C++

Coding Guidelines

Contents

1. Naming

This section contains indications on how to choose names for all entities over which the programmer has control, e.g. classes, functions, variables, namespaces, files etc.

Following are two naming conventions

Pascal case:

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized.

Example: BlackColor

Camel case:

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.

Example: blackColor

1.1. Variable Naming

1.1. Local variables

- Use camel case for local variables.
- Variable names should be meaningful.
- Use Hungarian notation prefix.
- The character next to prefix should be capital.

Type	Prefix	Example
int	n	nStudentAge
float	f	fAverageMark
double	d	dMaximumLetters
char	c	cStartLetter
bool	b	bFlagValue
BOOL	b	bSuccess
BYTE	by	byInputBuffer

unsigned	u	unMaxLength
short	s	sAge
struct	st	stStududent
enum	e	eDayOfTheWeek
class	Nil	car
char[](for storing null terminated string)	sz	szStudName
Pointer variable	p	int *pnBuffLen; char *pszBuffer;

1.2. Class Member Variables

- Member variable name should be prefixed with the character 'm_'.
- Pointer variables should be prefixed with 'm_p'.
- Use camel case to declare member variables.

Example:

```
class CCardManager
{
private:
    int    m_nCardNumber;
    float *m_pfLookUpTable;
};
```

1.3. Pointer Variable

- Pointers should be prefixed with a 'p'.
- Place the '*' *symbol* close to the pointer variable.

Example:

```
int   *pnStudentAge    = NULL;
float *pfStudentHeight = NULL;
```

1.4. Function Argument

- When defining a function, the order of arguments is: inputs, then outputs.
- For argument naming, use the same rule as that of local variable naming.
- Use "IN" and "OUT" keyword with input and output arguments respectively. Use "IN" and "OUT" as comment if the compiler does not support the keywords.
- Use "const" keyword with non-modifiable arguments.

Example:

```
void Concatinate( IN  const char *pszStr1,
                  IN  const char *pszStr2,
                  OUT       char *pszDest);
```

```
void Concatinate( const char *pszStr1  /*IN*/ ,
                  const char *pszStr2  /*IN*/ ,
                        char *pszDest /*OUT*/);
```

1.5. Global Variables

- Global variables should be prefixed with 'g_'.

Example:

```
int g_nMaximum = 50;
```

1.6. Structure Name

- Use uppercase letters.
- Use '_' (underscore) as word separator.
- Use "tag" as prefix.

Example:

```
typedef struct tagSTUDENT
{
    int RollNumber;

}STUDENT, *LP_STUDENT;

typedef const STUDENT* LPC_STUDENT;
```

1.7. Function Names

Use Pascal case for function names.

Example:

```
int DoFileCompression()
{
    int nRetVal = SUCCESS;

    // TODO: Perform file compression.

    return nRetVal;
}

void HandleError(int nErrno /*IN*/)
{
    // TODO: Handle error
}
```

1.8. Class Names

- Use Pascal case for calss names.
- Use “C” as prefix**.**
- It is better to prefix with Module Name identifier.

Example:

```
class CStudent
{
    // TODO: Add member variables..
};
```

1.9. Namespaces

- Use Pascal case to declare namespace.

Example:

```
namespace CartoonNameSpace
{
};
```

1.10. Constants

- Use block letters while naming.
- Separate words using ‘_’ (underscore).

Example:

```
const int MAX_WIDTH = 300;
```

1.11. Enum Names

- Use uppercase letters.
- Use '_' (underscore) as word separator.

Example:

```
enum PIN_STATE
{
    OFF,
    ON
};
```

1.12. Naming for File

- A file name should be unique, in as large a context as possible.
- An include file or implementation file for a class should have a file name of the form <class name> + extension.
- Use Pascal case for file names.
- Include files in C++ always have the file name extension ".h".
- Implementation files in C++ always have the file name extension ".cpp".
- An include file should not contain more than one class declaration.

