reference-to-const input parameter.

Bad:

```
void foo(std::string s);
```

Good:

```
void foo(const std::string& s);
```

If the caller's argument is a pointer, the input parameter should be pointer-to-const, not the pointed-to type.

Bad:

```
void foo(Thing t);// forces a dereference in the call, and possibly unnecessary construction.
```

Good:

```
void foo(const Thing * p);  // since we are referring to things by pointer anyway.
```

For "output arguments" where the function returns a value in one of the caller's arguments in addition to the value returned by the function:

For overloaded operators that modify the caller's argument, a reference parameter is often required.

- Example: overloaded output operator - the stream parameter must be a reference parameter because the stream object gets modified during the output. A proper Standard Library will not even allow a string object to be copied for a call-by-value.
- References added to the language to allow simple syntax in these cases.

But for ordinary functions, a pointer or a reference parameter could be used to return another value. Which one? No clear consensus, but here is some guidance:

In an ordinary function, if you assume that the programmer is following the guidelines for input parameters, then the appearance of a pointer argument conveys very clearly that the caller's object is going to be modified - why else would a pointer be supplied? However, using a pointer argument is more verbose and error prone.

Good:

```
Thing t;
if(process(&t)) {      // obvious that t is going to be modified by the function
      ...
      }

bool process(Thing * thing_ptr)
{
...
}
```

In an ordinary function, you can't tell just from the syntax of a call whether a reference parameter is involved - no hint at all. However, a reference parameter can make the code simpler and still be comprehensible if the name of the function tells you what's going on well enough that you don't have to study the function prototype or code to tell that the caller's argument will be modified.

Poor:

```
Thing t;
if(process(t)) { // not obvious what happens to t - could be an input-only parameter
      ...
      }

bool process(Thing & thing)  // hmm - by reference - so thing must get modified
{
...
      thing = ...// yup, it does get modified
}
```

Good:

```
Thing t;
if(update_Thing_data(t)) {  // obviously, t must get modified by update!
      ...
      }

bool update_Thing_data(Thing& thing);  // function must modify Thing
// but I already knew that from the name
```

Code structure

Put function prototypes or struct/class declarations in the header file if they are part of the module interface, or at the beginning of the implementation file if not.

This ensures that the function definitions can appear in a meaningful and human-readable order - e.g. from top-level down to lowest-level.
See Layout discussion.

Declare variables in the narrowest scope possible, and at the point where they can be given their first useful value.

If a variable will be assigned to its first useful value, declare the variable at that point.

If a variable will be given its first useful value in an input statement, declare the variable just before the input statement.
Note that {} defines a new scope wherever it appears.

Declare variables within the for/if/while block if possible.

Declare variables whose type involves complex default construction (e.g. std::string) at a point where the construction is not wasted (e.g. before the body of a loop)

Bad:

```
for(int i = 0; i < n_big; i++) {
     string s;  // ouch - default construct every time through the loop
     cin >> s;
     ...
```

Better:

```
string s;
for(int i = 0; i < n_big; i++) {
     cin >> s;
     ...
```

Understand the default constructor of a complex type and trust it to properly initialize the variable.

Bad:

```
std::string s = "";     // simply duplicates work done by the constructor.
```

Good:

```
std::string s;
```

Use a default-constructed unnamed variable if you need to set a variable to an "empty" or default value that can't be assigned directly.

```
// set thing to contain  default-constructed Thing object
thing = Thing();
// valid, but not idiomatic (except in instantiated template code)
i = int();
```

Prefer "flat" code to deeply nested code.

Deeply nested code is hard to read and fragile to modify. Prefer a code organization of a "flat" series of condition-controlled short code segments. This will usually require re-thinking the logic, but the result is simpler and easier to work with.

Bad:

```
if(...) {
     ...
     if(...) {
          ...
          if(...) {
               ...
               if(...) {
                    ...
                    }
               ...
```

```
                  }
            ...
            }
      ...   // I'm lost - just when does this code execute?
      }
```

Better:

```
if(...) {
      ...
      }
else if(...) {
      ...
      }
else if (...) {
      ...
      }
etc
```

The "single point of return" guideline usually results in deeply nested conditional code. Such code can usually be rewritten as a simple series of conditionals each controlling a block of code that ends with a return. This works especially well if the conditions are checking for error situations.

