C++ Coding Standards for EECS 381

Revised 12/28/2015

- **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.*
- **The Concept of Not Looking Clueless**
  - ** Most of these coding standards are based on the wisdom of the "gurus" of programming. But some of them address common student errors that professionals would rarely make, and so are unlikely to appear in industry standards. The worst of these are marked with a leading asterisk (like this item). If you don't want to look clueless, you should especially pay attention to these items.*
- **C++ compiler options for this course**
  - *Set compiler options to require ISO Standard C++14*
  - *for gcc 5.1.0, LLVM clang 6.1 and higher use: -std=c++14 -pedantic-errors*
- **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));`
- *Use double, not float.*
  - doubles are precise enough for serious numerical computation; float may not be.
  - 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.
  - Accurate floating-point computation is a highly technical field, not really addressed by this course. Don't assume you know anything about it until you have studied it.

- **Enum types**
  - *C++11 has enum classes in which the enumerated values are scoped, similar to wrapping an C-style enum inside a class declaration, but with no implicit conversions. Enum classes are much better behaved than C-style enums, and should be used instead if possible. Many of the Coding Standards for C-style enums (see the C Coding Standards) do not apply to enum classes.*
    - No implicit conversion means that i/O has to be done with explicit conversions either involving the underlying integer type, or using code that explicitly maps between input/output values and enum class values.
    - The name of the enum value is scoped in the enum class, so the traditional all-upper-case names used in C enums are unnecessary and distracting; use names that are clear and simple instead
  - *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 or an increased number of points of maintenance.
      - 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, and means that more things have to be added iadditional commands are added.
      - A solution with simpler code and single points of maintenance is better.
  - *Use the default for how enum values are assigned by the compiler*

- Bad: `enum class Fruit {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 class Fruit {Apple = 3, Orange = 1, Pear, Banana};`
  `// Potential fatal confusion!`
  - There needs to be a VERY GOOD reason to override the compiler-assigned values.
- Good: `enum class Fruit {Apple, Orange, Pear, Banana};// Let the compiler keep track!`

- **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.
  - ** 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.
    - The initial upper-case makes the distinction between type names and variable names obvious.
  - ** Don't start variable or function names or #define symbols with underscores.*
    - Leading underscores are reserved for the C/C++ preprocessor, compiler, and library implementation internal symbols and names - break this rule, and you risk name collisions leading to confusing errors.
    - Yes, some "authorities" break this rule, but once you've seen the mess from a name collision, you'll be glad you followed it!
    - Note: The actual rule on reserved leading underscore names is somewhat complex; it is simplified here because there is no good reason to take a chance by pushing the envelope.
  - *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.`
  - *Preprocessor symbols defined with #define must be all upper case.*
    - Bad: `#define thing_h`
    - Good: `#define THING_H`
  - *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-side 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 or type aliases (better) to provide a more meaningful, shorter, or detail-hiding name for a type.*
    - Little value if the typedef/alias name is as verbose as the type.
    - Bad: `typedef struct Thing * Thing_struct_ptr_t;`

- Good: `typedef struct Thing * Thing_ptr_t;`
- Better: `using Thing_ptr_t = struct Thing *;`

- *Distinguish typedef or type alias names with a final "_t", as in Thing_list_t;*
- *Typedef/alias the container type, not the iterator type for simplicity and improved readability:*
  - Not helpful:
    ```
    std::map<std::string, int> lookup_map;
    typedef std::map<std::string, int>::iterator Lookup_map_iterator_t;
    Lookup_map_iterator_t it = lookup_map.find(x);
    ```
  - Good:
    ```
    typedef std::map<std::string, int> Lookup_map_t;
    Lookup_map_t lookup_map;
    Lookup_map_t::iterator it = lookup_map.find(x);
    ```
  - Better:
    ```
    using Lookup_map_t = std::map<std::string, int>;
    Lookup_map_t lookup_map;
    Lookup_map_t::iterator it = lookup_map.find(x);
    ```
  - Best: Use auto where possible:
    ```
    using Lookup_map_t = std::map<std::string, int>;
    Lookup_map_t lookup_map;
    auto it = lookup_map.find(x);
    ```
- *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; // a traditional equation for a line
    ```
- *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 - 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.