2. Formatting.

2.1.Blank Spaces

2.1.1. Where to use blank spaces

Scenario	Example
Both sides of Assignment operators.	• `nTimeInterval = 0`
Both sides of Compound-assignment operators	• `nFirst *= nSecond`
Both sides of Logical operators.	• `nFirst && nSecond` • `nFirst || nSecond`
Both sides of Bitwise operators.	• `nFirst & nSecond` • `nFirst | nSecond` • `nFirst ^ nSecond`
Ternary conditional operator.	• `nFirst = ( nFirst < 0 ) ? 0 : 1`
Both sides of Binary operators.	• `nSum = nFirst + nSecond`

After each comma.	• `int nRow = 0` `int nCol = 0` • `CreateTable(nRow, nCol)`
After open bracket	• `if( nRow < MAX_ROWS )`
Before closing bracket	• `if( nRow < MAX_ROWS )`

2.1.2. Where not to use blank space

Scenario	Example
Unary operator and their argument.	• `const int TEST_FAILED = -1` • `++nCount`
Indirection (“variable pointed by”)	• `*pnCardIndex`
Reference (“address of “)	• `&pnCardIndex`
Member access	• `pCar->GetColour()` • `Car.GetColour()`
Inside the angle braces.	• `static_cast<int>( dValue )`
Before and after array index.	• `int nMyArray[MAX_LENGTH]`
Inside the parentheses.	• `nPerimeter = ( nRectLength + nRectBreadth ) * 2`
Before semicolon.	• `int nRectArea = 0;`
Before comma.	• `int nRow, nCol;`
After the Logical Not	• `!bSuccess`
After the Bitwise Not	• `nFirst = ~nSecond`
Scope resolution operator	• `Line::GetInstance()`
Before delete storage (array)	• `delete[] pnCardIndex;`
Avoid trailing white spaces	
Use space to align the code (Avoid Tab).	

Start each 'case' block after four spaces ***(Avoid Tab)***.

Example:

```
switch( nPinNumber )
{
case PIN 1:
    // Do something;
break;
}
```

2.2.Blank Lines.

One blank line should always be used in the following circumstances:

- After a set of include statements.
- Between function definitions.
- Between the local variable declarations in a block and its first executable statement.
- Before a block or single-line comment.
- Between logical sections inside a function, to improve readability.
- After the declaration of access specifier (such as public, protected and private).
- After "break" statement (in switch) if a 'case' follows.

2.3.Braces {}

All curly braces should be placed in a new line.

Example:

```
void AddNumbers( int nFirstNumber ,
                 int nSecondNumber,
                 int *pnResult /*OUT*/)
{
    // TODO: Perform addition.
}
```

Note:

For array initialization, curly braces can use at the same line.

Example:

```
int nArray[] = { 10, 25, 45 };
```

2.4.Comments.

2.5. Single line comments.

For the variables, add comments to the same line. If multi line comments are needed, then give it above the variable declaration.

Example:

```
int nCount;     // Holds the total count of shapes
```

2.6. Avoid block of commented source-code

There should not be any commented code block.

2.7. Document the null statements.

empty block must be commented properly.

Example:

```
if( nFirstNumber > nSecondNumber )
{
    cout << nFirstNumber;
}
else
{
    // Unimplemented
}
```

2.8.Headers.

2.8.1. File header.

```
/**
 * File <FileName>.cpp - Copyright (C) <Year>
 * <Company Name>. All rights reserved.
 * This file contains implementation of the class <Class name>
 *
 * Author  :<Author Name>
 * Version :<Version>
 * Date    :<Date>
 *
 */
```

2.8.2. Class header.

```
/**
 * Class <Class name>
 * <Description>
 * Author  :<Author Name>
 * Version :<Version>
 * Date    :<Date>
 *
 */
```

2.8.3. Function header.

```
// ----------------------------------
// Method      : <Method name>
// Parameters  :
//   <param 1> - <Type> - Description.
//   <param 2> - <Type> - Description.
// Returns     : <Return Type>
// Description : <Method description>
// ----------------------------------
```

3. Programming Practices

1. Source code should be aligned properly. Code should be readable and maintainable.
2. Group the lines of code with proper comment which does a particular functionality.
3. Number of lines in a function should not exceed 100 lines.
4. Break the lines longer than 80 characters.
5. Keep the ordering of methods in the header file and in the source files identical.
6. Use curly braces with control statements.

