Coding Conventions for C++ Developer

1 개요

Code Convention은 프로그래머들에게 매우 중요하다 그 이유를 다음과 같이 들 수 있다.

- 소프트웨어의 80%는 유지/보수에 해당된다.
- 소프트웨어는 처음 개발자에 의해 유지되기가 힘들다.
- Code Convention은 소프트웨어의 Readability를 높여주며, Engineer들이 새로운 코드에 쉽게 적응할 수 있도록 만든다.
- 소스코드를 제품으로 판매할 경우 개발된 다른 제품들과 잘 어울려야 한다.

2 파일명

2.1 File name suffix

C++ 프로그램 파일은 다음과 같은 Suffix를 사용한다.

파일형태	**Suffix**
C++ source	.cpp
C++ header file	.h

3 Naming

3.1 General Naming Convention

3.1.1 타입을 표현하는 이름은 대소문자를 혼용할 수 있지만, 반드시 명사를 사용하고 시작 글자를 대문자로 지정한다.

```
Line, SavingsAccount
```

3.1.2 변수의 이름은 대소문자를 혼용할 수 있지만 반드시 소문자로 시작한다.

```
line, savingsAccount
```

이렇게 변수와 타입을 손쉽게 구별할 수 있도록 하면, 이름의 충돌이 발생하는 것을 효과적으로 해결할 수 있다.

```
SavingsAccount savingsAccount; // 선언문
```

3.1.3 상수(enumeration 변수포함)를 표현하는 이름은 반드시 모두 대문자로 지정

하되 단어들을 구분하기 위해서는 '_' 를 사용한다.

```
MAX_ITERATIONS, COLOR_RED, PI
```

일반적으로, 이러한 상수의 사용은 최소화해야만 한다. 대다수의 경우 상수 변수를 메소드로 구현하는 것이 더 낫다.

```
int getMaxIterations()      // NOT: MAX_ITERATIONS = 25
{
  return 25;
}
```

이러한 양식이 보다 가독률을 높여주며, 클래스의 값을 참조하는 일관된 인터페이스를 제공할 수 있다는 장점을 가진다.

3.1.4 메소드나 함수를 표현하는 이름은 대소문자를 혼용할 수 있지만 반드시 동사를 사용하며 소문자로 시작한다.

```
getName(), computeTotalWidth()
```

3.1.5 네임스페이스를 표현하는 이름은 소문자를 사용한다.

```
analyzer, iomanager, mainwindow
```

3.1.6 템플릿 타입을 표현하는 이름은 대문자 한글자를 사용한다.

```
template<class T> ...
template<class C, class D> ...
```

이러한 점은 다른 명명하는 변수들에 비해서 상대적으로 두드러져보일수도 있다.

3.1.7 축약형(Abbreviations)과 두문자어형(acronym)을 이름에 사용할 경우에는 전부 대문자로 지정하지 않는다[4].

```
exportHtmlSource();     // NOT: exportHTMLSource();
openDvdPlayer();        // NOT: openDVDPlayer();
```

축약형 혹은 두문자어형 이름을 모두 대문자로 지정하게 되면 앞서 기술한 명명

지침들과 충돌이 발생한다 (대표적으로 상수에 대한 명명 지침과 혼동될 수 있다). 그렇다고 이 유형의 이름을 dVD, hTML 등 과 같이 일정하지 않은 다양한 형태로 지정할 경우에는 가독성이 더 떨어진다. 다른 문제점으로는 위의 예제에서 살펴볼 수 있는 바와 같이, 이 유형의 이름이 다른 이름과 결합되는 경우 가독성이 극도로 나빠진다는 것이다. 후속하는 단어가 있을 경우는 더욱 각별히 주의해서 명명할 수 있어야 한다.