- **Named constants and "magic numbers"**
  - ** Numerical constants that are "hard coded" or "magic numbers" that are written directly in the code are almost always a bad idea, especially if they appear more than once.*
    - No single point of maintenance: If the value needs to be changed, you have to try to find every appearance in the code and fix it. If the value isn't unique or distinctive and appears many times, you are almost guaranteed to mess up a change!
      - E.g. array sizes are especially important - often need to be changed as a group.
      - Bad:
        - char buffer[64];
          ...
          char name[64];
          ...
          ... = new char[64]; /* allocate memory for an input string •/
          ...
      - Good:
        - const int input_size_c = 64 /* a single point of maintenance */
          ...
          char buffer[input_size_c];
          ...
          char name[input_size_c];
          ...
          ... = new char[input_size_c]; /* allocate memory for an input string •/
          ...
    - Lack of clarify: Why does this naked number mean? What is its role? Why does it have the value it does? Could it change?
      - Bad:
        - horizontal = 1.33 * vertical;
      - Good:
        - horizontal = aspect_ratio_c * vertical;
  - ** Exceptions: When a hard-coded value is acceptable or even preferable to a named constant:*
    - If the value is set "by definition" and it can't possibly be anything else, and it has no meaning or conventional name apart from the expression in which it appears, then giving it a name as a constant can be unnecessary or confusing. Constants that appear in mathematical expressions are an example, but notice that some such constants have conventional names in mathematics, and those names should be used.
      - Example - notice that there is no conventional name for the value of 2 in this formula, but it can't have any other value - giving it a name is pointless. In contrast, the value of pi is fixed but has a conventional name:
        - circumference = 2. * PI * radius; /* 2 can only be 2 */
    - The value is a constant whose value can't be anything else by definition, even if it could be given a name of some sort - such names are unnecessary and distracting.
      - Other examples
        - const int noon_c = 12 /* pointless definition - what else could it be? */
        - double degrees_to_radians(double degrees)
          {

```
    /* 180 degrees in a half-circle by definition */
    return degrees * PI/180.;
}
```

- In some situations, a string literal is a “magic string” in that it might have to be changed for correct program functioning. However, output text strings can help make code understandable if it appears as literal strings in place rather than in a block of named constants. See discussion below under "Output text string constants - in place, or in named constants?”

- ** 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.*
  - Criterion for a useful symbol: Could you meaningfully change the value without changing the name?
  - Bad: `const int kTwo = 2 // what else could it be? 3? This is stupid!`
  - Bad: `const int X = 4    // what is this? Can't tell from this name!`
  - Good: `const int max_input_size_c = 255    // the maximum input size, currently this value`
  - Good: `const double kAspect_ratio = 16./ 9.;    // We can tell what this is!`
- *In C++, declaring and initializing a const variable is the idiom for defining a constant. Don’t use #define - it is an inferior approach in C++ because it does not give type information to the compiler .*
- *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-side 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!";`
- ** Preprocessor symbols defined with #define must be all upper case.*
  - Bad: `#define collection_h`
  - Good: `#define COLLECTION_H`
- *String literal constants can be given names either as a preprocessor symbol with #define, or as a const char * const variable declared and initialized at file scope.*
  - const char * const prompt_msg = "Enter command";
- *Global constants defined as `const` variables at file-scope or externally linked (program-scope) are not global variables - restrictions on global variables 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.
    - File-scope const variables are automatically internally linked by default in C++.
  - If program-scope, follow global variable guidelines:
    - Put only `extern` declarations in a header file.
    - Put definition and initialization in a .c or .cpp file that `#includes` the header file.

