Coding Convention

강의에 사용할 Convention

강사코딩컨벤션 + Google Coding Convention

강사 코딩 컨벤션

- 한 페이지에 최대한 많은 코드가 보이도록, 꾹꾹 눌러담은 코드
- cpp / header 파일을 나누어 전환하며 구현시 헷갈림

팀에서 사용중인 Convention / MISRA C++ 표준 등
본인에게 맞는 컨벤션을 사용할 것을 권장.

Convention

Coding Convention

- 팀원 모두가 지켜야할 코딩 룰.

링크

- 다양한 코딩 표준 : https://github.com/abougouffa/awesome-coding-standards
- AUTOSAR C++14 : https://github.com/abougouffa/awesome-coding-standards/blob/master/autosar-cpp14-2017.pdf

MISRA C++ 08 spec 중...

char, int, short, long, double 같은 basic numerical type 대신 크기를 명확히 알 수 잇는 타입 사용을 권장.

Rule 3–9–2 (Advisory) ***typedefs*** **that indicate size and signedness should be used in place of the basic numerical types.**

[Implementation 3.9.1(1, 5)]

Rationale

The basic numerical types of *char*, *int*, *short*, *long*, *float*, *double* and *long double* should not be used, but specific-length *typedefs* should be used. This rule helps to clarify the size of the storage, but does not guarantee portability because of the asymmetric behaviour of integral promotion. See

Example

The ISO (POSIX) *typedefs* as shown below are recommended and are used for all basic numerical and character types in this document. For a 32-bit integer machine, these are as follows:

```
typedef          char   char_t;
typedef signed   char   int8_t;
typedef signed   short  int16_t;
typedef signed   int    int32_t;
typedef signed   long   int64_t;
typedef unsigned char   uint8_t;
typedef unsigned short  uint16_t;
typedef unsigned int    uint32_t;
typedef unsigned long   uint64_t;
typedef          float  float32_t;
typedef          double float64_t;
typedef long     double float128_t;
```

typedefs are not considered necessary in the specification of bit-field types.

cstdint 와 같은 너비 정수 유형 사용

cstdint (C++11)

Google, Autosar 공통사양

cppreference 참조

- https://en.cppreference.com/w/cpp/header/cstdint

```
#include <cstdint>

int main(){
    int32_t x = 10;

    if (x == 10) {

    }
}
```

괄호

강의는 괄호를 옆에 붙여서 사용함.
교재 제작 / 강의 진행 시 스크롤을 줄이기 위함

```cpp
int main()
{
    std::cout << "HI";

    int a = 10;

    if (a == 10)
    {
        for (int i = 0; i < 5; i++)
        {

        }
    }
}
```

MISRA C++ 2008,
AUTOSAR C++ 14 style

```cpp
int main() {
    std::cout << "HI";

    int a = 10;

    if (a == 10) {
        for (int i = 0; i < 5; i++) {

        }
    }
}
```

Google C++ Style

증감연산자

i++ / i-- 가 아닌, ++i / --i 사용

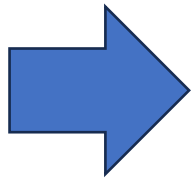
Google / Autosar C++ 공통사양

```
int a = 10;

a++;

for (int i = 0; i < 5; i++) {

}
```



```
int a = 10;

++a;

for (int i = 0; i < 5; ++i) {

}
```

초기화

문자열은 string class 사용
초기화는 Uniform 초기화 { } 를 사용할 것.

```cpp
#include <iostream>
#include <cstdint>
#include <string>

int main() {
    int32_t x = 10;
    std::string str = "ABC";
}
```

AUTOSAR 비권장

```cpp
#include <iostream>
#include <cstdint>
#include <string>

int main() {
    int32_t x = { 10 };
    std::string str = { "ABC" };
}
```

AUTOSAR 비권장

```cpp
#include <iostream>
#include <cstdint>
#include <string>

int main() {
    int32_t x{ 10 };
    std::string str{ "ABC" };
}
```

AUTOSAR C++ 권장

Google Style Guide 요약

Style Guide 링크

Google

- 공식문서 : Google C++ Style Guide
- Example 1 : Tensorflow 코드 중 Header 파일
- Example 2 : Tensorflow 코드 중 Source 파일

헤더파일

1. .cc / .h 파일명
2. 모든 헤더파일은 헤더가드 필수
   - foo/src/bar/baz.h 인 경우 헤더가드 예시
     - #ifndef FOO_BAR_BAZ_H_
     - #define FOO_BAR_BAZ_H_
     - #endif // FOO_BAR_BAZ_H_