Usually good:

```
if(...) {
      ...
      return;
      }
if(...) {
      ...
      return;
      }
if (...) {
      ...
      return;
      }
...
return;
```

Prefer using a switch statement to if-else-if constructions for selecting actions depending on the value of a single variable.

Generally results in simpler, clearer, and faster code than the equivalent series of if-else-if statements.

Exceptions: switch statement cannot be used if strings or floating point values are being tested.

Not a good choice if ranges of integer values are being tested.

Always include a default case with an appropriate action (e.g. an error message or assertion).

Terminate each case with a break statement unless you deliberately want to arrange "drop through" to the next case; if so, you must comment on it.

Arrange iterative or performance-critical code to minimize function calls that might not be optimized away.

Be aware of what iterative code implies ... what has to be done each time around a loop?

Often, the compiler cannot tell whether the code will result in a function computing a different value during execution, and so will not attempt to replace multiple calls with a single call.

Bad: strlen gets called every time around the loop - and what does it do?

```
void make_upper(char * s)
{
      for(size_t i = 0; i < strlen(s); i++)
            s[i] = toupper(s[i]);
}
```

Better: compute the length of the string only once before starting the loop.

```
void make_upper(char * s)
```

```
{
      size_t n = strlen(s);
      for(size_t i = 0; i < n; i++)
            s[i] = toupper(s[i]);
}
```

Best - take advantage of how C-strings work - no need to compute length of string; we have to access each character anyway, so just stop at the end.

```
void make_upper(char * s)
{
      for(; *s; s++)
            *s = toupper(*s);
}
```

or

```
void make_upper(char * s)
{
      while(*s = toupper(*s))
            s++;
}
```

Organize input loops so that there is a only a single input statement, and its success/fail status controls the loop, and the results of the input are used only if the input operation was successful.

In C/C++ the idiom is to place the input operation in the condition of a while loop.

Do not control an input loop based only on detecting EOF.

Input might have failed for some other reason, but bogus results will still be used, and EOF may never happen in the situation.

Bad:

```
while(!infile.eof()) {
     infile >> x;
     // use x;
}
// garbage value of x might get used, loop might never terminate!
```

Good: Use data only if read was successful; diagnose situation if not:

```
while (infile >> x) {
     /* use x */
     }
if(!infile.eof())
     /* not end of file - something else was wrong */
```

If only character or string data is being read, normally only EOF will cause the input to fail, so separate check to diagnose EOF is optional.

Ensure that input or data brought into the program cannot overflow an array or memory block in which it is to be stored.

A basic and essential security and reliability precaution.

Assume user or file input can contain arbitrarily long random strings, and write code that can handle it safely and reliably, even if it simply ignores over-length input.

Prefer length-safe input facilities, such as inputting into a std::string.

Using the Standard Library

Don't recode the wheel - know and use the Standard Library classes and functions.

You can assume that the Standard Library is well-debugged and optimized for the platform.

If it seems at all likely that another programmer has needed what you need, look it up and see if it is in the Standard Library.

Unnecessary DIY coding wastes both coding time and debugging time.

E.g. why write, test, and debug code that reads characters until the first non-whitespace character and then reads and stores it, when cin >> char_var; will do it for you?

If there is a reason why the obvious Standard Library facility can not be used, comment your own function with an explanation.

Understand what Standard Library facilities do, and trust them to do it correctly.

Don't waste time writing code that only makes sense if the Standard Library is defective.

Bad:

```
std::string s = ""; // let's make sure the string is empty!
cin >> s;
if(cin.fail()) // check and return an error code just in case this failed somehow
      return 1;
if(s.size() <= 0)      // check that we read some characters,
      return 1;
// looks like we can use the contents of s now
```

Good:

```
std::string s;   // automatically initialized to empty
cin >> s;  // will always succeed in this course unless something is grossly wrong
// s is good to go
```

Do not write functions that simply wrap a Standard Library function.

Assume that your reader is (or should be) familiar with the Standard Library; this means that the Standard Library function will be more comprehensible than figuring out your particular function that does little or nothing more than call the Standard function.

Bad:

```
/* with reader's comments comments shown */
...
      int i;
      if(read_int(i)) {      /* uh ... exactly what does that function do? */
...
/* let's find the function definition and check it out */

bool read_int(int & ir) {
      cin >> ir;
      return !cin;
}
/* gee - doesn't really do anything! */
/* why did the programmer bother with this function? */
```

Good:

```
...
      int i;
      if(cin >> i) {   /* no problem understanding this */
...
```

Do not use the memmove/memcpy/memset family of functions in this course.

