Type	**Naming Convention**	**Examples**
Function	Follow lower Camel Case convention which requires capitalizing the first letter of every word in a name or sentence except that of the first word.	function, myFunction

Variable	Follow lower Camel Case convention which requires capitalizing the first letter of every word in a name or sentence except that of the first word.
	Add prefix "m_" for data members.
	Add prefix according to data types as mentioned below.
	▪ 'n' for type int
	▪ 's' for string
	▪ 'c' for char
	▪ 'p' for pointer
	▪ 'f' for float
	▪ 'd' for double
		x, var, myVariable
		int nLength;
		string sStudentName;
		char cAplha;
		int * pnStart;
		float fPi;
		double dLength;

Class	Start each word with a capital letter. Do not separate words with underscores. This style is called Upper camel case. Add prefix "C" to class name.	CModel, CMyClass
Method	Follow Lower Camel Case convention which requires capitalizing the first letter of every word in a name or sentence except that of the first word.	classMethod, method
Constant	Follow Upper Camel Case convention which requires capitaliziing the first letter of every word. Add prefix "k" for constants.	kConstant, kMyConstant, kMyLongConstant
Module	Follow Lower Camel Case convention which requires capitalizing the first letter of every word in a name or sentence except that of the first word.	module.cpp, myModule.cpp
Package	Use a short, lowercase word or words. Do not separate words with underscores.	package, mypackage
Preprocessor Macro Names	Use UPPERCASE and Underscores for Preprocessor Macro Names	MY_MACRO
Structures	Start each word with a capital letter. Do not separate words with underscores. This style is called Upper camel case. Add prefix "S" to structure name.	SMyStructure
Enumerated datatype (enum)	Follow Upper Camel Case convention which requires capitaliziing the first letter of every word. Add prefix "e" for constants.	enum eGender { MALE, FEMALE};

	Note:- enum values are all in capital letters.	
Namespace	Use all lower-case alphabets, with words separated by underscores to improve redability.	sample_namespace

Header Files

In general, every .cc file should have an associated .h file. There are some common exceptions, such as unit tests and small .cc files containing just a main() function.

Self-Contained Headers

- Header files should be self-contained (compile on their own) and end in .h.
- A header should have header guards and include all other headers it needs.
- When a header declares inline functions or templates that clients of the header will instantiate, the inline functions and templates must also have definitions in the header, either directly or in files it includes. Do not move these definitions to separately included header (-inl.h) files; this practice was common in the past, but is no longer allowed. When all instantiations of a template occur in one .cc file, either because they're explicit or because the definition is accessible to only the .cc file, the template definition can be kept in that file.
- Add class declarations and function declarations in header file and their definitions in respective source files which include the respective header files.

The #define guard

- All header files should have #define guards to prevent multiple inclusion. The format of the symbol name should be *<PROJECT>*_*<FILE>*_H.

```
#ifndef DEMO_HELPER_H

#define DEMO_HELPER_H

...

#endif // !DEMO_HELPER_H
```

- 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. It should not include header files for any other reason.

Preprocessor Macro Names

- Use UPPERCASE and Underscores for Preprocessor Macro Names

```
#define DEPRECATED_FEATURE

#define MIN(a,b)  ((a)  < (b) ? (a)  :  (b))
```

- Avoid defining macros, especially in headers; prefer inline functions, enums, and const variables. Name macros with a project-specific prefix.
- The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:
    - Don't define macros in a .h file.
    - #define macros right before you use them, and #undef them right after.
    - Do not just #undef an existing macro before replacing it with your own; instead, pick a name that's likely to be unique.
    - Try not to use macros that expand to unbalanced C++ constructs, or at least document that behavior well.
    - Prefer not using ## to generate function/class/variable names.

Type and Constant Names

- Use "UpperCamelCase" with specific prefixes for Classes, Constants, Structures, Enumerations, and Typedefs Capitalize the first letter of each word that appears in a type name or constant name to provide a visual cue for separating the individual words within a name. The leading capital letter provides a mechanism for differentiating between types and variables or functions.
    - 'C' for Class. Example:- class CLibrary;
    - 'S' for structure. Example:- struct SCar;
    - 'e' for enum Example:- enum eWeek{ MON, TUE, WED, THUR };
    - 'k' for constants. Example:- const int kDaysInAWeek = 7;

Variables:

- Use "lowerCamelCase" for Variable and Function Parameter Names.
- Variable name start with the letter denoting the type of variable, followed by the name of variable in camel case. Example:- If there is a string variable StudentName then it will be named as - String sStudentName;
    - 'n' for type int :- int nLength;
    - 's' for string :- string sStudentName;
    - 'c' for char :- char cAplha;
    - 'p' for pointer:- int * pnStart;
    - 'f' for float :- float fPi;
    - 'e' for enum :- enum eGender { MALE, FEMALE}; **Note:- enum values are all in capital letters.**
    - 'd' for double :- double dLength;
- If variable is a data member of a class then variable name starts with prefix 'm_'.

  Example: - int m_nDoorCount; (let's say m_nDoorCount belong to a class Door)

- Do Not Use Case to Differentiate Names

Functions:

- Use “lowerCamelCase” for Function Name

  Example- int getDoorCount();

Classes:

- Class name starts with the capital letter. Also it starts with the letter ‘C’ denoting it’s a class.

  Example- class CSubtitleItem;

- **Declare the Access Level of All Members :** Do not assume that others will remember the default access level of a class [private] or struct [public]. When possible, group declarations into a single section for each access level.
- **Declare All Member Variables Private:** Prefer private over protected access for class data members. Declare all data members private and use “accessor” functions to give subclasses access to those private data members. This to give subclasses access to those private data members. This is especially true for classes that you provide as superclasses for end users to extend by subclassing. Always treat the implementation details as private information to reduce the impact on dependent classes should the implementation change.

Structures:

- Structure name start with the ‘S’.

  Example: struct SAddressDetails;

Namespace Names:

- Namespace names are all lower-case, with words separated by underscores. Top-level namespace names are based on the project name . Avoid collisions between nested namespaces and well-known top-level namespaces.
- The name of a top-level namespace should usually be the name of the project or team whose code is contained in that namespace. The code in that namespace should usually be in a directory whose basename matches the namespace name
- Avoid nested namespaces that match well-known top-level namespaces. Collisions between namespace names can lead to surprising build breaks because of name lookup rules. In particular, do not create any nested std namespaces.

Spaces vs Tabs:

- Use only spaces, and indent 2 spaces at a time.

- We use spaces for indentation. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key.

Scoping:

- **Declare Enumerations within a Namespace or Class**

  To avoid symbolic name conflicts between enumerators and other global names, nest enum declarations within the most closely related class or common namespace. If this is not possible, prefix each enumerator name with a unique identifier such as the enumeration or module name.

- **Declare Global Functions, Variables, or Constants as Static Members of a Class**

  If a global variable or constant closely relates to one class more than any other, make that variable or constant a member of that class. If you have constants that relate to a specific topic or field of study, use an appropriately named class to group those constants; for example, use a class named Math to hold constants for values such as pi, e, etc.

  Prefer classes over namespaces when scoping static variables and constants. Each class should have its own source file in which you can place the initialization statements for these static class members. Namespaces are not typically assigned independent source files.

- **Declare for-loop Iteration Variables Inside of for Statements**

  Iteration variables should be declared inside a for statement. This limits the scope of the variable to the for statement:

  ```
  for (size_t count = 0; count < length; count++){

  // count is only visible inside this block ... }
  ```