HYU CODING CONVENTION
[C/C++]

Version: 1.0.0

목차

Introduction

이 문서의 목표는 C++을 학습하는 학생들이 자신의 코드를 일관성 있게 작성할 수 있도록 돕는 것이다. 일관성 있는 코드는 가독성을 높이고, 보다 효율적인 유지 및 보수를 가능하게 한다. 또한 다른 학생들과의 공동 작업 시에도 한 사람이 작성한 것과 같은 코드를 생산할 수 있어 프로젝트 진행 능률을 높일 수 있다.

Naming

가독성은 코딩을 하는데 있어서 매우 중요한 요소이다. 가독성을 높이기 위해서는 comment 의 작성도 중요하지만, 기본적으로 comment 없이도 코드를 이해할 수 있을 정도로 코드 내부의 naming 을 잘 하는 것이 가장 중요하다. 여기서 naming 을 잘 한다는 말은 변수, 함수 등의 각종 명칭들이 스스로의 의미를 잘 나타내고 코드를 새로 읽는 사람의 입장에서 코드만 봐도 이해할 수 있는 정도를 이야기 한다.
기본적으로 Naming 을 잘 하기 위해서는 시스템 전체를 알고 이해하고 있으면 도움이 된다.

Make Names Fit

- 명칭을 정할 때에는 최대한 설명을 묘사하고 줄여 쓰는 것을 피해야 한다. 한 라인의 길이가 길어지는 부분에 대해 걱정할 필요는 없다.
- 다음과 같이 대중적으로 쓰이는 abbreviation 은 사용해도 나쁘지 않다.
    - num - number of
    - max - maximum value
    - cnt - counting value
    - key - key value
- 시간, 무게와 같은 단위가 있는 변수는 단위 이름을 추가하는 것도 좋은 방법이다.

```
// Good Examples!
int price_count_reader;
int num_errors;
int num_dns_connections;
uint32 timeout_msecs;
uint32 my_weight_lbs;

// Bad Examples!
int n;                 // Meaningless.
int nerr;              // Ambiguous abbreviation.
int n_comp_conns;      // Ambiguous abbreviation.
int wgc_connections;  // Only you or your group knows what this
stands for.
```

```
int pc_reader;         // Lots of things can be abbreviated "pc".
int cstmr_id;          // Prefer a full written word "customer".
```

Function Names

- function 은 어떠한 액션을 취하는 것이기 때문에 동사로 시작 하는 것이 좋다.
- 모든 function 은 대문자로 시작하며, 여러 word 로 조합되는 경우 각 word 의 맨 앞 글자를 대문자로 표기한다.

```
// Good Examples!
CheckForErrors();
DumpDataToFile();
GetCustomerID();
SetMaxThread();

// Bad Examples!
ErrorCheck();
DataFileDump();
do_not_use_underscores();
```

Local Variable Names

- 모든 local variable 은 소문
- 자를 쓰며 각 word 사이에 underscore (_)를 이용한다.

```
int HandleError(int error_number) {
    int error;
    Time time_of_error;
    int error_processor;

    ...
}
```

Global Variables

- 모든 global variable 은 맨 처음에 'g_'로 시작하여 명칭을 정한다.
- 소문자를 쓰며 각 word 사이에 underscore (_)를 이용한다.
- 참고로 global variable 은 가능한 사용을 하지 않는 것이 좋다.

```
int g_log;
```

```
int g_table_info;
```

Global Constants

- 모든 global constant 는 전부 대문자로 쓰며 각 word 사이에 underscore (_)를 이용한다.

```
const int DAYS_IN_A_WEEK = 7;
const int HOURS_IN_A_DAY = 24;
const int MINUTES_IN_AN_HOUR = 60;
const int SECONDS_IN_A_MINUTE = 60;
```

Structures

- 모든 structure 는 대문자로 시작하여 명칭을 정한다.
- 여러 word 로 조합되는 경우, 각 word 의 맨 앞 글자를 대문자로 표기한다.
- member variable 은 소문자로 표기하고, 각 word 를 underscore (_)로 구분한다.

```
struct StudentInfo {
    unsigned int student_id;
    char name[16];
};
```