3.1.8 글로벌 변수들은 :: 연산자를 이용하여 사용한다.

```
::mainWindow.open(), ::applicationContext.getName()
```

3.1.9 private 접근 제한자를 갖는 클래스 변수에 '_' 접미사를 사용한다.

```
class SomeClass {
  private:
    int   length_;
}
```

변수의 이름이나 타입과는 별개로, 변수의 범위(scope)는 매우 중요한 특성이다. 접미사 '_' 를 사용하게 되면 클래스 범위의 변수(로컬 변수가 아닌)임을 쉽게 구별할 수 있게 된다. 클래스 변수는 메소드 내에서 선언되는 로컬 변수에 비해 중요도가 높기 때문에 프로그래머가 각별히 주의해야 하여 선언한다.
'_' 를 사용하는 명명 규칙은 부가적으로 setter 메소드에서의 이름 충돌문제를 깔끔하게 해결해 준다:

```
void setDepth (int depth)
{
  depth_ = depth;
}
```

제기될 수 있는 이슈거리는 '_' 를 변수의 접두사로 쓸 것인가 접미사로 모두 사용할 수 있으나 이름을 읽기 편하다는 측면에서 접미사로 활용할 것을 권고한다.

3.1.10 반적인 변수의 이름은 타입의 이름과 동일하게 지정한다.

```
void setTopic (Topic *topic)        // NOT: void setTopic (Topic *value)
                                    // NOT: void setTopic (Topic *aTopic)
                                    // NOT: void setTopic (Topic *x)

void connect (Database *database) // NOT: void connect (Database *db)
                                  // NOT: void connect (Database *oracleDB)
```

용어나 이름의 수를 줄이는 것이 코드의 복잡도를 줄인다. 또한 변수의 이름만으로도 그 타입을 손쉽게 유추할 수 있게 해준다는 장점도 있다.
일반적이지 않은 변수들은 각기 나름의 역할(role)을 가지고 있기 때문에 이러한 변수들은 다음과 같이 역할과 변수의 타입을 결합하여 이름을 짓는다.

```
Point startingPoint, centerPoint;
Name   loginName;
```

3.1.11 모든 이름은 영어로 작성한다.

```
fileName;     // NOT:    filNavn
```

국제적인 개발에 있어서 영어가 선호되기 때문에 영어로 작성한다.

3.1.12 넓은 범위에 영향을 미치는 변수는 긴 이름을 부여하고, 좁은 범위의 변수는 짧은 이름을 부여한다 [1].

```
임시 저장공간이나 인덱스로 사용되는 Scratch variable (주: 의미있는
값을 갖지 않고 그때그때 상황에 따라 값들을 잠시 보관해 두기 위한
변수로, 대개 보유한 값이 얼마 후에 의미가 없어지거나 삭제됨)들은
매우 짧은 이름을 부여해서 프로그래머가 그러한 변수들을 읽음과
동시에, 이 변수는 몇 라인 뒤에 그 값이 유효하지 않을 것임을 짐장할
수 있게 해야 한다.

보편적인 scratch variable 로는 정수를 저장하는 i, j, k, m, n 가 있고
문자를 저장하는 c, d 가 있다.
```

3.1.13 호출하려는 객체의 이름을 통해 의미를 짐작할 수 있다면, 메소드의 이름을 간략화할 수 있다.

```
line.getLength();     // NOT:   line.getLineLength();
```

3.2 Specific Naming convention

3.2.1 get/set 이라는 용어는 반드시 애트리뷰트에 직접 접근하는 메소드에 사용한다.

```
employee.getName();         matrix.getElement (2, 4);
employee.setName (name);    matrix.setElement (2, 4, value);
```

3.2.2 compute 라는 용어는 무엇인가를 계산(시간이 소요되는)하는 메소드에 사용

할 수 있다.

```
valueSet->computeAverage();  matrix->computeInverse()
```

프로그래머로 하여금 보는 즉시 이 메소드는 잠재적으로 시간이 소요되는 작업을 수행하고 있다는 것을 주지심으로써, 이 메소드의 결과를 반복해서 사용하고자 할 경우에 프로그래머는 계산된 값을 캐시해 둘 것을 고려하게 만들 수 있다. 용어를 일관성있게 사용하는 것은 가독성을 높이는 지름길이다.

3.2.3 find 라는 용어는 무엇인가를 찾는 메소드에 사용할 수 있다.

```
vertex.findNearestVertex();   matrix.findMinElement();
```

3.2.4 initialize 라는 용어는 객체나 개념이 확립되는 곳에 사용할 수 있다.

```
printer.initializeFontSet();
```

3.2.5 GUI컴포넌트를 나타내는 변수들은 각 컴포넌트의 타입을 접두사로 사용한다.

```
mainWindow, propertiesDialog, widthScale, loginText, leftScrollbar,
mainForm, fileMenu, minLabel, exitButton, yesToggle etc.
```

변수의 이름 자체가 자신이 어떤 타입의 변수인지 알려주기 때문에 객체가 어떤 자원인지 쉽게 알 수 있다.

