1.1 변수의 표기는 아래와 같은 표기법을 사용한다.

Member변수 앞에는 m_로 표기한다.
Global변수 앞에는 g_로 표기한다.
Local변수 앞에는 l_로 표기한다.
변수 이름은 모두 소문자로 작성하며 언더바를 사용할 수 있다.

1.2 각 변수 및 함수의 이름은 대문자로 시작한다.

(소문자만을 이용하거나 '_'을 이용하거나 또는 소문자 + 첫 단어 대문자를 이용하는 많은 사용 예가 있기는 하지만 결국 여러 스타일을 동일
프로젝트에서 혼용하여 사용하는 것은 통일성과 가독성 등에 저해를 가져온다. 때문에 다양한 방법 중 한가지를 선택할 필요성을 느꼈고,
첫 문자를 대문자로 쓰는 것으로 통일하였다.)

예) int nTypeMax, char szFileInfomation, (O)
int ntype_max, char szfile_infomation (X)
int GetType(); BOOL SetMaxRange(int CX, int CY), void SetFont(HFONT hFont)

함수이름은 반드시 맨 앞을 대문자로 시작한다.
접근자와 변경자는 해당하는 변수의 이름과 같은 것을 쓴다.

1.3 포인터(*) 및 참조(&) 표기는 변수 앞에 붙여서 선언한다.

(변수 앞에 붙여서 표기 하는 가장 중요한 이유는 char* szName, szType 과 같이 표기 할 경우 szType 이 포인터인지 아닌지 구분이 잘되지 않기 때문이다.)

올바른 예)
POINT *pPoint, char *szName;

```
POINT &pPoint, char &szName;

잘못된 예)
POINT* pPoint, char * szName;
POINT& pPoint, char &szName;
```

1.4 #define enum 및 const 는 대문자와 단어 사이에 '_' 를 붙인다.

```
예) #define WM_USER_PING_THREAD_END WM_USER+0x10232
    #define PACKET_LOGIN_ANSWER_BAD_USER_OVER 8
    enum BUTTON_STATE{ BUTTON_NORMAL, BUTTON_OVER,
BUTTON_DOWN, BUTTON_DISABLE };
```

1.5 클래스 내부에서 사용하는 상수 값은 클래스 내부에서 선언된 enum 을 사용한다.

(#define은 전역적으로 사용된다. 따라서 MAX_ITEM과 같은 일반적인 상수 이름은 다른 곳에서 겹칠 우려가 있기 때문에 사용하기가 꺼려진다. 하지만 enum은 클래스 내부에서만 사용되기 때문에 MAX_ITEM와 같은 일반화된 상수를 사용할 수가 있다.)

1.6 구조체 이름은 대문자로 하고 각 단어 구분은 '_' 로 한다.

(구조체의 이름은 관습적으로 대문자만을 사용한다.)

```
예) struct TEST{ … }
    struct BANNER_UNIT{ … }, struct GAME_ITEM{ … },
```

1.7 구조체의 멤버 변수는 소문자를 이용하고 단어 별 구분 시에 '_' 을 사용한다.

(헝가리안 표기법이 사용되지 않는 이유는 오랜 기간 관습적으로 소문자 표기법이 이용되었기 때문이다.)

예)

```
struct GAME_UNIT
{
    BYTE type;
    char id[13];
    WORD version;
    BYTE* buffer;
    CString ip_name,
};
```

1.8 일반적으로 지역 변수는 용도에 맞는 이름으로 표기한다.

이름을 보는 것만으로도 어떤식으로 사용되는지 알 수 있도록 표기한다.

1.9 템플릿 타입은 대문자 한 자로 표기한다.

(관습에 의하여 템플릿은 한 자로 표기한다.)

예) `template< class T>, template< class C, class D >`

1.10 클래스를 이름은 첫 자를 C 로 시작한다.

(클래스를 구분하는 헝가리안 표기법의 일종이다.)

예) CSurface, CPoint

1.11 클래스 함수의 경우 클래스 명에서 이미 명시된 부분은 제외한다.

(이미 클래스 이름으로도 충분히 알 수 있는 내용을 중복하여 적는 것은 비효율적이며 올바르지 않는 방법이다. 이것은 마치 "사각형의 사각형 가로크기는 얼마인가요?" 와 같은 표현이다.)

올바른 예) Rect.GetWidth()
잘못된 예) Rect.GetRectWidth()

1.12 역 BOOL형 변수는 사용하지 않는다. (역방향형)

(빠른 이해와 잘못된 사용을 방지하기 위해서 BOOL은 항상 '예' 인 값을 사용한다. 역 BOOL 형 변수는 '사람이 아니다가 아니다' 와 같은 복잡한 상황을 연출하기 때문이다.)

올바른 예) BOOL bError
잘못된 예) BOOL bNoError

1.13 Enum의 경우 열거형으로 사용시 변수명의 앞에 표기한다. (단 클래스 내부에서 #define의 대용으로 사용시에는 예외이다.)

(enum을 열거형으로 사용시 타입을 항상 표기함으로써 값들이 사용시 어떠한 열거형 인지 구분을 쉽게 해준다.)

