CS100
Introduction to Programming

Lecture 27. Coding Standards

Today's learning objectives

- Basic introduction of coding standards for:
  - Header Files
  - Functions
  - Naming
  - Comments
  - Formatting

Outline

- Introduction
- Header Files
- Functions
- Naming
- Comments
- Formatting

What are Coding Standards?

- Coding standards are guidelines for code style and documentation
- The goal is that any developer familiar with the guidelines can work on any code that follows them
- The produced code will be/remain consistent even if developed by multiple programmers
- Standards range from a simple example statements to involved documents

Why use a Coding Standard?

- Greater consistency between developers
- Easier to develop and maintain
- Saves time and money

Coding Standard Typically Covers

- Header Files
- Functions
- Naming
- Comments
- Formatting

Basic Rules

- Coding standards should not be a burden
  - Standard needs to work for existing team
- Standardize early - the effort to bring your old work into the standard will be too great otherwise
- Encourage a culture where standards are valued and obeyed

Just a convention

- The following is just an example convention
- Follows common sense
- Largely follows Google coding standard

Outline

- Introduction
- Header Files
- Functions
- Naming
- Comments
- Formatting

Header files

- In general, every `.cpp` file should have an associated `.hpp` file
- There are only few exceptions, such as unit tests and “small” `.cpp` files containing just a `main()` function
- Correct use of header files can make a huge difference to the readability, size and performance of your code
  - Separation of interface and implementation
- The following rules aim at avoiding various pitfalls occuring with the use of header files

Header files

- Header files should be self-contained
  - Make sure no substantial part of a (interface) definition ends up in the associated `.cpp` file
- C++ header-files should end with `.hpp`
- Non-header files that are meant for inclusion should end in `.inc` and be used sparingly
  - Example: the implementation of template classes
- Users and refactoring tools should not have to adhere to special conditions to include the header
- A header should have **header guards** (include-guards) and include all other required headers

The include-guard

- Example:

```
#ifndef MYCLASS_HPP_
#define MYCLASS_HPP_

//…

#endif
```

The include-guard

- The format of the symbol name should be standardized
  - Example:

```
<PROJECT>_<PATH>_<FILE>_HPP_
```

- To guarantee uniqueness, they should be based on the full path in a project's source tree

The include-guard

- For example: the file `foo/src/bar/baz.hpp` in project `foo` should have the following guard:

```
#ifndef     FOO_BAR_BAZ_HPP_
#define     FOO_BAR_BAZ_HPP_
...
#endif      // FOO_BAR_BAZ_HPP_
```

Forward declarations

- Avoid using **forward declarations** where possible. Just `#include` the headers you need
- Forward declaration:
  - A declaration of a class, function, or template without an associated definition
  - Pros:
    - Save compile time
    - Save on unnecessary recompilation
  - Cons:
    - Hide a dependency, skip necessary recompilation when headers change
    - May be broken by subsequent changes to the library
    - Difficult to determine whether a forward declaration or a full `#include` is needed
    - Declaring multiple symbols can be more verbose and complex

Forward declarations

- Conclusion:
  - Avoid forward declarations of entities defined in another project
  - When using a function declared in a header file, always `#include` that header
  - When using a class template, prefer to `#include` its header file

Names and order of includes

- All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory shortcuts `.` (the current directory) or `..` (the parent directory).

- For example:
  - `project/src/base/logging.hpp` should be included as:

```
#include "base/logging.hpp"
```

Names and order of includes

- In **`dir/foo.cpp`** or **`dir/foo_test.cpp`**, whose main purpose is to implement or test the stuff in **`dir2/foo2.hpp`**, order your includes as follows:
  - **`dir2/foo2.hpp`**
  - A blank line
  - C system files
  - C++ system files
  - A blank line
  - Other libraries' **`.hpp`** files.
  - Your project's **`.hpp`** files.
- If dir2/foo2.hpp omits any necessary includes, the build of **`dir/foo.cpp`** or **`dir/foo_test.cpp`** will break
- Build breaks show up first for the people working on these files!

Names and order of includes

- All the headers that define the symbols that we rely upon should be included
- Except in the unusual case of **forward declaration**
- However, any includes present in the related header do not need to be included again in the related cpp-file (i.e., `foo.cpp` can rely on `foo.hpp`'s includes).