Classes

- 모든 class 는 대문자로 시작하여 명칭을 정한다.
- 여러 word 로 조합되는 경우, 각 word 의 맨 앞 글자를 대문자로 표기한다.
- member variable 은 소문자로 표기하고, 각 word 를 underscore (_)로 구분하며 마지막에 underscore 를 하나 추가한다.
- member function 의 이름은 대문자로 시작하며, 여러 word 로 조합될 경우 각 word 의 맨 앞 글자를 대문자로 표기한다.

```
class StudentInfo : PersonInfo {
Public:
    StudentInfo();
    ~StudentInfo();

    void PrintStudentInfo(const unsigned int id);
    ...
```

```
private:
    void UpdateAverageScore();

    unsigned int student_id_;
    char major_[32];
    float average_score_;
    ...
};
```

Enumerators

- 모든 enumerator 은 대문자로 시작하며, 여러 word 로 조합되는 경우 단어의 맨 앞글자를 대문자로 표기한다.
- 각 elements 는 모두 대문자로 표기하며, 각 word 를 underscore (_)로 구분한다.

```
enum PinStateType {
    PIN_OFF,
    PIN_ON
};
```

#define and Macro Names

- 모든 #define 의 명칭은 전부 대문자로 쓰며 각 word 사이에 underscore(_)를 이용한다.
- #define 은 가급적이면 사용을 하지 않는 것이 좋다. #define 은 global constant 와 inline function 으로 대체할 수 있다.

```
#define PI_ROUNDED 3.14
#define ROUND(x) ...
```

Comments

- Comment 의 사용은 귀찮다고 느껴질 수 있지만 코드의 가독성을 향상 시키기 위해 아낌 없는 comment 작성이 필요하다.
- 앞서 설명한 바와 같이 naming 을 잘 한다면 comment 가 많이 필요하지 않을 수가 있다. 하지만 naming 만으로는 부족하고 추가적인 설명이 필요하다고 느낄 시에는 자세한 comment 작성이 필요하다.

File Comments

- 파일에 대한 설명, 작성자와 파일 생성일을 기술한다.

```
/**
 *  this file is ...
 *
 *  @author Jongbin Kim
 *  @since  2016-08-24
 */
```

Function Comments

- 함수의 역할, 사용처 등에 대해 자세히 기술한다.
- 각 parameter 의 이름과 역할을 기술하고, in / out 정보를 포함한다.
- return value 에 대한 설명을 기술한다.

```
/**
 *  this function is used to ...
 *  continue descripting ... blah ..
 *  @param[in]      foo meaning1...
 *  @param[out]     bar meaning2...
 *  @param[in,out]  baz meaning3...
 *  @return         what to return..?
 */
int FuncName(int foo, int *bar, int *baz) {
    ...
}
```

TODO Comments

- 임시 Comment. 해야 할 작업에 대한 내용을 기술한다.

```
int FunctionToImplement() {
    // TODO: implement function and return calculated value
    return 0;
}
```

Common comments

- 구현 과정에서, 설명이 필요한 중요하다고 생각되는 부분에는 필수적으로 comments 를 추가한다.
- 복잡하다고 생각하는 로직의 구현 부분에도 필수적으로 comments 를 작성하여 reader 의 이해를 돕는다.
- /* */, // 중 원하는 것을 사용해도 무방하다.

Formatting

Common

- tab size 는 4 로 한다.
- 코드의 각 line 의 최대 글자 수는 80 자로 한다.