- **Global variables**

- *Definition: A global variable is any variable that is declared outside of a function - its scope is either the remainder of the file in which the declaration appears (internal linkage) or program-wide (external linkage). In either case, it is a global variable.*
- ** In this course, global variables can be used only where specifically authorized.*
  - This restriction and the required usage are justified in the following rules for where and why global variables should or shouldn't be used.
- ** 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, or your code is not at all responsible for their values (e.g. cout, stdout)..
  - 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. See above on constants.*
  - Read-only, with a single definition, does not present any maintenance or debugging problem, and helps ensure consistency.

- **String literal constants**
  - *Declare and define string constants as const pointers to const characters initialized to string literals 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.
- **Output text string constants - in place, or in named constants?**
  - *Output statements often contain constant text that labels or documents the output.*
    - cout << "x = " << x << " y = " << y << endl;
  - *Sometimes these might be very lengthy:*

- cout << na << ” autosomes were detected during ” << nah << ” auto-hybridization processes at “ << temp << “ degrees" << endl;
- *In this course, you can do either one of the following with output text constants, with pros and cons:*
  - Code the output text constants in place, in the output statement itself:
    - cout << na << ” autosomes were detected during " << nah << " auto-hybridization processes at " << temp << “ degrees" << endl;
    - Pros: Output statements are more self-explanatory because you can see the output text right there. Overall code structure is simplest.
    - Cons: If messages duplicated, now have a problem because of no single point of maintenance. In a real application, if you need to change the messages (e.g. to translate into another language), you have to find and modify messages all over the code.
  - Define the text in one or more constants that are defined at the top of the source code file:
    - const char * const out1_txt_c = "autosomes were detected during";
      const char * const out2_txt_c = "auto-hybridization processes at";
      const char * const out3_txt_c = "degrees";

      . . .
      cout << na << out1_txt_c << nah << out2_txt_c << temp << out3_txt_c << endl;
    - Pros: No problem with duplicated messages - just use the same constant. Messages are all in one place for easy modification.
    - Cons: Code structure is now a lot more complicated. Unless names of constants are extremely descriptive, you can't tell much about the output from looking at the output statement.
  - It is your choice which approach to use in your code. You can also use a sensible consistent combination of both approaches. For example, if the same message is used more than once, define the text as a constant; otherwise, define the text in place.

- **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: compensating for different C++11 implementations depending on the compiler.
  - *All symbols defined with #define must be ALL_UPPER_CASE.*
    - A critical reminder that a macro is involved.
  - *In C++, use variables defined as const or constexpr instead of #define for program constants.*
    - The compiler knows it has a type instead of just being an ignorant text-editing operation.
    - 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 nullptr (C++11) or plain zero (in C++98)*
    - C++11 added nullptr, a 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++.
- *Use "typename" instead of "class" in template parameter declarations.*
  - A template parameter doesn't have to be a "class" type, so "typename" is more clear.
- *Prefer type alias (with "using") instead of typedef.*
  - A type alias can be a template, unlike a typedef.
- ** Do not used #define to define constants; use const variables or constexpr instead.*
- ** 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 this course.*
  - Too easy to cause serious problems in construction/destruction processing.
- ** Don't use 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 when 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`
- ** Don't used std::string::compare when the regular comparison operators will work.*

- Obfuscated:
  - if(!str.compare("Hello")) // ??? What the heck is going on here ???!!!
- Obvious what you mean:
  - if(str == "Hello")
- The compare function returns {negative, 0, positive}, basically like applying strcmp to the corresponding C-string. But why use it when std::string gives you the obviously meaningful comparison operators? It is there only for the rare occasions when the old strcmp logic might be useful.