Names and order of includes

- For example:
  - The includes in `project/src/foo/internal/fooserver.cpp` might look like this:

```
#include "foo/server/fooserver.hpp"

#include <sys/types.h>
#include <unistd.h>
#include <vector>

#include "base/basictypes.hpp"
#include "base/commandlineflags.hpp"
#include "foo/server/bar.hpp"
```

System headers (included with <>)

System headers (included with "")

Outline

- Refresher
- Header Files
- Functions
- Naming
- Comments
- Formatting

Write short functions

- Prefer small and focused functions
- If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program
  - Key-words: Modularisation, reusability, unit testing
- Long functions result in bugs that are hard to find
- Short, concise functions make it easier for other people to read and modify the code!

Reference arguments

- All input parameters passed by lvalue reference should be labeled `const` (const-correctness!)
  - In C, if a function needs to modify a variable, the parameter must use a pointer. e.g. `int foo(int *pval)`
  - In C++, the function can alternatively declare a reference parameter: `int foo(int &val)`
- Pros:
  - Defining a parameter as reference avoids ugly code like `(*pval)++`
  - Makes it clear, a null pointer is not a possible value.
- Cons:
  - References can be confusing

```
void Foo(const string &in, string *out);
```

Function overloading

- Writing a function that takes a `const string&` and overload it with another that takes `const char*`

```
class MyClass {
  public:
     void Analyze(const string &text);
     void Analyze(const char *text, size_t textlen);
};
```

Function overloading

- Pros:
  - Making code more intuitive by allowing an identically-named function to take different arguments
  - Overloading based on const or ref qualification may make utility code more usable, more efficient, or both
- Cons:
  - If a function is overloaded by the argument types alone, a reader may have to understand C++'s complex matching rules in order to tell what's going on
  - Many people are confused by the semantics of inheritance if a derived class overrides only some of the variants of a function

Default arguments

- Allowed on non-virtual functions when the default is guaranteed to always have the same value
- Follow the same restrictions as for function overloading, and prefer overloaded functions if the readability gained with default arguments doesn't outweigh the downsides below

Default arguments

- Pros:
  - Allows an easy way to use default values and also override the defaults without having to define many functions for rare exceptions
  - Cleaner syntax compared to overloading the function
  - Less boilerplate and a clearer distinction between 'required' and 'optional' arguments
- Cons:
  - Another way to achieve the semantics of overloaded functions, so all the reasons not to overload functions apply
  - No guarantee that all overrides of a given function declare the same defaults in a virtual function
  - Function pointers are confusing in the presence of default arguments, since the function signature often doesn't match the call signature. Adding function overloads avoids these problems

Inline functions

- Define functions inline only when they are small, 10 lines or fewer
  - Declare functions that allow the compiler to expand them inline rather than calling them through the usual function call mechanism
- Pros:
  - Generates higher efficiency
- Cons:
  - Overuse of inlining can actually make programs slower
  - Inlining function can cause the code size to increase or decrease depending on the function's size

Inline functions

- Conclusion:
  - Do not inline a function if it is more than 10 lines long
  - Inline functions with loops or switch statements are not cost effective
  - Functions are not always inlined even if they are declared as such
    - Examples virtual and recursive functions

Outline

- Refresher
- Header Files
- Functions
- Naming
- Comments
- Formatting

Why naming?

- Govern naming is almost the most important aspect of consistency
- The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc.
- Naming rules are pretty arbitrary, but consistency is more important than individual preference

General naming rules

- Names should be descriptive
- Do not worry about saving space, make the code understandable to a new reader
- Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project
- Do not abbreviate by deleting letters within a word
- Certain universally-known abbreviations are OK, such as `i` for an iteration variable and `T` for a template parameter

General naming rules

- Example:

```
int price_count_reader; // No abbreviation.
int num_errors; // "num" is a widespread convention.
int num_dns_connections; // Most people know what "DNS" stands for.
int lstm_size; // "LSTM" is a common machine learning abbreviation.
```

- Do not use:

```
int n; // Meaningless.
int nerr; // Ambiguous abbreviation.
int n_comp_conns; // Ambiguous abbreviation.
int wgc_connections; // Only your group knows what this stands for.
int pc_reader; // Lots of things can be abbreviated "pc".
int cstmr_id; // Deletes internal letters.
FooBarRequestInfo fbri; // Not even a word.
```

File names

- Filenames should be all lowercase and can include underscores (_) or dashes (-). Prefer "_".
- C++ files should end in `.cpp` and header files should end in `.hpp`.
- Files that rely on being textually included at specific points should end in `.inc`
- Do not use filenames that already exist in **`/usr/include`**, such as **`db.h`**.
- Make your filenames very specific. Such as using **`http_server_logs.hpp`** rather than **`logs.hpp`**.

