

Week 2-1 : Good Programming Style

● C++ Programming Style

■ Name Convention

- ◆ Give as descriptive a name as possible, within reason. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

```
int price_count_reader;    // No abbreviation.
int num_errors;           // "num" is a widespread convention.
int num_dns_connections;  // Most people know what "DNS" stands for.
```

```
int n;                    // Meaningless.
int nerr;                 // Ambiguous abbreviation.
int n_comp_conns;        // Ambiguous abbreviation.
int wgc_connections;     // Only your group knows what this stands for.
int pc_reader;           // Lots of things can be abbreviated "pc".
int cstmr_id;            // Deletes internal letters.
```

- ◆ File Names
- ◆ Filenames should be all lowercase and can include underscores (_) or dashes (-). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "_".
- ◆ Type Names
- ◆ Type names start with a capital letter and have a capital letter for each new word, with no underscores: MyExcitingClass, MyExcitingEnum.
- ◆ The names of all types — classes, structs, typedefs, and enums — have the same naming convention.

```
// classes and structs
class UrTable { ...
```

```
class UrlTableTester { ...
struct UrlTableProperties { ...

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

// enums
enum UrlTableErrors { ...
```

- ◆ Variable Names
- ◆ The names of variables and data members are all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores. For instance: `a_local_variable`, `a_struct_data_member`, `a_class_data_member_`.

```
string table_name; // OK - uses underscore.
string tablename; // OK - all lowercase.
```

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

- ◆ Class Data Members

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

- ◆ Function Names
- ◆ Regular functions have mixed case; accessors and mutators match the name of the variable: `MyExcitingFunction()`, `MyExcitingMethod()`, `my_exciting_member_variable()`, `set_my_exciting_member_variable()`.

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

◆ Accessor and Mutators

```
class MyClass {
public:
    ...
    int num_entries() const { return num_entries_; }
    void set_num_entries(int num_entries) { num_entries_ = num_entries; }

private:
    int num_entries_;
};
```

- ◆ Namespace Names
- ◆ Namespace names are all lower-case, and based on project names and possibly their directory structure: `google_awesome_project`.
- Comment Convention
 - ◆ Use either the `//` or `/* */` syntax, as long as you are consistent.
 - ◆ You can use either the `//` or the `/* */` syntax; however, `//` is *much* more common. Be consistent with how you comment and what style you use where.
- ◆ File Comments
- ◆ Legal Notice and Author Line, File Contents
- ◆ Every file should have a comment at the top describing its contents.
- ◆ Class Comments
- ◆ Every class definition should have an accompanying comment that describes what it is for and how it should be used.
- ◆ If you have already described a class in detail in the comments at the top of your file feel free to simply state "See comment at top of file for a complete description", but be sure to have some sort of comment.
- ◆ Function Comments
- ◆ What the inputs and outputs are.
- ◆ For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.

- ◆ If the function allocates memory that the caller must free.
 - ◆ Whether any of the arguments can be a null pointer.
 - ◆ If there are any performance implications of how a function is used.
 - ◆ If the function is re-entrant. What are its synchronization assumptions?
-
- ◆ Variable Comments
 - ◆ In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for.
 - ◆ Line Comments

```
DoSomething();           // Comment here so the comments line up.
DoSomethingElseThatIsLonger(); // Two spaces between the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse(); // Two spaces before line comments normally.
}
vector<string> list{ // Comments in braced lists describe the next element ..
    "First item",
    // .. and should be aligned appropriately.
    "Second item"};
```

- Formatting
 - ◆ Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.
 - ◆ Line Length
 - ◆ 80 characters is the maximum.
 - ◆ If a comment line contains an example command or a literal URL longer than 80 characters, that line may be longer than 80 characters for ease of cut and paste.
 - ◆ A raw-string literal may have content that exceeds 80 characters. Except for test code, such literals should appear near top of a file.
 - ◆ An #include statement with a long path may exceed 80 columns.

- ◆ You needn't be concerned about header guard that exceed the maximum length.

- ◆ 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.