Brace Placement

- if, for 등의 키워드 뒤에 공백 하나를 넣고 소괄호 (를 연다.
- 소괄호 (뒤와) 앞에 공백을 넣지 않는다.
-) 이후 공백 하나를 넣고 중괄호 { 를 연다.

```
if (condition) {
    var = 1;
}
```

if Formatting

- else if, else 는 이전 block 의 닫는 중괄호 } 와 같은 줄에, 공백을 하나 넣은 후 작성한다.

```
if (condition == 1) {
    someting = 2;
} else if (condition == 2) {
    something = 3;
} else {
    something = 4;
}
```

- conditional statement 가 하나라도 { }를 사용하도록 한다.
- 이는 추후 해당 영역의 코드 추가 과정에서의 실수를 방지한다.

```
if (simple == 1)
    var = 2;
if (simple == 1) var = 2;
```

---

switch Formatting

- case block 의 { } 사용은 선택적이다.
- { }를 사용한다면, case 키워드와 같은 줄에서 중괄호 { 를 연다.
- default case 의 사용은 필수적이다. default case 가 실행되는 경우가 없다면 assert 를 통해 예외 처리 한다.

```
switch (var) {
    case 0: {
        ...
        break;
    }
    case 1:
        ...
        break;
    default:
        assert(false);
}
```

---

for Formatting

- for 문 내부에 single-statement 만을 포함하는 경우에도 { }는 필수적이다.

```
for (int i = 0; i < 100; i++) {
    printf("I take it back\n");
}
```

---

while Formatting

```
while (condition) {
    ...
}
```

No statement in loop

- {} 또는 continue 키워드를 사용한다.

```
for (int i = 0; i < some_number; i++) {}
while (condition) continue;
```

Calling Function

- 함수명과 괄호는 붙여 쓴다.
- 한 줄에 표현이 불가능하면 parameter 를 정렬하여 호출한다.

```
bool result = DoSomething(arg1, arg2, arg3);

bool result = DoSomething(very_very_very_very_long_argument1,
                          argument2, argument3);
```

- indent 가 많이 들어간 상황에서 한 줄에 표현할 수 있는 양이 적다면 다음 방식을 허용한다.
- 한 줄에 가능한 한 많은 parameter 를 적어 line 수를 줄이는 것이 좋다.

```
if (...) {
    if (...) {
        if (...) {
            bool result = DoSomething(
                argument1, argument2,
                argument3, argument4);
        }
    }
}
```

Operator

- operator 좌, 우에 공백을 한 칸 넣는다.

```
a = b + c;
d = (e - f) * g;
if (temp == NULL) {
    ...
```

```
}
(condition == 1) ? Func1() : Func2();
```

- 문장이 길어질 경우 operator 앞에서 한 줄을 내리고 indent 를 추가한 뒤 남은 코드를 넣는다.

```
very_long_variable = LongFunctionName()
    + 123456789;

(long_condition == 123456789)
    ? VeryLongStatement()
    : AnotherLongStatement();

if (long_variable == long_statement
    && other_variable == other_statement) {
    ...
}
```

Pointer Variables

- 포인터 variable 선언시 *는 variable 이름과 붙도록 쓴다.

```
char *name = NULL;
char *name, *address;
```

Function Definition

- 한 줄에 너무 많은 글자가 필요한 경우 parameter 를 정렬하여 배치한다.

```
void FunctionName(int arg_name1, char arg_name2) {
    DoSomething();
    ...
}

void MyLongClassName::ReallyLongFunctionName(int arg_name1,
                                             int arg_name2,
                                             int arg_name3) {
    DoSomething();
    ...
}
```

Class Definition

- class 의 Access specifier(public, private, protected)는 추가적인 indent 없이 작성한다.
- public, protected, private 영역 순으로 작성한다.
- 상속 관계의 경우, 부모 class 명을 자식 class 명과 동일한 line 에 작성한다.

```
class StudentInfo : PersonInfo {
Public:
    StudentInfo();
    ~StudentInfo();

    void PrintStudentInfo(const unsigned int id);
    ...
private:
    void UpdateAverageScore();

    unsigned int student_id_;
    char major_[32];
    float average_score_;
    ...
};
```

The #define Guard

- Header file 이 중복으로 include 되는 것을 #define 을 이용해 방지한다.
- format 은 __<PROJECT>_<PATH>_<FILE>_H__ 로 한다.

```
- for project “MyProject”, file “foo.h” in path “dir1/dir2/
#ifndef __MYPROJECT_DIR1_DIR2_FOO_H__
#define __MYPROJECT_DIR1_DIR2_FOO_H__
...
#endif  // __MYPROJECT_DIR1_DIR2_FOO_H__
```