Unless the rest of the code is completely free of inefficiency - "lipstick on a pig" otherwise.

Be aware that these functions can completely destroy or garble the internal structure of an object in the affected memory - they can be extremely dangerous when used on anything except raw memory containing only built-in type data.

Design the run-time error handling policy for a program.

The run-time errors discussed in this guideline are caused by events outside of the programmer's control, but must be handled well by the programmer to produce robust, dependable software.

Not programming errors that are defects in logic and coding due to mistakes made by the programmer, but events in the program's run-time environment - outside the program and thus beyond the ability of the programmer to control. Examples:

The user enters an invalid command.

There is garbage in a data file.

The system runs out of memory or some other resource.

Network connections disappear.

Explicitly design what a program will do in case of errors.

Do not let error handling policies develop haphazardly as a result of random thoughts while coding.

Avoid designs in which the program simply "muddles through" and attempts to keep going in the presence of run-time errors. It is almost always better to take positive action to either inform the user and/or stop processing and restore to a known state before resuming.

Use exceptions to allow a clear and simple separation of error and non-error flow of control, and to clearly assign responsibility for error detection and error handling.

Allows uncluttered "normal" flow of control, and clear error-handling flow of control.

The code that can best detect an error is usually not the place where the error can be best recovered from.

Do not use exceptions for a "normal" flow of control.

Exception implementations are too inefficient for that purpose; reserve them for true error situations where the program processing has to be stopped, terminated, or redirected in some way.

Be aware of the C/C++ philosophy: For faster run speed, the language and Standard components do not do any checks for run-time errors; instead, your code is expected to ensure that an operation is valid before performing it - either by selective checks or careful code design.

Fast run time performance but at the expense of programmer care and effort.

Examples:

Dereferencing a zero pointer - check the pointer for non-zero before dereferencing it, but only if there is any possibility that it might be zero. Some algorithms require such a check, but ideally, the code will be written so that it can't happen.

Accessing the front or top element in an empty list or queue - write the code so that this will never happen - for example, by using the empty() function to check first.

Calling strcpy to copy a C-string to a destination that is too small - write the code so that the destination is guaranteed to be large enough, or use std::string if the guarantee is impractical.

Following a dangling pointer - often no simple check, so must design the code so that it will never happen. Alternatively, use a smart pointer that provides a guarantee that the object is still present, or a way to check whether it is or not.

Using an invalid STL iterator - write the code so that the iterator is guaranteed to be valid.

Using dynamically allocated memory (new/delete)

Where possible, use "automatic" function-local variables. Do not allocate memory with new if a local variable or array will work just as well.

In Standard C++, the new operator with throw a bad_alloc exception, so no check of the returned pointer value is needed.

If the exception is not caught, then as is the case for all uncaught exceptions, the program will be immediately terminated.

Design your code with a clear and explicit policy that states where and when the call to delete will be for every call to new.

Attempt to write the deallocation code immediately after the allocation code to avoid forgetting it.

Remember to use delete[] - the array form - if you allocated an array with new.

In this course, all allocated memory must be deallocated by the program before terminating.

Represents the "clean up before quitting" philosophy - a good practice even if often not strictly necessary in modern OS environments.

Program must terminate with a return from `main` which completes deallocation of all memory.

In high-quality code, class destructor functions will perform much of the cleanup automatically.

Header files should be a minimal declaration of the module interface.

See the header file guidelines document for more discussion and detail.

Program modules or re-usable components consist of a header (.h) file and an implementation (.cpp) file.

The header file should contain exactly the interface declarations required for another module (the client) to use the module or component, and no more.

Any declarations or definitions not strictly required as part of the interface should be in the implementation file, not the header file.

It is a problem in C++ that a class declaration in the header file exposes at least the names and types private members, but everything else that is not part of the public interface must be kept out of the header file.

Arrange to have the minimum number of #includes in a header file.

Use forward/incomplete declarations of pointer types instead of #includes if possible.

The header file should be complete; it should compile correctly by itself.

Create a .cpp file that contains nothing but an #include of the header file. This file should compile without errors.

Guidelines for #including header files in an implementation file.

The first #include in a .cpp file for a module should be the corresponding header file.

Project-specific includes (using double quotes) should appear before Standard Library or system includes (with angle brackets).

Always ensure that the relevant Standard Library header gets included even if the code happens to compile without it.