- ◆ Function Declarations and Definitions
- ◆ Functions look like this:

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

If you have too much text to fit on one line:

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

or if you cannot fit even the first parameter:

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

```
}
```

- ◆ If you cannot fit the return type and the function name on a single line, break between them.
- ◆ If you break after the return type of a function declaration or definition, do not indent.
- ◆ The open parenthesis is always on the same line as the function name.
- ◆ There is never a space between the function name and the open parenthesis.
- ◆ There is never a space between the parentheses and the parameters.
- ◆ The open curly brace is always at the end of the same line as the last parameter.
- ◆ The close curly brace is either on the last line by itself or (if other style rules permit) on the same line as the open curly brace.
- ◆ There should be a space between the close parenthesis and the open curly brace.
- ◆ All parameters should be named, with identical names in the declaration and implementation.
- ◆ All parameters should be aligned if possible.
- ◆ Default indentation is 2 spaces.
- ◆ Wrapped parameters have a 4 space indent.
- ◆ Function Calls
- ◆ Function calls have the following format:

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

- ◆ If the arguments do not all fit on one line, they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. Do not add spaces after the open paren or before the close paren:

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

- ◆ Arguments may optionally all be placed on subsequent lines with a four space indent:

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

◆ Conditionals

- The most common form is without spaces. Either is fine, but *be consistent*. If you are modifying a file, use the format that is already present. If you are writing new code, use the format that the other files in that directory or project use. If in doubt and you have no personal preference, do not add the spaces.

```
• if (condition) { // no spaces inside parentheses  
•     ... // 2 space indent.  
} else if (...) { // The else goes on the same line as the closing brace.  
    ...  
} else {  
    ...  
}
```

- If you prefer you may add spaces inside the parentheses:

```
if ( condition ) { // spaces inside parentheses - rare  
    ... // 2 space indent.  
} else { // The else goes on the same line as the closing brace.  
    ...  
}
```

- Note that in all cases you must have a space between the if and the open parenthesis. You must also have a space between the close parenthesis and the curly brace, if you're using one.

```
if(condition) { // Bad - space missing after IF.
```

```
if (condition){ // Bad - space missing before {.  
if(condition){ // Doubly bad.
```

```
if (condition) { // Good - proper space after IF and before {.
```

- Short conditional statements may be written on one line if this enhances readability. You may use this only when the line is brief and the statement does not use the else clause.

```
if (x == kFoo) return new Foo();  
if (x == kBar) return new Bar();
```

- This is not allowed when the if statement has an else:

```
// Not allowed - IF statement on one line when there is an ELSE clause  
if (x) DoThis();  
else DoThat();
```

- In general, curly braces are not required for single-line statements, but they are allowed if you like them; conditional or loop statements with complex conditions or statements may be more readable with curly braces. Some projects require that an if must always always have an accompanying brace.

```
if (condition)  
    DoSomething(); // 2 space indent.  
  
if (condition) {  
    DoSomething(); // 2 space indent.  
}
```

- However, if one part of an if-else statement uses curly braces, the other part must too:

```
// Not allowed - curly on IF but not ELSE  
if (condition) {
```



```
    foo;
} else
    bar;

// Not allowed - curly on ELSE but not IF
if (condition)
    foo;
else {
    bar;
}
```

```
// Curly braces around both IF and ELSE required because
// one of the clauses used braces.
if (condition) {
    foo;
} else {
    bar;
}
```

◆ Loops and Switch Statements

```
switch (var) {
    case 0: { // 2 space indent
        ... // 4 space indent
        break;
    }
    case 1: {
        ...
        break;
    }
    default: {
        assert(false);
    }
}
```

- Braces are optional for single-statement loops.

```
for (int i = 0; i < kSomeNumber; ++i)
    printf("I love you\n");

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

- Empty loop bodies should use `}` or `continue`, but not a single semicolon.

```
while (condition) {
    // Repeat test until it returns false.
}

for (int i = 0; i < kSomeNumber; ++i) {} // Good - empty body.
while (condition) continue; // Good - continue indicates no logic.

while (condition); // Bad - looks like part of do/while loop.
```

◆ Pointer and Reference Expressions

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

Note that:

- There are no spaces around the period or arrow when accessing a member.
- Pointer operators have no space after the `*` or `&`.

When declaring a pointer variable or argument, you may place the asterisk adjacent to either the type or to the variable name:

```
// These are fine, space preceding.
char *c;
const string &str;
```

```
// These are fine, space following.  
char* c; // but remember to do "char* c, *d, *e, ...;"  
const string& str;
```

```
char * c; // Bad - spaces on both sides of *  
const string & str; // Bad - spaces on both sides of &
```

◆ Boolean Expressions

```
if (this_one_thing > this_other_thing &&  
    a_third_thing == a_fourth_thing &&  
    yet_another && last_one) {  
    ...  
}
```

◆ Return Values

- ◆ Do not needlessly surround the return expression with parentheses.
- ◆ Use parentheses in return expr; only where you would use them in x = expr;.

◆ Variable and Array Initialization

- ◆ Your choice of =, (), or {}.
- ◆ You may choose between =, (), and {}; the following are all correct:

```
int x = 3;  
int x(3);  
int x{3};  
string name = "Some Name";  
string name("Some Name");  
string name{"Some Name"};
```

◆ Class Format

- ◆ Sections in public, protected and private order, each indented one space.

- ◆ The basic format for a class declaration (lacking the comments, see Class Comments for a discussion of what comments are needed) is:

```
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_;
};
```

Things to note:

- ◆ Any base class name should be on the same line as the subclass name, subject to the 80-column limit.
- ◆ The `public:`, `protected:`, and `private:` keywords should be indented one space.
- ◆ Except for the first instance, these keywords should be preceded by a blank line. This rule is optional in small classes.
- ◆ Do not leave a blank line after these keywords.
- ◆ The `public` section should be first, followed by the `protected` and finally the `private` section.

- ◆ See Declaration Order for rules on ordering declarations within each of these sections.

- ◆ Namespace Formatting
- ◆ The contents of namespaces are not indented.
- ◆ [Namespaces](#) do not add an extra level of indentation. For example, use:
- ◆ Do not indent within a namespace

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

Reference : C++ Good Programming Style - Google

- Basic of C++

- scan and print
cout<<1<<'a'<<"String"<<endl;
cin>>val1>>val2;

- ◆ Function Overloading

```
#include <iostream>  
  
void function(void)  
{  
    std::cout<<"function(void) call"<<std::endl;  
}
```

```
void function(int a, int b)
{
    std::cout<<"function("<<a<<","<<b<<") call"<<std::endl;
}

int main(void)
{
    function();
    function(12,13);
    return 0;
}
```

- Same name functions : Error occur on C
- Same name functions, and different parameter's type and number : function overloading on C++

◆ Default Parameter

```
#include <iostream>

using std::cout;
using std::endl;

int func(int a=0)
{
    return a+1;
}

int main(void)
{
    cout<<func(11)<<endl;
    cout<<func()<<endl;

    return 0;
}
```

- func() parameter : int type data & default parameter = 0

- ◆ in-line function
 - function inline : Swap function call statement to function's body.

```
#include <iostream>

inline int SQUARE(int x)
{
    return x*x;
}

int main(void)
{
    std::cout<<SQUARE(5)<<std::endl;
    return 0;
}
```

[Exercise]

함수 오버로딩의 이해 (Understanding a function overloading)

1. 2개의 정수와 2개의 문자를 입력 받아 swap 결과를 출력하시오.

Get two integers and two characters and swap them. And print out swapped result.

```
#include <iostream>
using std::cout;
using std::endl;
using std::cin;

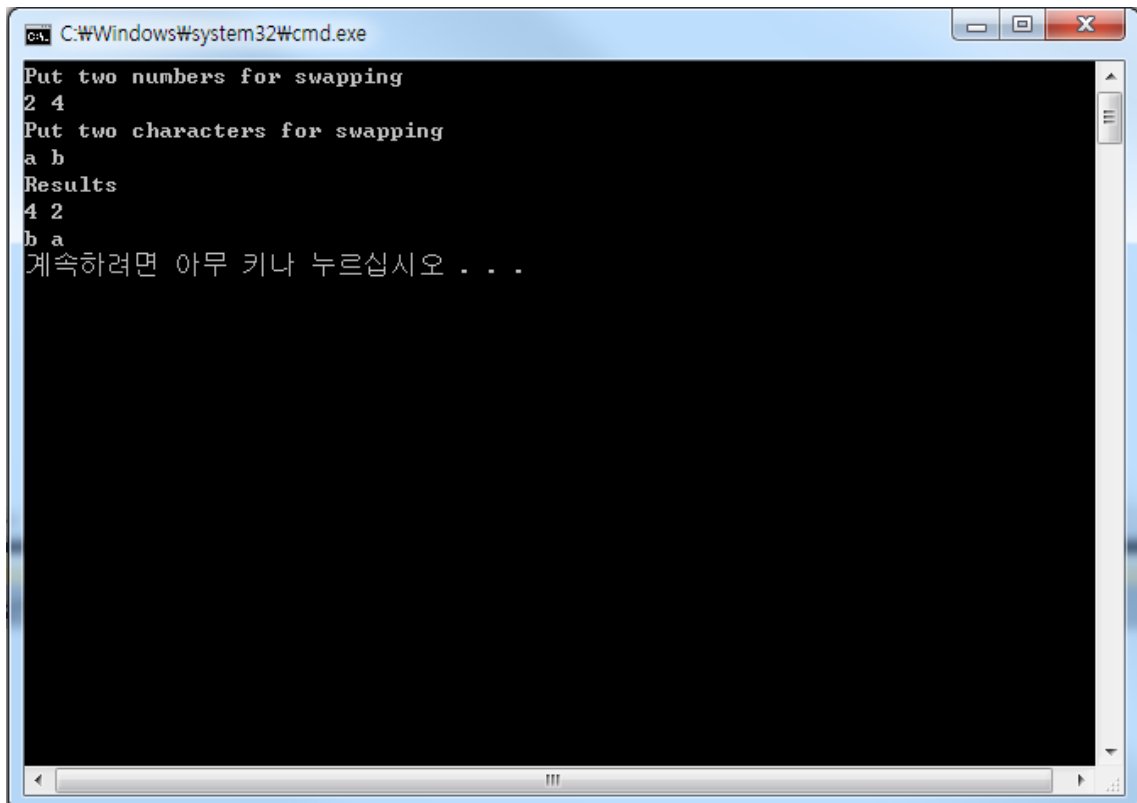
void swap(int *a, int *b);
void swap(char *a, char *b);