File names

- Examples of acceptable file names:
  - `my_useful_class.cpp`
  - `my-useful-class.cpp`
  - `myusefulclass.cpp`
  - `myusefulclass_test.cpp` `// _unittest and _regtest are deprecated.`

Type names

- Type names start with a capital letter and have a capital letter for each new word, with no underscores
  - all types — classes, structs, type aliases, enums, and type template parameters — have the same naming convention
  - should start with a capital letter and have a capital letter for each new word, and no underscores

Type names

- For example:

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// using aliases
using PropertiesMap = hash_map<UrlTableProperties *, string>;

// enums
enum UrlTableErrors { ...
```

Variable names

- The names of variables (including function parameters) and data members are all lowercase, with underscores between words
- Data members of classes (but not structs) additionally have trailing underscores

Variable names

- Common variable names:

```
string table_name; // OK - uses underscore.
string tablename; // OK - all lowercase.
string tableName; // Bad - mixed case.
```

- Class data members:
  - Data members of classes, both static and non-static, are named like ordinary nonmember variables, but with a trailing underscore.

```
class TableInfo {
...
private:
  string table_name_; // OK - underscore at end.
  string tablename_; // OK.
  static Pool<TableInfo>* pool_; // OK.
};
```

Variable names

- Struct data members:
  - Data members of structs are named like ordinary nonmember variables. Do not have the trailing underscores.

```
struct UrlTableProperties {
string name;
int num_entries;
static Pool<UrlTableProperties>* pool;
};
```

Constant names

- Variables declared **`constexpr`** or **`const`**, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case.
- Underscores can be used as separators in the rare cases where capitalization cannot be used for separation
- All such variables with static storage duration should be named this way

```
const int kDaysInAWeek = 7;
const int kAndroid8_0_0 = 24; // Android 8.0.0
```

Variables: Occasionally indicate type

- Common variable names:

```
string s_table_name;
int i_number_tables;
double d_table_size;
```

- Class data members:
  - Data members of classes, both static and non-static, are named like ordinary nonmember variables, but with a trailing underscore.

```
class TableInfo {
...
private:
  string s_table_name_;
  int i_number_tables_;
  double d_table_size_;
};
```

Function names

- Regular (auxiliary) functions have mixed case
- Accessors and mutators may be named like variables (starting with lower case, using underscore as separator, but no trailing underscore)
- Ordinarily, functions should start with a capital letter and have a capital letter for each new word
- Examples:

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

Outline

- Refresher
- Header Files
- Functions
- Naming
- Comments
- Formatting

Why comments?

- Comments are absolutely vital to keeping our code readable
- While comments are very important, the best code is self-documenting
- When writing your comments, write for your audience: the next contributor who will need to understand your code

Comment style

- Use either the `//` or `/* */` syntax, as long as you are consistent
- You can use either the `//` or the `/* */` syntax, though `//` is much more common
- Be consistent with how you comment and what style you use where

File comments

- Start each file with license boilerplate
- Add a description of the contents of a file
- If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, file comments are not required
- All other files should have file comments

File comments

- Legal Notice and Author Line
  - Every file should contain license boilerplate
  - Choose the appropriate boilerplate for the license used by the project (for example, Apache 2.0, BSD, LGPL, GPL).
- File Contents
  - If a `.hpp` declares multiple abstractions, the file-level comment should broadly describe the contents of the file, and how the abstractions are related. A 1 or 2 sentence file-level comment may be sufficient
  - Do not duplicate comments in both the .hpp and the .cpp. Duplicated comments diverge

Class comments

- Every non-obvious class declaration should have an accompanying comment that describes what it is for and how it should be used
- It should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class
- When sufficiently separated (e.g. `.hpp` and `.cpp` files), comments describing the use of the class should be in the definition of the class; comments about the class operation and implementation should be in the implementation of the class's methods

Class comments

- Example:

```
// Iterates over the contents of a GargantuanTable.
// Example:
//    GargantuanTableIterator* iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
//    delete iter;
class GargantuanTableIterator {
  ...
};
```

Function comments

- Declaration comments describe use of the function
- Definition comments describe operation of the function
- Function declarations:
  - almost every function declaration should have comments
  - describe what the function does and how to use it
  - could be omitted only if the function is simple and obvious
  - do not describe how the function performs its task

Function comments

- Types of things to mention in comments at the function declaration:
  - What the inputs and outputs are
  - For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not
  - If the function allocates memory that the caller must free
  - Whether any of the arguments can be a null pointer
  - If there are any performance implications of how a function is used
  - If the function is re-entrant. What are its synchronization assumptions?

Function comments

- For example:

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

Function comments

- Function definitions:
  - add an comment only if there is anything tricky about how a function does its job
  - describe any coding tricks you use
  - give an overview of the steps you go through
  - explain why you chose to implement the function in the way you did
  - should not just repeat the comments given with the function declaration

Variable comments

- In general the actual name of the variable should be descriptive enough. In certain cases, comments are required
- Class data members:
  - any invariants not clearly expressed by the type and name must be commented
  - if the type and name suffice, no comment is needed
  - add comments to describe the existence and meaning of sentinel values, such as nullptr or -1. For example:

```
private:
 // Used to bounds-check table accesses. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

Variable comments

- Global variables:
  - All global variables should have a comment describing what they are, what they are used for, and (if unclear) why it needs to be global
  - For example:

```
// The total number of tests cases that we run through
in this regression test.
const int kNumTestCases = 6;
```

Implementation comments

- You should have comments in tricky, non-obvious, interesting, or important parts of your code.
- Explanatory Comments:
  - Tricky or complicated code blocks should have comments before them. Example:

```
// Divides result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
x = (x << 8) + (*result)[i];
(*result)[i] = x >> 1;
x &= 1;
}
```

Implementation comments

- Line comments:
  - lines that are non-obvious should get a comment at the end of the line
  - Should be separated from the code by 2 spaces
  - If you have several comments on subsequent lines, it can often be more readable to line them up

```
// If we have enough memory, mmap the data portion too
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  // Error already logged
```

Punctuation, spelling and grammar

- Pay attention to punctuation, spelling, and grammar
  - Comments should be as readable as narrative text, with proper capitalization and punctuation
  - complete sentences are more readable than sentence fragments
  - Shorter comments can sometimes be less formal, but still should be consistent with your style

TODO comments

- Use **`TODO`** comments for code that is temporary, a short-term solution, or good-enough but not perfect
  - **`TODO`**s should include the string **`TODO`** in all caps
  - Attach the name, e-mail address, bug ID, or other identifier of the person or issue with the best context about the problem referenced by the **`TODO`**
  - A **`TODO`** is not a commitment that the person referenced will fix the problem. Thus when you create a **`TODO`** with a name, it is almost always your name that is given
  - If your **`TODO`** is of the form "At a future date do something" make sure that you either include a very specific date or a very specific event

TODO comments

- Examples:

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(bug 12345): remove the "Last visitors" feature
```

Outline

- Refresher
- Header Files
- Functions
- Naming
- Comments
- Formatting

Why formatting

- Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style
- Individuals may not agree with every aspect of the formatting rules, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily

Formatting

- Line length:
  - Each line of text in your code should be at most 80 characters long (though variations here exist)
  - A line may exceed 80 characters if it is:
    - a comment line which is not feasible to split without harming readability
    - a raw-string literal with content that exceeds 80 characters
    - an include statement
    - a header guard
    - a using-declaration

Formatting

- Non-ASCII characters:
  - Non-ASCII characters should be rare, and must use UTF-8 formatting
- Spaces vs. tabs:
  - Use only spaces, and indent 2 spaces at a time!
  - Using spaces really is the only way to fix the appearance of your code!

Formatting

- Function declarations and definitions:
  - Return type on the same line as function name, parameters on the same line if they fit

    or

    Return type in the previous line (be consistent!)
  - Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call

Formatting

- Function declarations and definitions:
  - Example:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

Formatting

- Function declarations and definitions:
  - Some points to note:
    - A parameter name may be omitted only if the parameter is not used in the function's definition
    - If you cannot fit the return type and the function name on a single line, break between them
    - If you break after the return type of a function declaration or definition, do not indent
    - The open parenthesis is always on the same line as the function name
    - There is never a space between the function name and the open parenthesis

Formatting

- Some points to note:
  - There is never a space between the parentheses and the parameters
  - The open curly brace is always on the end of the last line of the function declaration, not the start of the next line
  - The close curly brace is either on the last line by itself or on the same line as the open curly brace
  - There should be a space between the close parenthesis and the open curly brace
  - All parameters should be aligned if possible
  - Default indentation is 2 spaces
  - Wrapped parameters have a 4 space indent

Formatting

- Function calls:
  - Either write the call all on a single line