선언 순서

1. 본인의 헤더파일
2. 한줄
3. C 헤더
4. 한줄
5. C++ 헤더
6. 한줄
7. 다른 Library들

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

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

#include <string>
#include <vector>

#include "base/basictypes.h"
#include "foo/server/bar.h"
#include "third_party/absl/flags/flag.h"
```

네임 스페이스

- using namespace foo 금지
- 편의성을 위해 using 금지 (가독성은 OK)
  - using ::std::unordered_set; // Bad: just for local convenience
  - using ::std::hash; // Bad: just for local convenience
- 가능한 예제

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

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

지역변수는 항상 초기화

```
int i;
i = f();      // Bad -- initialization separate from declaration.
```

```
int i = f();  // Good -- declaration has initialization.
```


```
int i;
i = f();      // Bad -- initialization separate from declaration.
```

```
int i = f();  // Good -- declaration has initialization.
```

변수 선언은 사용 가까운 곳에 배치

```
int jobs = NumJobs();
// More code...
f(jobs);      // Bad -- declaration separate from use.
```

```
int jobs = NumJobs();
f(jobs);      // Good -- declaration immediately (or closely) followed by use.
```

struct vs class

데이터를 전달할때만 구조체
그 외는 모두 class를 사용한다.

struct vs Pairs / Tuples

struct를 더 선호하여 쓰자.

class 선언 순서

1. Static 상수
2. Factory function
3. 생성자
4. 소멸자
5. 메서드
6. 필드

접두사 증감을 쓰자.

i++ / i– 가 아닌,

++i / --i
(단, i++이 정말 필요할때 제외)

cstdint 와 같은 너비 정수 유형 사용

- <cstdint>
- exact-width integer type

unsigned

음수가 아님을 표현하기 위해 unsigned를 쓰지 말자.

중요할 때만 unsigned 를 쓰자.

C++ 스러운 구현

1. nullptr을 쓰자.
2. 매크로 대신 const
3. C++ 스타일의 타입캐스팅.
4. 스마트 포인터를 사용.
5. 파라미터에 const 사용.

기타

- 적절한 람다 사용은 OK.
- 템플릿 프로그래밍 금지.
- Boost Library 만 사용할 것.

네이밍

변수 n의 의미를 유추할 수 있다면 괜찮다.

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int n = 0;  // Clear meaning given limited scope and context
    for (const auto& foo : foos) {
      ...
      ++n;
    }
    return n;
  }
  void DoSomethingImportant() {
    std::string fqdn = ...;  // Well-known abbreviation for Fully Qualified Domain Name
  }
 private:
  const int kMaxAllowedConnections = ...;  // Clear meaning within context
};
```

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int total_number_of_foo_errors = 0;  // Overly verbose given limited scope and context
    for (int foo_index = 0; foo_index < foos.size(); ++foo_index) {  // Use idiomatic `i`
      ...
      ++total_number_of_foo_errors;
    }
    return total_number_of_foo_errors;
  }
  void DoSomethingImportant() {
    int cstmr_id = ...;  // Deletes internal letters
  }
 private:
  const int kNum = ...;  // Unclear meaning within broad scope
};
```

파일명

FooBar Class는
foo_bar.h / foo_bar.cc 에 존재해야 한다.

- `my_useful_class.cc`
- `my-useful-class.cc`
- `myusefulclass.cc`
- `myusefulclass_test.cc // _unittest and _regtest are deprecated.`

변수이름 / 맴버

맴버변수는 맨 끝 밑줄있음 (구조체 제외)

For example:

```
std::string table_name;  // OK - lowercase with underscore.

std::string tableName;   // Bad - mixed case.
```

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

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

상수와 함수 네이밍

함수는 대문자 카멜케이스
상수는 k로 시작
네임스페이스는 모두 소문자와 밑줄을 사용

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

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

enum class로 쓸 것

매크로 아닌 상수처럼 쓸 것

- 대문자가 아닌 상수 네이밍을 따른다.

```
enum class UrlTableError {
  kOk = 0,
  kOutOfMemory,
  kMalformedInput,
};
```

```
enum class AlternateUrlTableError {
  OK = 0,
  OUT_OF_MEMORY = 1,
  MALFORMED_INPUT = 2,
};
```

공백과 탭

탭키를 누를 때 마다, 공백을 내보내도록 설정을 해야하므로,
공백만 사용하고, 기본적으로 띄어쓰기 2칸을 쓴다.

생성자 관리

사용하지 않을 생성자는 delete 처리를 한다.

```
class Foo {
 public:
  Foo(const Foo&) = delete;
  Foo& operator=(const Foo&) = delete;
};
```

가시성

public > protected > private

- 띄어쓰기는 한칸이다.

```
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_;
};
```

Namespace Formatting

Namespace에 인덴트를 두지 않는다.

```
namespace {

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

}  // namespace
```

```
namespace {

  // Wrong!  Indented when it should not be.
  void foo() {
    ...
  }

}  // namespace
```

함수 선언

아규먼트가 많다면, 다음과 같이 배치할 수 있다.

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

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

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

함수 매개변수 Formatting

파라미터가 많다면, 다음과 같이 배치가 가능하다.

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

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