This prevents platform-specific compile failures due to how the Standard doesn't say which Library headers have to include which other Library headers.

Do not #include unnecessary header files.

Causes serious problems with spurious coupling and slower compile times.

`using` *statements must appear only after all #includes.*

To prevent changing how #includes get processed.

In C++, include a C Standard Library Header by using its C++ name, not the C name:

<cstring> not <string.h>

<cassert> not <assert.h>

<cmath> not <math.h>

Follow guidelines for `using` statements.

No using declarations or directives are allowed in a header file.

Only possible exception: if scoped within an inline or member function body or a class declaration.

In .cpp files,

Place using statements only after all #includes.

Prefer using declarations of specific Standard Library functions or classes to using namespace directives.

Especially in this course, prefer using declarations or directives to explicitly qualifying Standard Library names with "std::".

Using a project Utilities module

Place in the Utilities module only functions or declarations that are used by more than one module.

Examples: A typedef used throughout a project; a function that compares two struct type variables that is needed in two modules.

Secondarily, place in the Utilities module functions that are generic and would be generally useful in related projects.

Examples: a function to convert 12-hour time to 24-hour time; a function to produce a lower-cased copy of a string.

Negative example: a function that isolates words in a string following the rules for a particular spell-checking implementation.

Do NOT use the Utilities module as a dumping ground for miscellaneous scraps of code - it is reserved for the above two uses.

Project-specific code used by only one module should never be placed in the Utilities module.

Layout

Arrange function definitions in a .cpp file in a human-readable order corresponding to the top-down functional decomposition or usage order of the module.

The reader should be able to read the code in increasing order of detail to take advantage of the information-hiding value of functions. So the root(s) for the function call tree should be the first functions listed; leaf functions called only from one branch should appear before the start of the next branch; leaf functions called from all branches should appear last.

Don't make the reader rummage through the file trying to find functions listed in a haphazard order.

If a function object class is used only in a .cpp file, and for a purely local purpose, place its declaration/definition in the .cpp file immediately before the first function that uses it.

Do not put at the beginning of the .cpp file - the programmer will just have to rummage for it when reading the code that appears later.

Do not put in the header file - not part of the public interface for the module.

Use a consistent indenting scheme and curly brace scheme.

Imitating Kernigan & Ritchie or Stroustrup is certainly one good approach.

Avoid excessively long lines - 80 characters is a traditional value.

If lines won't fit on standard paper when printed in 10 pt font, probably too long.

Especially bad: long lines due to excessively nested code, which has other serious problems.

Be careful with leaving out optional curly braces, especially with if.

Clear: a simple thing that also looks simple:

```
if(x == 3)
      foo(x);
```

But if we later add some more code to the if, it is just too easy to write:

```
if(x == 3)
      foo(x);
      zap();     /* uh ... why doesn't it work right? */
```

Uglier but more reliable when coding late at night:

```
if(x == 3) {
      foo(x);
      }
```

Comments

See the posted article on comments for more discussion and examples.

Keep comments up-to-date; at least annotate or delete them if they are no longer valid.

Obsolete comments suggest sloppy coding at best, and are often worse than none at all because they confuse and mislead, and cast doubt on all of the other comments. Out-of-date comments will be considered a major failure of code quality.

Comments should never simply paraphrase the code.

You should assume that the reader knows the language at least as well as you do. The purpose of comments is to explain aspects of the code that will not be obvious to an experienced programmer just by looking at it.

Each function prototype in a header file, and function definition in a .cpp file, should be preceded by a comment that states the purpose of the function and explains what it does.

The function name and parameter names should well chosen, which will help explain how the function is used. If a value is returned, it is important to explain how it is determined - this will usually be less obvious than the role of well-named parameters.

In a .cpp file that has a block of function prototypes at the beginning, comments are not required for the function prototypes, but are required on the function definitions.

The initial prototypes are declarations for the compiler, and enable the functions to be defined in a readable order, but the prototypes are inconveniently located for the human reader - comments there are wasted.

The purpose of constants should be commented, especially if they may need to be changed.

E.g. a constant for the maximum length of an input line.

Comments should appear within a function to explain code whose purpose or operation is obscure or just not obvious.

Comments should explain what is being done where the code will be less than completely obvious to the reader. A common student error is comment simple code, but then make no comment at all to explain a chunk of difficult and complicated code that obviously took a lot of work to get right. If it was hard for you to write, it will be hard for a reader to understand!