```
bool retval = DoSomething(argument1, argument2, argument3);
```

  - Or wrap the arguments at the parenthesis

```
bool retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

  - Or start the arguments on a new line indented by four spaces and continue at that 4 space indent

```
if (...) {
    DoSomething(
        argument1, argument2,  // 4 space indent
        argument3, argument4);
}
```

Formatting

- Function declarations and definitions:
  - For arguments forming a structure that is important for readability, format the arguments according to structure:

```
// Transform the widget by a 3x3 matrix.
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

Formatting

- Conditionals:
  - Prefer no spaces inside parentheses.
  - The `if` and `else` keywords belong on separate lines
  - Have a space between the `if` and the open parenthesis.
  - Have a space between the close parenthesis and the curly brace

```
if (condition) {  // Good - proper space after IF and before {.
```

  - Short conditional statements may be written on one line if this enhances readability but not allowed when the if statement has an else

```
if (x == kFoo) return new Foo();
```

Formatting

- Loops and **`switch`** statements:
  - **`switch`** statements may use braces for blocks
  - **`case`** blocks in **`switch`** statements can have curly braces or not, depending on your preference

```
switch (var) {
  case 0: {  // 2 space indent
    ...      // 4 space indent
    break;
  }
  ...
  default: {
    assert(false);
  }
}
```

Formatting

- Loops and **`switch`** statements:
  - Braces are optional for single-statement loops
  - Empty loop bodies should use either an empty pair of braces or continue with no braces

```
for (int i = 0; i < kNumber; ++i) {}  // Good - one newline is also OK.
while (condition) continue;  // Good - continue indicates no logic.
```

Formatting

- Pointer and reference expressions:
  - No spaces around period or arrow
  - Pointer operators do not have trailing spaces

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

- Return Values:
  - Do not surround the return expression with parentheses

```
return result;  // No parentheses in the simple case.
// Parentheses OK to make a complex expression more readable.
return (some_long_condition &&
        another_condition);
```

Formatting

- Variable and array initialization:
  - You may choose between `=`, `()`, and `{}` (though the latter one may be very uncommon)

```
int x = 3;
int x(3);
int x{3};
string name = "Some Name";
string name("Some Name");
string name{"Some Name"};
```

Formatting

- Class format:
  - The public section should be first, followed by the protected and finally the private section
  - Keyword should be indented one space
  - Except for the first instance, these keywords should be preceded by a blank line
  - Do not leave a blank line after these keywords

Formatting

- Example:

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
};
```

Formatting

- Constructor initializer lists:
  - Constructor initializer lists can be all on one line

```
// When everything fits on one line:
MyClass::MyClass(int var) : some_var_(var) {
  DoSomething();
}
```

  - Or with subsequent lines indented four spaces.

```
// If the signature and initializer list are not all on one line,
// you must wrap before the colon and indent 4 spaces:
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}
```

Namespaces

- Namespaces for classes
  - A way to create logical grouping

```
namespace MyMathLibrary {

template<class T>
class Vector {
    //…
};

template<class T>
class Matrix {
    //…
};

}
```

Namespaces

- Namespaces for classes
  - A way to create logical grouping
  - Namespace additions can be in different files

File 1

```
namespace MyMathLibrary {

template<class T>
class Vector {
    //…
};

}
```

File 2

```
namespace MyMathLibrary {

template<class T>
class Matrix {
    //…
};

}
```

Nested namespaces

- Namespaces can be nested
  - Creates a hierarchy of the functionality

```
namespace MyMathLibrary {
namespace Linear {
//…
}

namespace Nonlinear {
namespace ClosedForm {
//…
}

namespace Iterative {
//…
}
}
}
```

Formatting

- Namespace formatting:
  - Namespaces do not add an extra level of indentation

```
namespace X {

void foo() {  // Correct.  No extra indentation within namespace.
...
}

}  // namespace
```

- Whitespace:
  - Minimize use of vertical whitespace
  - Don't put more than two blank lines between functions
  - Don’t start or end functions with a blank line

Formatting

- Suggestion:
  - Keep directory structure in agreement with namespaces!
  - Referring to previous example:
    - **MyMathLibrary** is a sub-directory of src
    - **Linear** and **NonLinear** are sub-directories of **MyMathLibrary**
    - **ClosedForm** and **Iterative** are sub-directories of **NonLinear**