- *Use the empty() member function to check a container for being empty rather than comparing the size() to zero.*
  - Poor: We don't really care how many elements there are, so why are we asking?
    - if(container.size() == 0)
  - Good: Simpler, and more expressive of what we want to know:
    - if(container.empty())
- *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. Note that a value of nullptr will test as if it was zero or false.*
  - Clumsy:
    - `if(ptr != 0) or if(ptr == 0)`
  - Better:
    - `if(ptr) or if(!ptr)`
- *Write* for *statements used with arrays or vectors 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, use 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.

- *The concept of the* `for` *statement is that the looping (or iterating) variable is initialized, tested, and incremented in one place. If you want to modify its value in the body of the loop, use a while or do-while instead.*
  - Bad:
    ```
    for(auto it = cont.begin(); it != cont.end();) {
       if (condition(*it)) {
          cont.erase(it++);  // why are we modifying the looping variable?
           }
       else {
          ++it;  // why are we modifying the looping variable?
           }
    }
    ```
  - Better:
    ```
    auto it = cont.begin();
    while(it != cont.end();) {
       if (condition(*it)) {
          cont.erase(it++);
           }
       else {
          ++it;
           }
    }
    ```

- *When iterating through a container or other data structure, if possible arrange the iteration code to handle the empty case automatically instead of as a separate special case.*
  - Bad: Check for empty is completely redundant clutter:
    - ```
      if(container.empty())
           return;

      for(auto iter = container.begin(); iter != container.end(); ++iter)
           { /* do stuff */ }
      ```
  - Good: For statement automatically does nothing if the container is empty!
    - ```
      for(auto iter = container.begin(); iter != container.end(); ++iter)
           { /* do stuff */ }
      ```
- *Catch exceptions by reference, not by value.*
  - Not only is there no point in copying the exception object, but they are often from a class hierarchy; catching by reference prevents them from being "sliced" and enables virtual functions to be called on them, such as the `what()` function defined for `std::exception`.
  - Bad:
    ```
    catch (Error x) { ... }
    catch (std::exception x) { ... }
    ```
  - Good:
    ```
    catch (Error& x) { ... }
    catch (std::exception& 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.

- Occasionally you need to call a member function of “this” object’s through a pointer-to-member-function, which requires explicit use of `this.`
- 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 declare 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 of a member function. Reserve const member functions that modify a mutable member variable for situations such as value caching in which you 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.
    - Likewise, don’t use indirection of data to fake constness of a member function - the fact that this can be done does not mean that it is contributes to the clarity of the design - it obfuscates it.
  - *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.
- *The std::set container requires that its contained items be const so that the ordering is maintained. If there is an advantage to using std::set for objects that must be changed, you must handle the changes correctly.*
  - Do not circumvent set's requirements by faking constness with tricks using mutable or const_cast.
  - Instead, use something that always works: 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 using a set of pointers to non-const objects.
- *Storing pointers to non-const objects in a std::set container makes it possible to change the pointed-to objects using pointers in the set, but avoid designs in which it is possible to change the key field used for ordering the pointers.*
  - E.g. ensure that the key field is a const member variable, or has no setter function.
  - Otherwise, use something that always works: remove the pointer from the set container, change the pointed-to object, and put the pointer back into the set.
  - Do not play tricks with mutable or const_cast.

- **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, use inline functions to get both performance and clarity.
  - *Define functions that correspond to significant 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 (see the next guideline).
      - 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.
  - ** Don't clutter code with tiny trivial functions that add no value to clarity or maintainability.*
    - Putting code in a function should add value to the code in some way, either by avoiding code duplication, hiding implementation details, or expressing a concept about the program behavior. If there are no details to put in one place or to hide, or concept being expressed, the function is distracting and pointless. Why make the reader find the definition just to discover the same code could have been written in place with no problem?
      - Bad because there are no details to hide or provide a single point of maintenance for:

```
if (/* some condition */)
      throw_error();

. . .
void throw_error()
{
      throw Error("An error has occurred!");
}
```