3.2.6 n 접두사는 객체의 개수를 나타내는 변수에 사용한다.

```
nPoints, nLines
```

이 표기법은 개체의 개수를 나타내는 수식의 표기법에서 차용한다.

3.2.7 No 접미사는 엔터티 번호를 나타내는 변수에 사용한다.

```
tableNo, employeeNo
```

이 표기법은 엔터티의 번호를 나타내는 수식의 표기법에서 차용한다.
또 다른 표기 방법으로는 i 을 접두사로 사용하는 것입니다: iTable, iEmployee.
이 방법은 효과적으로 변수에 iterator 로의 역할을 부여하게 됩니다.

3.2.8 Iterator 변수들은 *i, j, k* 등 과 같은 이름을 사용한다.

```
for (int i = 0; i < nTables); i++) {
  :
}

vector<MyClass>::iterator i;
for (i = list.begin(); i != list.end(); i++) {
  Element element = *i;
  ...
}
```

이 표기법은 iterator 를 나타내는 수식의 표기법에서 차용한다. j, k 등과 같은 이름의 변수들은 중첩된 반복문 내에서만 사용한다.

3.2.9 축약형 이름의 사용은 피한다.

```
computeAverage();      // NOT:  compAvg();
```

프로그래밍 언어를 구사하는 데에 있어 빈번하게 사용되는 일반적인 단어들은 절대로 축약형을 사용하지 않아야 한다.

command 대신 cmd
copy 대신 cp
point 대신 pt
compute 대신 comp
initialize 대신 init

그 다음, 도메인에서 사용하는 특수한 두문자어나 축약어들은 축약형을 그대로 사용해도 좋다. 다음과 같이 사용하면 안된다:

html 대신 HypertextMarkupLanguage
cpu 대신 CentralProcessingUnit
pe 대신 PriceEarningRatio

3.2.10 불리언 변수 이름은 절대로 부정적인(거짓인) 이름을 사용하지 않는다.

```
bool isError;      // NOT:   isNoError
bool isFound;      // NOT:   isNotFound
```

문제는 불리언 변수에 부정적인 이름을 사용할 경우, not 연산자를 사용할 경우 부정의 부정을 계산해야 하는 혼동을 초래한다. 예를 들어, !isNotError 에서 처럼, 신속하게 이것이 무엇을 의미하는지 인지할 수 없게 된다

3.2.11 예외(Exception) 클래스들은 *Exception* 이라는 접미사를 사용한다.

```
class AccessException
```

```
{
  :
}
```

사실 예외 클래스들은 프로그램의 핵심 디자인에 해당되지 않기 때문에, 이와 같은 방식으로 이름을 지정함으로써 상대적으로 클래스들 중에서 눈에 띄게(구분을 용이하게) 해준다.

3.2.12 함수(객체/결과를 리턴하는 메소드)의 이름은 '처리 후 무엇을 리턴하는지'를 의미하고, 프로시저(void 메소드)의 이름은 '무엇을 처리하는지'를 의미한다.

```
메소드가 무엇을 하는지. 의도하지 않는 것들이 무엇인지를 명확하게
해주기 때문에, 가독성을 높여준다. 또한 이 방법은 예기치 못한
부작용(side effect)들로부터 코드를 명쾌하게 유지시켜 줄 수 있다.
```

4 Files

4.1 Source Files

4.1.1 파일의 내용물은 반드시 80 컬럼을 벗어나지 않는다.