Example:

```
if( condition )
    // Statements;   // Not recommended.
else
    // Statements;   // Not recommended.

if( condition )// Right
{
    // Statement;
}
else
{
    // Statement;
}
```

7. Avoid mixing declaration of pointer variable and normal variable in a single line.

   Example:

```
float* pfMyPointer = NULL, fAverage = 0.0; // Not recommended.
float *pfMyPointer = NULL;                 // Right
float  fAverage    = 0.0;                  // Right
```

8. Do not use Magic numbers.

   Instead of magic numbers, constants can be declared (#define preprocessors can also be used).

   Example:

```
const int MAX_AGE = 20;

if( MAX_AGE == nAge )
{
    // Some statements
}
```

9. Initialize variable at the time of declaration.

   Example:

```
int nStart;
nStart = 0;    // Not recommended.

int nStart = 0; // Right
```

10. Provide necessary error checks while coding.

    Example:

    Check the resources handles whether its valid before using them.

```
if( NULL == pstFileHandle )
{
    // TODO: Handle error case.
}
```

11. Use constants in the left side of an equality checking.

    Example:

```
const int MAX_AGE = 20;

if( MAX_AGE == nAge )
{
    // some statements
}
```

12. Logging.

    Include log messages in the application, this will help to debugging.

13. After deleting a pointer it must be assigned to NULL.

    Example:

```
delete pstHead;
pstHead = NULL;
```

14. Allocated resources must be released properly.
15. Use optimization in coding, if it's necessary.
16. Header files should be declared in such a way that its content should be included once, even if the header file included multiple times.(Header file may be included to a module from different locations, this will generate redefinition error).

    Example:

    - #define preprocessor can be used for this
    - #pragma once

```
#ifndef _STUDENT_H_INCLUDED_
#define _STUDENT_H_INCLUDED_

// TODO: Add required statements here.

#endif  // _STUDENT_H_INCLUDED_
```

17. Use union type, if and only if it is really needed.

18. Use 'enum' for related constants rather than 'const'.

Example:

```
enum STATE_e
{
    STARTING_STATE,
    RUNNING_STATE ,
    PAUSED_STATE  ,
};
```

19. Use structure definition outside the class.

Example:

```
typedef struct tagNODE
{
    int             Data;
    struct tagNODE *NextNode;

}NODE, *LPNODE;

typedef const NODE* LPC_NODE;

class LinkedList
{
    LPNODE pHead = NULL;
};
```

20. Use 'while' statement immediately after the closing brace. (in the same line)

Example:

```
do
{
    // Do something.

}while( condition );
```

21. Use stream methods where ever it's applicable.
22. Use 'new' and 'delete' ('malloc ' , 'calloc' and 'free' can be used according to need).

```
int *pnLocation = NULL;

pnLocation      = ( int * )malloc( sizeof( int ) * 100 );

// TODO: Do something

free(( void * ) pnLocation );
pnLocation      = NULL;
```

Instead of above code we can use the following code:

```
int *pnLocation = new int[100];

// TODO: Do something

delete []pnLocation;
pnLocation = NULL;
```

23. There should be a single return statement inside a method.
24. Avoid break statements inside the loop; give proper condition expression so that the loop will finish properly.
25. Resolve all the build errors and warnings. Ensure zero warnings and zero error(s) during compilation.
26. All the declared variables should be initialized.

27. Use the standard library (STL) whenever it has the desired functionality.
28. Try to avoid ‘asm’ as much as possible.
29. Do not use file scope objects (ie, global static objects); use class scope instead.
30. Use the bool type of C++ for Boolean.
31. Pointer arithmetic should be handled carefully.
32. Headers supplied by the implementation (system or standard libraries header files) should go in<> brackets; all other headers should go in "" quotes.
33. The Public, Protected and private sections should be declared in the order of
    1. Public
    2. Protected
    3. Private

34. Include method prototypes inside the header file and implement them in corresponding cpp file.
35. Argument names should be meaningful. It will be good if formal arguments and dummy arguments have the same name.
36. Avoid nested classes.