int main(void)
{
    // Blank
}

void swap(int *a, int *b)
{
    // Blank
}
```

```
    }  
  
    void swap(char *a, char *b)  
    {  
        // Blank  
    }
```

● 결과 예



2. character를 입력 받아 대문자는 소문자로, 소문자는 대문자로 변환하여 출력하시오.
Get a character. Swap an upper case to a lower case, Swap a lower case to an upper case. And print out swapped result:.

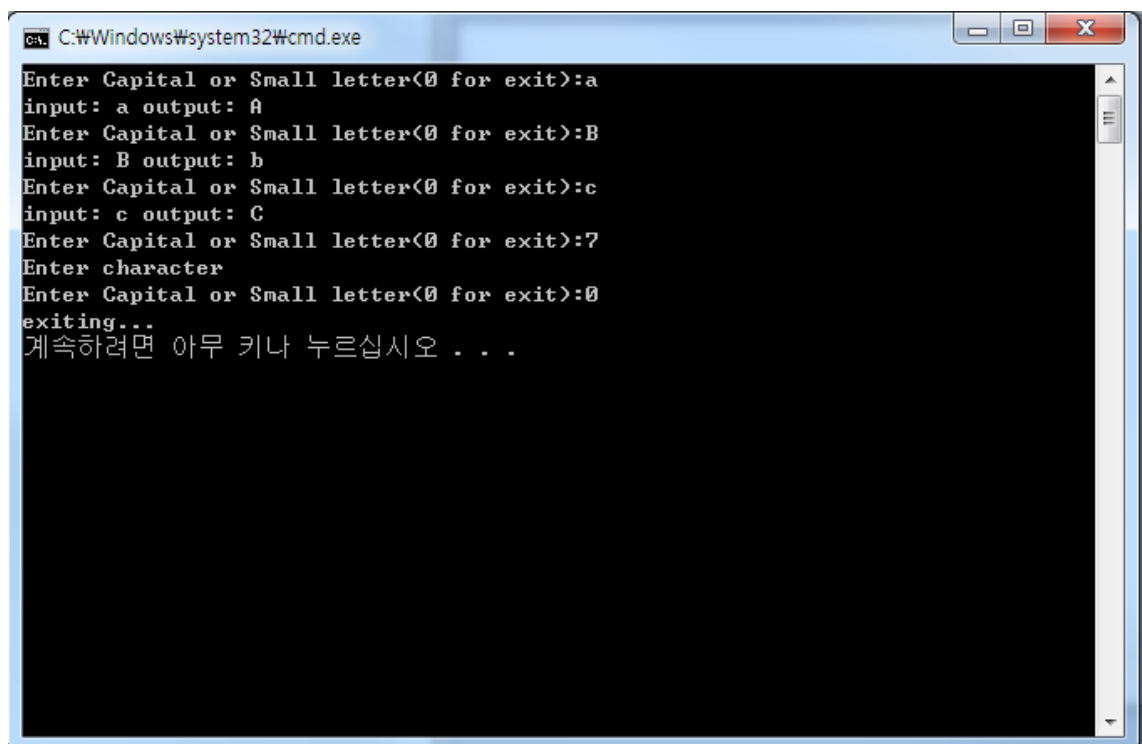
ASCII code : A = 65 / a= 97

```
#include <iostream>  
using namespace std;  
  
int main(void)  
{
```



```
char alphabet;  
bool out=false; // out==true 일 때 종료  
  
//Blank  
  
return 0;  
}
```

- 결과 예



```
C:\Windows\system32\cmd.exe  
Enter Capital or Small letter(0 for exit):a  
input: a output: A  
Enter Capital or Small letter(0 for exit):B  
input: B output: b  
Enter Capital or Small letter(0 for exit):c  
input: c output: C  
Enter Capital or Small letter(0 for exit):?  
Enter character  
Enter Capital or Small letter(0 for exit):0  
exiting...  
계속하려면 아무 키나 누르십시오 . . .
```