4.1.2 탭(TAB) 문자나 페이지 구분(page break) 문자와 같은 특수한 문자들은 절대로 사용하지 않는다.

```
이러한 문자들은 프로그래머가 여럿이거나 다수의 플랫폼에서 개발하는
환경에서, 편집기 별, 프린터 별, 터미널 에뮬레이터 별, 디버거 별로
지정된 설정에 따라 소스코드를 들쭉날쭉하게 만들어 가독성을
떨어뜨리게 된다.
```

4.1.3 여러 라인으로 분리한 문장들은 반드시 명확하게 만든다 [1].

```
totalSum = a + b + c +
           d + e;
function (param1, param2,
          param3);
setText ("Long line split"
         "into two parts.");
for (tableNo = 0; tableNo < nTables;
     tableNo += tableStep)
```

한 문장이 80 컬럼을 벗어날 때 라인을 분리해서 작성한다. 어떻게 라인을 분리해

야 하는지에 대한 엄격한 규칙을 제시하지 않지만 일반적으로:

- 콤마(,) 뒤에서 분리한다.
- 연산자 뒤에서 분리한다.
- 이전 라인에서의 표현식 시작 부분에 맞추어 정렬한다.

4.2 Include Files and Include Statements

4.2.1 헤더파일은 다중 include를 방지하기 위해서 반드시 헤더파일에 대한 매크로 선언문을 추가시킨다. 매크로 선언문은 모듈이름, 파일이름, 그리고 h 접미사를 조합하여 만들고 반드시 모든 문자를 대문자로 사용한다.

```
#ifndef MOD_FILENAME_H
#define MOD_FILENAME_H
 :
#endif
```

4.2.2 Include 문장은 그룹화하여 정렬시키도록 한다. 우선적으로 시스템레벨의 라이브러리나 헤더파일을 정렬시키고 나머지를 알파벳의 순서로 정렬시킨다. 그룹 사이에는 빈 라인을 하나 추가하여 쉽게 알아볼 수 있게 한다.

```
#include <fstream>
#include <iomanip>

#include <Xm/Xm.h>
#include <Xm/ToggleB.h>

#include "ui/PropertiesDialog.h"
#include "ui/MainWindow.h"
```

개별적으로 파일을 include하게 되면 이를 패키지에 표시해두고 절대패스를 사용하지 않는다.

5 Statements

5.1 Types

5.1.1 클래스들을 public, protected, private 순서대로 나열한다[2][3].

5.1.2 타입 컨버전(변환)은 반드시 명시적으로 한다. 묵시적인 타입 컨버전(변환)은 절대로 사용하지 않는다.

```
#include <fstream>
#include <iomanip>

#include <Xm/Xm.h>
#include <Xm/ToggleB.h>
```

```
#include "ui/PropertiesDialog.h"
#include "ui/MainWindow.h"
```

5.2 Variables

5.2.1 변수는 선언된 지점에서 초기화하며, 가능한 사용범위를 최소화하여 선언한다.

```
이 규칙은 어느 시점에서든 변수가 유효한 값을 가진다는 것을
보장한다. 종종 선언하는 시점에 변수에 유효한 값을 초기화하는 것이
불가능한 경우가 있지만, 그러한 경우에도 초기화하지 않은 상태로
내버려두는 것보다는 임의의 값이라도 사용하여 초기화 해두는 것이
좋다.

int x, y, z;
getCenter (&x, &y, &z);
하지만 위의 경우와 같다면 임의의 값을 사용하여 초기화하는 것보다
그냥 초기화하지 않고 둔다.
```

5.2.2 변수는 절대로 하나 이상의 의미를 가져서는 안된다.

```
무의미한 이름을 부여한 변수(scratch 변수)를 가지고 이값 저값을
할당하여 사용하게 되면, 어느 순간 이 변수가 현재 어떤 의미의 값을
가지고 있는지 모호해지는 경우가 발생한다. 일관되고 유일한 의미를
부여함으로써 가독성을 향상시키고 더불어 부작용까지 예방할 수 있다.
```

5.2.3 글로벌 변수의 사용을 최소화한다.

```
C++에서 글로별 변수를 사용할 이유가 없다. 실제로 이와 같은 효과를
내기 위해서 글로벌 함수나 파일 Scope 를 가지는 static 변수를
사용하면 된다.
```

5.2.4 클래스 변수들은 절대로 public 으로 선언하지 않는다.

```
클래스 변수의 public 선언은 c++의 정보은닉과 캡슐화 컨셉에
위배됩니다. 대신 변수를 private 으로 선언하시고 이 변수를 접근할 수
있는 메소드를 사용하게 하십시오.

한 가지 예외적인 상황은 클래스가 어떠한 행위(메소드)도 갖지 않고
오로지 데이터를 보관하는 데이터 구조로만 사용될 때입니다. 이 경우
클래스는, 즉 struct 와 동일한 형태로 사용됩니다. 이 경우에는
클래스의 인스턴스 변수들을 public 으로 선언할 수도 있습니다 [2].
```

5.2.5 동일한 타입의 변수 중 관련있는 변수들은 하나의 구문에서 선언할 수 있다.