      - No better: there is still no value added:

```
if (/* some condition */)
      throw_error("An error has occurred!");

. . .
void throw_error(const char* msg)
{
      throw Error(msg);
}
```

      - Good - reader can see the same information without looking for the pointless function.

```
if (/* some condition */)
      throw Error("An error has occurred!");
```

      - See related guideline: * *Do not write functions that simply wrap a Standard Library function.*
  - ** 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.
  - IMPORTANT: the function call is always commented with an explanation of the switching parameter value

- *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);
```

- *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 stream 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();`
  - ** Return the value of a logical expression instead of wrapping it in a redundant if or ? statement that returns true or false. Invert or negate the expression if necessary.*
    - Bad:
```
if(a > b)
     return true;
else
     return false;
```
    - So bad it is ridiculous:
```
return (a > b) ? true : false;
```
    - Good:
```
return(a > b);
```

- *Use the ternary ? operator only to compute a value to be stored or used. This is the unique ability of this operator. It is not a general substitute for the if statement.*
  - A one-liner for choosing a value:
    x = (a > b) ? z + w : z - w;
  - A good solution to initializing a reference to two different variables:
    ```
    Thing& t = (a > b) ? thing1 : thing2;
    ```
  - Sometimes handy in output:
    ```
    cout << "Your code is " << (good_flag) ? "very good" : "terribly bad"
    << " please act accordingly" << endl;
    ```
- ** Use "flat" conditional code instead of deeply nested code.*
  - Deeply nested conditional code is hard to read and fragile to modify. The consequences of conditions end up being far away from the conditions. Instead, use 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 or silly uses of "flag" variables to keep track of what to return. 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 and returning or throwing exceptions.
    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.
- *Do not use the "short circuit evaluation" feature of the logical operators && and ll as a way to specify flow of control that could be expressed more explicitly with if-else.*
  - Using short-circuit evaluation for flow of control is an hold-over from early non-optimizing compilers and less expressive languages. Modern compilers will generate optimized code either way, so there is no advantage to making your reader solve a logic problem to discover the sequence of program activity. Note that short-circuit evaluation is not a logic principle, but an old optimization trick.
  - How to tell:
    - If the values being combined are simple booleans (or the C equivalent) or calls to simple functions with no side-effects that return boolean values (often called predicates), then the expression is a purely logical combination, and letting short-circuit evaluation happen is natural and causes no problems.
    - If the values being combined with logical operators are return values from functions that do real work or have side effects (such as I/O), then understanding the flow of control becomes more complex; be more clear by rearranging the code to use explicit if-else structures.
- *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 a function called in a loop will always return the same value, and so will not attempt to replace the calls inside the loop with a single call before the loop.
  - Bad: `my_special_strlen` gets called every time around the loop - and what does it do?

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

  - Better: If you know it always computes the same result, call it only once before starting the loop.

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

- ** 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.
- You can input multiple variables and test only the final result. Since a input stream failure causes subsequent input operations to do nothing, testing only the final result is well-defined and well-behaved.
  - Good:

```
while (infile >> x >> y >> z) {
     /* use x, y, and z */
     }
/* possible check of specific stream state */
```

- *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 */
...
```

  - *When searching a Standard container, be aware of what the container member functions and Standard algorithms do.*
    - Bad: Searches twice for no good reason:
      -

```
Thing* find_Thing(map<string, Thing*> c, const string& name)
{
      if(c.count(name) > 0) // search for Thing* stored under name
```

```
            return c[name];  // search again for Thing* stored under name
        else
            throw Error("Thing not found!");
    }
```

