Google Style

★ Philosophies of the Style Guide

① Optimize for the reader, not for the writer.
② Rule should pull their weight.
③ Tracking the standard is valuable. Not everything in the standard is equally good.
   ↳ cppreference.com
   ↳ stackoverflow
④ Be Consistent
⑤ If Something unusual is happening, leave explicit evidence for the reader.
⑥ Avoid tricky & hard to maintain constructs.
⑦ Avoide polluting the global namespace.
⑧ Concede to optimization and practicalities when necessary.

★ Google Style

① Header files

⇒ In general, every .cc file should have an associated .h file.

↳ There are some common exceptions, such as unit tests & small .cc files containing just a main() function.

(a) Self-contained Headers

→ Header files should be self-contained (compile on their own) and end in ".h".

→ Prefer placing the definations for template & inline functions in the same file as their declarations.

(b) The #define Guard

→ All header files should have #define guards to prevent multiple inclusion

→ The format of the symbol name should be <Project>_<Path>_<File>_H_

(c) Include what you use

→ If a source or header file refers to a symbol defined elsewhere, the file should directly include a header file which properly intends to provide a declaration or definition of that symbol.

→ Do not rely on transitive inclusions.

(d) Forward Declarations

→ Avoid using forward declerations when possible.

→ Instead include the headers you need.

(e) Inline Functions

→ Define functions inline only when they are small, say 10 lines or fewer.

(f) Names and Order of Inclusion

→ Include headers in the following order:

- → Related header { Example: foo.h is related to foo.cc }
- → C System headers
- → C++ standard library
- → Other library headers
- → Your Project headers

{ All are seperated by a blank line }

→ Within each section the includes should be ordered alphabetically.

(2) Scoping

(a) Namespaces

→ Place code in namespace.

→ Namespace should have unique name based on the project name, & possibly its path.

→ Do not use using-directives

(eg: Using namespace foo)

→ Terminate multi-line namespace with Comments.

→ Do not use Namespace aliases at namespace scope in headerfiles.

(b) Unnamed Namespaces & Static Variables

→ When definitions in a .cc file do not need to be referenced outside that file, Place them in an unnamed namespace or declare them static.

→ Do not use either of these constructs in .h files.

(c) Non member, Static Member, and Global Functions

→ Prefer placing nonmember functions in a namespace.

→ Use Completely global functions rarely.

→ Do not use a Class to simply group static members.

(d) local variables

→ Place a functions variables in the narrowest scope possible, and initialize variables in the declaration.

bad	Good

int i;
i = f();
 	
std::vector<int> v;
v.push_back(1);
v.push_back(2);
	int i = f();
	
	std::vector<int> v = {1, 2};

⇒ Variables need for if, while and for statement should normally be declared within those statement, so that such variables are confined to those scopes.

ⓐ Static and Global Variables

→ Objects with static storage duration are forbidden unless they are trivially destructible.

③ Classes

ⓐ Doing Work In Constructors

→ Avoide virtual method calls in constructors
→ Avoide initialization that can fail if you can't signal an error.

ⓑ Implicit Conversion

→ Do not define implicit conversions.
→ Use the explicit keyword for conversion operators & single-argument constructors.

ⓒ Copyable and Movable Type

→ A class's public API must make clear whether the class is copyable, move-only, or neither copyable nor movable.

|→ Use delete & default keywords.

→ Support copying and/or moving if these operations are clear and meaningful for your type.

ⓓ Structs vs Classes

→ Use a struct only for passive objects that carry data; everything else is a class.
→ For struct, all fields must be public.

(e) Struct vs pairs and Tuples

→ Prefer use of a struct instead of a pair or a tuple whenever the elements can have meaningful names.

(f) Inheritance

→ Composition is often more appropriate than inheritance.

→ When using inheritance make it public.

→ Do not use virtual when declearing an override.

→ Multiple inheritance is permitted, but multiple implementation inheritance is strongly discouraged.

(g) Operator Overloading

↳ Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators.

(h) Access Control

↳ Make class data members private.

(i) Declaration order

Public → Protected → Private

→ Within each section:

↳ Prefer grouping similar kind of declarations together.

↳ Prefer the following order:

① types (typedef, using, nested structs & class)

② Constants

③ factory functions

④ Constructors & assignment operators

⑤ destructors

⑥ all other methods

⑦ Data members.

④ Functions

ⓐ Input and Output

→ The output of a C++ function is naturally provided by a return value & sometime via output Parameter.

→ Prefer using return values over output parameters.

→ Prefer using return by value or, failing that return by reference.

→ Avoide returning a pointer unless it can be null.

→ When ordering function parameters
  ↳ Put all input only parameters before any output parameters.

ⓑ Write Short functions

→ Prefer small and focused functions.

ⓒ Function overloading

→ Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called.

ⓓ Default Arguments

→ Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value.

⑤ Other C++ Features

ⓐ Rvalue References

→ Use rvalue reference only in certain special cases.

ⓑ friends

→ Friend should usually be defined in the same file so that the reader does not have to look in another file to find use of private members of a class.

→ Common use:

- Builder class
- Unittest class

ⓒ Exceptions

→ Do not use exceptions.

ⓓ Casting

→ Use C++ Style Casts

→ Use Brace Initialization for arithmetic type `int{1}`

→ Do not use cast formed like `(int)x`, unless the cast is to void.

(e) Preincrement and Predecrement

→ Use ++i or --i.

(f) Integer Type

→ Use integer types defined in <stdint.h>

(g) 0 and nullptr/NULL

→ Use nullptr for pointers & '\0' for chars.

(h) Sizeof

→ Prefer sizeof(varname) to sizeof(type)

(i) Type Deduction (auto)

→ Use type deduction only if it makes the code clearer to reader.

6 Naming

- File Name → lowercase and can include (_)
- Type Name → Start with a capital letter & have a capital letter for each new word, and no underscores.
- Variable
  - → a_local_Variable
  - → a_Struct_local_data_member
  - → a_Class_data_member.

Constant Names

→ kDaysInAWeek

Function Names

→ AddTableEntry()

Namespace Name

→ All lower-case, with words separated by underscore.

Enumeration Names

→ enum class UrlTableErrors {

kOk = 0,

kOutOfMemory,

kMalformedInput,

};

Macro Name

→ #define PI_ROUNDED 3.0

⑦ Comments

// //!

~~//Todo~~ // TODO(- - - -): - - - - -