즉, 관련이 없는 변수들을 같은 라인에서 선언하지 않는다.

```
float   x, y, z;
float   revenueJanuary, revenueFebruary, revenueMarch;
```

위의 예제를 여러 라인에 나누어 선언하는 것은 오히려 바람직하지 않다. 변수들을 그룹핑함으로써 가독성을 높일 수 있기 때문이다. 그러나 여기서 주의할 점은, 대다수의 경우에서는 이 규칙보다 각각의 변수는 선언 시 초기화 한다는 규칙이 우선적으로 적용되어야 한다.

5.2.6 C++ 포인터와 레퍼런스 선언은 타입의 이름의 옆보다 변수이름의 옆에 설정하도록 한다.

```
float *x;     // NOT:   float* x;
int    &y;     // NOT:   int&    y;
```

5.2.7 변수의 생존기간을 가능한 짧게 유지한다.

변수에 대한 조작을 작은 범위 내에 국한시킴으로써, 영향과 부작용을 통제하는 것이 용이해진다.

5.3 Loops

5.3.1 반드시 반복을 제어하는데 사용되는 문장들만 for() 구문 내에 포함시킨다.

```
sum = 0;                        // NOT: for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)   //          sum += value[i];
  sum += value[i];
```

5.3.2 반복문에 사용하는 변수는 반복문 직전에 초기화한다.

```
isDone = false;          // NOT:    bool isDone = false;
while (!isDone) {       //              :
  :                          //          while (!isDone) {
}                            //             :
                                            }
```

5.3.3 do .... while 문의 사용을 피한다.

```
여기에는 두 가지 이유가 있다.
첫번째, 제어문에 사용되는 구문 요소(키워드)들이 분산되어 있어
난잡하다. do .... while 문을 사용한 모든 문장은 while 문이나 for 문을
사용하여 동일하게 바꾸어 작성할 수 있는데 이렇게 함으로서 사용되는
구문 요소들을 최소화해서 복잡도를 줄일 수 있다.
두번째 이유는 가독성과 관련되어 있습니다. 반복의 조건을 체크하는
```

부분이 하단에 위치하는 것은, 상단에서 조건을 체크하는 반복문보다 읽기가 더욱 어렵다.

5.3.4 반복문 내에서 break 와 continue 의 사용을 자제한다.

이 두 문장은 구조화된 반복문보다 훨씬 가독성이 높을 때에만 사용한다.

5.3.5 무한루프로 반복문을 작성할 때 while(true)형태로 사용되어야 한다.

```
while (true) {
  :
}

for (;;) { // NO!
  :
}

while (1) { // NO!
  :
}
```

5.4 Conditionals

5.4.1 복잡한 조건식은 반드시 피한다. 그 대신 임시 불리언 변수를 활용하라 [1].

```
if ((elementNo < 0) || (elementNo > maxElement)||
    elementNo == lastElement) {
  :
}
```

다음과 같이 변경한다:

```
isFinished      = (elementNo < 0) || (elementNo > maxElement);
isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}
```

조건식을 불리언 변수에 할당함으로써, 프로그램 그 자체로 의미를 명확하게 전달한다. 조건문을 평가하는 구문은 보다 쉽게 해석할 수 있으며, 더불어 디버깅할 때에도 편리해진다.

5.4.2 if 문에서, 일반적인 상황은 if 블록에 위치시키고 예외 상황은 else 블록에 위치시킨다 [1].

```
isError = readFile (fileName);
if (!isError) {
  :
}
else {
  :
}
```

일반적인 처리 흐름과 예외상황 처리 흐름을 불명확하지 않게 하는 것은 가독성과 성능 모두에 영향을 미치는 중요한 사항이다.