- Bad: Although `std::set<>` can be searched in logarithmic time, and is often used that way, the `std::find` algorithm always takes linear time regardless of the container it is applied to.
  - ```
    bool is_present(set<Gizmo> c, const Gizmo& probe_Gizmo)
    {
        // iterate from begin() to end() looking for match
        auto it = find(c.begin(), c.end(), probe_Gizmo);
        return it != c.end();
    }
    ```

- *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.
  - These functions treat data as "raw memory" and thus are less expressive of the kind of data that is present, and usually harder to get right as a result. For example, using str functions to work on C-strings makes it clear that the data is a C-string and the null byte at the end is automatically handled. If the data is an array, then a loop that explicitly copies the array cells is similarly simpler and more clear. In this course, writing clear and accurate code is more important than capturing every bit of efficiency - which a good compiler and library might well deliver anyway.
- *When to use ordinary functions, `bind`, `mem_fn`, lambda, and custom function objects in STL algorithms or similar situations:*
  - If the relevant function is already defined, then use the following:
    - If it is an ordinary function with no extra parameters, simply use it directly.
    - If it is a member function with no parameters, simply wrap it with `mem_fn`.
    - If it is either an ordinary function with extra parameters, or a member function with extra parameters, use `bind` to specify the additional parameters.
      - * Note: `bind(mem_fn(&class::member_function_name) etc` is redundant nonsense. The `bind` template will do the `mem_fn` for you automatically.
  - If the code is not already in a defined function, then do the following:
    - If the code is short and simple and needed in only one place, define it in-place with a lambda expression in the algorithm call.
      - Use whitespace and indentation to make sure the lambda expression in the algorithm call is easy to read.
      - If the lambda expression is too long or complex to be readable when written in place, then do not use a lambda expression.
      - Storing the lambda expression in a variable defeats the purpose of lambda, and does not improve readability - reserve this approach for the rare special purposes where a lambda expression needs to be used repeatedly.
    - If the code is complex, or it needs to be used in more than one place, either define a function or custom function object class to contain it.
    - If you need to store state during the algorithm execution, use a custom function object class that has member variables, not clumsy and limited ordinary functions with static or global variables.
- *Don't overuse `std::function<>`*
  - Despite its apparent simplicity, it is a heavy-weight mechanism and is poor substitute for simple function pointers, pointers to member functions, a saved lambda, a `std::bind` object, or a simple custom function object class type. Its advantage is that it can store any type that can be called with
    ```

the same return and argument types, ranging from simply a function pointer to a complex function object with virtual functions and different member variables resulting in variable sizes. If you don't need this flexibility, don't pay the complexity and overhead.
    - Consider instead defining a function template in which a templated argument is a function object type created with `std::bind` - this is likely to be simpler and more efficient.
    - Storing the result of `std::bind` in a `std::function<>` variable is likely to be redundant. `std::function<>` can be used to hold a function object, but the result of `std::bind` is already a function object.

- **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.*
        - If the problem is due to a programmer's mistake in logic or coding, an assert() is almost always better than handling in the same way (e.g. throwing an exception) as a user's error in entering a command or data.
        - Run-time errors 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.
    - *Local try-catches around a single function call are probably poor design compared to normal flow of control techniques.*
        - E.g. you add some data to an object, but it might throw an exception if there is something wrong with the data, and you catch this locally rather than let the exception get caught at a higher level.
            - ```
              void add_new_data()
              {
                    try {
                          // prepare some new data
                          the_object->accept(new data);
                    }
                    catch (Error& e) {
                          cout << "data not accepted!" << endl;
                          }
              }
              ```
              ```

- The fact that the try/catch wraps a single function that can throw an exception, and this exception is immediately handled and not rethrown, is a strong hint that an exception is being used in a situation where a normal flow of control would suffice.
- Why does the accept() function throw an error, and why does it need to be caught locally instead of by the higher level?

- Possible better design: Let the higher level handle the error.
  - ```
    void add_new_data()
    {
          // prepare some new data
          the_object->accept(new data);
    }
    ```

- Reserve a local try-catch structure for the situation where you have to do some cleanup in this function before rethrowing the exception to the higher-level handler.
  - ```
    void add_new_data()
    {
          try {
                // prepare some new data
                the_object->accept(new data);
          }
          catch (Error& e) {
                // cleanup and discard new data here before leaving function
                throw;  // rethrow the exception
                }
    }
    ```

- Other better designs if the higher level should not be involved: Ask the object whether everything is OK and handle locally, no exceptions needed.
  - ```
    void add_new_data()
    {
          // prepare some new data
          if(the_object->accept(new data)) { // returns true if a problem
                cout << "data not accepted!" << endl;
                }
    }
    ```
  - ```
    void add_new_data_check_first()
    {
          // prepare some new data
          if(the_object->check(new data)) { // returns true if a problem
                cout << "data not accepted!" << endl;
                return;
                }
          the_object->accept(new data);
    }
    ```
- A combination approach: Ask the object whether everything is OK and throw the error after local cleanup - often the simplest approach:
  - ```
    void add_new_data()
    {
          // prepare some new data
          if(the_object->accept(new data)) { // returns true if a problem
                // cleanup and discard new data here before leaving function
                throw Error("data not accepted!");
                }
    }
    ```
  - ```
    void add_new_data_check_first()
    {
          // prepare some new data
          if(the_object->check(new data)) { // returns true if a problem
    ```