5.4.3 조건을 평가하는 문장은 별도의 라인으로 분리한다.

```
if (isDone)              // NOT:  if (isDone) doCleanup();
  doCleanup();
```

이는 디버깅을 위한 목적으로 한 것으로 한 라인에 이것저것 기술하게 되면, 조건식 테스트가 참인지 거짓인지를 확인하기가 어려워집니다.

5.4.4 절대로 조건식에 실행문을 사용하지 않는다.

```
// Bad!
if (!(fileHandle = open (fileName, "w"))) {
  :
}

// Better!
fileHandle = open (fileName, "w");
if (!fileHandle) {
  :
}
```

조건식에 실행문을 사용하는 것은 간편하지만 읽기가 매우 어려워진다. 이 지침은 특히 C/C++개발자들에게 강조하고 싶은 것이다.

5.5 Miscellaneous

5.5.1 코드 상에서 매직 넘버(magic number: constants, array size, character positions, conversion factors와 같이 프로그램내에 등장하는 숫자/문자값)의 사용을 자제한다. 0 과 1 이외의 숫자는 차라리 상수로 정의하여 사용하도록 한다.

5.5.2 실수값 상수는 항상 소수점과 최소한 소숫점 이하 한 자리 숫자를 사용하여 지정한다.

```
double total = 0.0;    // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

특별한 경우에 정수와 실수가 동일한 값을 갖는다 하더라도 기본적으로 특성이 다르기 때문에 구별하는 것이 좋다. 또한 위 마지막 라인의 예제에서 살펴볼 수 있듯이, 값을 할당받은 변수(sum)는 다른 변수의 타입을 알 수 없다고 하더라도 명백하게 실수값을 계산해 낸다는 것을 보장할 수 있다 (* 10.0 을 통해서)

5.5.3 실수값 상수는 항상 정수부에 숫자를 사용하여 지정한다.

```
double total = 0.5;    // NOT: double total = .5;
```

5.5.4 goto문을 사용하지 않는다.

5.5.5 “NULL”대신에 0을 사용한다.

6 Layout and Comments

6.1 Layout

6.1.1 기본 들여쓰기(indentation)는 4 자로 한다.

```
for (i = 0; i < nElements; i++)
    a[i] = 0;
```

1 자짜리 들여쓰기는 너무 작아서 논리적인 코드의 레이아웃에 사용합니다. 한편 4 자 이상 들여쓰기를 할 경우 중첩 들여쓰기한 코드를 읽기가 어려워지며, 한 문장을 여러 라인으로 쪼개야하는 상황이 자주 발생하게 됩다. 2, 3, 4 글자 중에서 선택하는 것이 바람직하지만, 4자를 권고한다.

6.1.2 블록 레이아웃은 아래 예제 1 이나 예제 2 를 따르되, 예제 3 의 형태는 결코 취해서는 안된다. 클래스, 인터페이스, 메소드 블록은 예제 2 레이아웃을 따른다.

```
예제 1
while (!done) {
  doSomething();
  done = moreToDo();
}
```

```
예제 2
while (!done)
{
  doSomething();
  done = moreToDo();
}
```

```
예제 3
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

6.1.3 class 선언문은 다음의 형태를 따른다.

```
class SomeClass : public BaseClass
{
  public:
    ...
  protected:
    ...
  private:
    ...
}
```

6.1.4 함수 선언문은 다음의 형태를 따른다.

```
void someMethod()
{
  ...
}
```

6.1.5 if-else 문장은 다음과 같은 형태를 따른다

```
if (condition) {
  statements;
}

if (condition) {
  statements;
}
else {
  statements;
}

if (condition) {
  statements;
}
else if (condition) {
  statements;
}
else {
  statements;
```

```
}
```

6.1.6 for 문장은 다음과 같은 형태를 따른다.

```
for (initialization; condition; update) {
  statements;
}
```

6.1.7 비어있는 for 문장은 다음과 같은 형태를 따른다.

```
for (initialization; condition; update)
  ;
```

6.1.8 while 문장은 다음과 같은 형태를 따른다.

```
while (condition) {
  statements;
}
```

6.1.9 do-while 문장은 다음과 같은 형태를 따른다.

```
do {
  statements;
} while (condition);
```