```
            // cleanup and discard new data here before leaving function
            throw Error("data not accepted!");
            }
        the_object->accept(new data);
    }
```

- *Use different exception types to signal different error situations rather than packing different values into the exception objects and testing their contents in the catch.*
  - This is why you can define your own exception type! Helps decouple what the object contains (e.g. a particular error message in some language) from what the type of error is.
- *Some Standard Library container member functions such as .at() throw Standard exception types, but these in practice are rarely useful when feedback needs to be provided to the user - the code generally does not know who or what caused the exception to be thrown.*
- *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.*
    - If possible, avoid allocations that might have to be immediately discarded by checking whether a new object is going to be usable or legal before creating it.
  - *In Standard C++, the new operator will 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.

- **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 both test or switch on the situation using similar code.
  - *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! So there is no need for an object just to hold code that could be in main() or its sub functions.
    - 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.
    - Two *good* reasons for classes that have only one object:
      - 1. 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.
      - 2. 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."
    - One *bad* reason for classes with only one object 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 do *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.
  - ** Scope typedefs, type aliases, or enums used in a class within the 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.
  - ** Keep "helper" class or struct declarations out of the header file; if not possible, put them inside the 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, consider whether the public interface design is defective, and fix the design - these are just “helpers.”

- *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.
  - Friend functions and classes must be part of the same module as the class granting the friendship.
  - 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.*
  - Friend functions and classes must be part of the same module as the class granting the friendship.
  - 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.
- *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.
- *Do not use static member variables to hold constant values for a class, especially if initialized in the class declaration.*
  - No conceptual or maintenance advantage over const file-scope variables declared at the start of the .cpp file.
  - Note that changing a constant value defined in the class declaration in a header file may force a recompile of other modules - not clear what the compensating advantage is.
  - In C++98, only integral types could be initialized this way, leading to inconsistent treatment of constants.
- *Be aware the initializing member variables in the class declaration will force recompilations of client code if the initializing value is changed.*
  - This coupling might be a disadvantage compared to initializing the member variable in the .cpp file.
  - If it is useful for the client code developer to be aware of the initializing values, then setting them in the class declaration could be valuable.
- *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.
  - List the constructor initializers in the same order as the member variables are declared. The compiler will call the initializers in the order of the variable declarations, so listing the initializers in that order will help avoid surprises in which an initializer depends on an uninitialized variable.
- *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.
- *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 the compiler-supplied "special member functions" (the destructor and the copy/move functions) are correct, and let the compiler supply them if so.*
    - Do not write code that the compiler will supply.
    - Unnecessary code is an unnecessary source of bugs.
- *Try to follow the "Rule of five or zero" - either explicitly declare all five of the special member functions, or declare none of them and let the compiler supply them automatically.*
    - "Declare" here means to either declare and define your own version of the functions, or declare what you want the compiler to do with =default or =delete.
    - If you have to write even one of these functions for some reason, explicitly declare the status of the rest of them to avoid confusion or possible undesired behavior.
    - If you have to write your own destructor function to manage a resource (like memory), you almost certainly have to either write your own copy/move functions or tell the compiler not to supply them (with =delete).
    - In writing a copy constructor, remember to copy over all member variables - a common error.
- *If copy or move operations are not meaningful for a class, explicitly prevent the compiler from supplying them.*
    - For example, to enforce the concept that objects in the domain are unique.
    - In C++11, disable compiler-supplied copy and move functions with the =delete syntax.
- *If you declare copy/move functions, use the normal form of declarations for them:*
    - `Classname(const Classname&);  // copy constructor`
    - `Classname(Classname&&);  // move constructor`
    - `Classname& operator= (const Classname&);  // copy assignment operator`
    - `Classname& operator= (Classname&&);  // move assignment operator`
- *Resist the temptation to store an object's state information in static member variables, especially container member variables.*
    - It is unusual for distinct objects to share state, so forcing them to share state data is probably an incorrect design or a premature optimization that will have to be undone at some point.
    - This forces all instances of the class to be in the exact same state at all times - which may be wrong if the objects can be created and destroyed at different times, or their state data are updated at different times.
- *Suspect designs in which a base class has member variables needed for derived class functionality, but has few or trivial member functions that operate on those member variables.*
    - Having only member variables in the base class is pointless because sharing only the *declarations* of member variables with derived classes is useless.
    - This suggests that there is either no real shared functionality, or that shared functionality exists and can be moved up into the base class
- *Suspect designs in which a base class has functionality that is used by only some of the derived classes.*
    - This suggests that shared functionality was moved too far up the inheritance tree - consider instead an intermediate base class for the shared functionality.
- **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.
- ** Put definitions or declarations in a header file into the smallest scope possible.*
  - If possible, at private level of a class, or not even in the header file at all.
  - Do not declare/define anything at the top level of the file unless it is supposed to be available to all clients in all scopes.
  - Do not declare/define anything at the public level of a class unless clients of the class need access to it for good reasons.
- *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 for namespaces 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 namespace `using` statements.**
  - *Namespace declarations and directives.*
    - * No namespace using declarations or directives are allowed at the top level of a header file.
      - OK 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::".
    - *Type aliases*
        - Same rules as a typedef - allowed in a header file only if:
            - Part of the module interface - logically required for the client to use the module.
            - The header file contains declarations are definitions needed in more than one module, as in Utilities.h.
- **Using a project Utilities module**
    - *Place in the Utilities module only functions or declarations that are used by more than one module - a strict requirement in this course.*
        - Examples: A typedef used throughout a project; a function that compares two struct type variables that is needed in two modules.
    - *Do NOT use the Utilities module as a dumping ground for miscellaneous scraps of code - it is reserved for the above use.*
    - *In the real world, or your own code, but not in this course: 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.

- **Code Order and 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, or in C++11, if it is only needed in one function, place the declaration/definition inside the function that uses it.*
    - Do not put at the beginning of the .cpp file - the reader 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 them as obsolete 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.
  - *Comments are for human readers, not the compiler, so place comments where they are most convenient and useful for the human who is looking at the code.*
    - This is almost always at the function definition, rather than the function declarations.

- ** Each public interface function prototype in a header file, and every function definition in a .cpp file, should be preceded by a comment that states the purpose of the function and explains what it does.*
  - No value in comments in a header file for private functions. Code reader looking at the header won't be interested. Comment private helpers in the .cpp file before the function definition.
  - 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 useful 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 to 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! Often, writing comments for complicated code while you are developing it can make the code easier to figure out.