6.1.10 switch문장은 다음과 같은 형태를 따른다.

```
switch (condition) {
  case ABC :
    statements;
    // Fallthrough

  case DEF :
    statements;
    break;

  case XYZ :
    statements;
    break;

  default :
    statements;
    break;
}
```

6.1.11 try-catch 문장은 다음과 같은 형태를 따른다.

```
try {
  statements;
}
catch (Exception &exception) {
  statements;
}
```

6.1.12 단일 문장의 if-else, for, while 문은 중괄호 없이 작성할 수도 있다.

```
if (condition)
  statement;

while (condition)
  statement;

for (initialization; condition; update)
  statement;
```

6.2 White space

6.2.1 – 모든 연산자(conventional operator)는 피연산자들 사이에 스페이스로 분리한다.

- C++ 예약어/키워드 뒤에 공백문자 하나를 추가한다.
- 콤마(,) 뒤에 공백문자 하나를 추가한다.
- 콜론(:) 의 앞뒤에는 공백문자를 추가한다.
- for 문 내의 세미콜론(;) 문자 뒤에 공백문자 하나를 추가한다.

```
a = (b + c) * d;              // NOT:   a=(b+c)*d
while (true) {                // NOT:   while(true) ...
doSomething (a, b, c, d);     // NOT:   doSomething (a,b,c,d);
case 100 :                    // NOT:   case 100:
for (i = 0; i < 10; i++) {    // NOT:   for (i=0;i<10;i++){
```

6.2.2 메소드의 이름과 메소드의 여는 괄호 사이에 공백문자를 사용한다.

```
doSomething (currentFile);    // NOT:   doSomething(currentFile);
```

6.2.3 블록 내의 논리적인 유닛들은 빈 라인을 하나 삽입하여 구분한다.

6.2.4 메소드 선언들 간에는 3개의 빈 라인을 삽입하여 구분한다.

6.2.5 선언문에서의 변수는 변수끼리 좌측정렬 한다.

```
AsciiFile  *file;
int        nPoints;
float      x, y;
```

6.2.6 문장은 가독성을 높이기 위하여 정렬한다.

```
if      (a == lowValue)     compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)    computeSomethingElseYet();


value = (potential        * oilDensity)    / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)    / constant3;


minPosition     = computeDistance (min,     x, y, z);
averagePosition = computeDistance (average, x, y, z);


switch (value) {
  case PHASE_OIL    : strcpy (string, "Oil");    break;
  case PHASE_WATER : strcpy (string, "Water"); break;
  case PHASE_GAS    : strcpy (string, "Gas");    break;
}
```

6.3 Comments

6.3.1 난잡한 코드에는 주석을 다는 것보다는, 코드를 구조화하여 재작성하고 명료한 코드에 주석을 다는 것이 낫다 [1].

```
if (condition)
  statement;

while (condition)
  statement;

for (initialization; condition; update)
  statement;
```

6.3.2 국제적 제품개발에 한해 모든 주석은 영어로 작성한다.

6.3.3 모든 주석과 여러 줄을 사용하는 주석에 //를 사용한다.

```
// 한 줄이상의 주석처리입니다.
```

6.3.4 주석은 설명하려는 코드와 같은 위치에서 들여쓰기하여 작성한다 [1].

```
while (true) {            // NOT:    while (true) {
  // Do something       //         // Do something
  something();          //            something();
}                        //        }
```

References

[1] Code Complete, Steve McConnel - Microsoft Press

[2] Programming in C++, Rules and Recommendations, M Henricson, e. Nyquist, Ellemtel (Swedish telecom)
http://www.doc.ic.ac.uk/lab/cplus/c%2b%2b.rules/

[3] Wildfire C++ Programming Style, Keith Gabryelski, Wildfire Communications Inc.
http://www.wildfire.com/~ag/Engineering/Development/C++Style/

[4] C++ Coding Standard, Todd Hoff
http://www.possibility.com/Cpp/CppCodingStandard.htm

[5] Doxygen documentation system
http://www.stack.nl/~dimitri/doxygen/index.html

[6] Geotechnical Software, "C++ Programming Style Guidelines"
http://geosoft.no/development/cppstyle.html