예)

```
enum COLOR
{
    COLOR_RED,
    COLOR_GREEN,
    COLOR_BLUE,
};
```

1.14 함수명의 경우 동사가 앞에 오고 명사가 뒤에 오도록 한다.

(영문법에 맞는 규칙으로 통일되게 사용한다.)

예) GetLine(), PlaySound(), MoveWindow()

1.15 BMP나 MP3, ID, XML등의 약어의 표현은 대문자로 한다.

예) userID, userPW, MyMP3

2.1 bool, true, false 는 사용하지 않고 BOOL, TRUE, FALSE 로 사용한다.

(BOOL등은 4바이트 자료형이고 bool등은 1비트(실제로는 1바이트) 자료형이다. 32비트 운영체계의 경우 속도와 비교하는데 걸리는 성능 부분에도 유리하고 알아보기도 좋기 때문에 위와 같은 대문자로 된 자료형을 사용한다.)

2.2 한 라인이 너무 길지 않도록 주의한다.

(정확히 컬럼의 크기가 정해진 것은 아니지만 일반적인 프로젝트 팀에서 사용하는 일반적인 해상도(1024~1280)에서 한눈에 보기 좋도록 작성한다.)

2.3 함수의 인자는 NULL 입력이 가능하다면 포인터(*)를 이용하고 그렇지 않다면 참조(&)을 사용한다.

(포인터 방식의 최대 단점은 NULL을 원하지 않은 경우에도 NULL이 입력될 수 있다는 점이다. 따라서 이러한 것을 근본적으로 막기 위해서는 NULL이

입력되지 않기를 원하는 함수의 인자는 참조를 사용한다.)

예) int Draw(CSurface &rSurface); (NULL이 입력 불가한 경우)
void GetValue(int *pId, int *pAge, int *pDate = NULL); (NULL이 입력 가능한 경우)

2.4 STL 사용시 std:: 는 사용하지 않는다.

(프로젝트에서 STL을 표준으로 사용하기를 결정했다면 'using std;'을 사용한다.)

올바른 예) std::list List
잘못된 예) list List

2.5 지역변수는 필요한 시점에서 선언한다.

(C++에서는 변수를 미리 선언할 필요성이 없다. 단 기능이나 사용이 비슷한 변수들이 선언된다면 가독성 증가를 위해서 미리 선언 될 수 있다.)

2.6 매직 넘버(Magic Number = 단순 숫자)는 가능한 사용하지 않는다.

(가능한 #define 이나 enum, static const 등을 이용하여 매직 넘버의 사용을 최소화한다.
개수, 최대값, 최소값, 범위 등의 매직 넘버는 사용을 최대한 자제하나 오히려 가독성과 사용용도가 떨어지는 이미지의 좌표값과 같은 일시적인 것은 특별히 사용을 회피 할 필요는 없다. 또한 0, 1, -1의 경우는 매직 넘버에서 예외가 될 수 있다.)

2.7 goto의 사용을 자제한다.

(goto의 사용은 프로그램의 구조를 파악하기 어렵게 한다. 하지만 매우 특수한 경우에는 사용 될 수 있다.)

2.8 전역 변수나 멤버 변수보다는 지역변수를 사용한다.

(전역 변수와 멤버 변수는 매우 오랜 기간 동안 생존해있고 중요한 변수이기 때문에 작성자를 포함한 프로그래머들은 그 변수들을 분석하고 기억하려고 한다. 따라서 사용을 최소화 한다면 그러한 수고가 줄어든다.)

2.9 포인터, 핸들은 NULL과 비교하고 숫자는 0과 비교한다.

2.10 복잡한 구문은 임시 변수를 사용한다.

(부적절한 3항 연산자나 각종 비교구문의 중첩을 이용한 복잡한 코드는 가독성을 저해한다.)

올바른 예)

```
BOOL bCompare1 = ( nCount >100 &&nSize <50 );
BOOL bCompare2 = ( nCount == 50 || nSize <20 );
if( bCompare1 || bCompare2 )…
```

잘못된 예)

```
if( ( nCount >100 &&nSize <50 ) || ( nCount == 50 || nSize <20 ) )…
```

3.1 클레스의 기본 구조는 다음과 같다.

(생성자, 공개함수, 비공개함수, 멤버변수의 순서로 배열하고 각각은 public,

protected 등으로 추가로 명기하여 구분한다.)

```
예)
class CSound
{
public:          : public 함수를 최 상단에 배치한다.
CSound();      : 생성자를 최 상단에 배치한다.
virtual ~CSound();          : 종결자는 일반적으로 virtual 이다.

void Play();          : 멤버 함수
void Stop();
         : public, protected 등은 1라인 비운다.
protected:              : protected 함수를 배치한다.
void PlayWav();
void PlayOgg();
                 : 각각의 함수별 관련성 및 블록이 있다면 1라인 비운다.
void StopWav();
void StopOgg();

protected:              : 멤버 변수를 배치한다.
char m_szFileName[256];
int m_nSize;
};
```

3.2 모든 멤버 변수는 public으로 선언하지 않는다.

(정보은닉과 캡슐화를 위해서 멤버 변수들은 public으로 선언하지 않는다.)

3.3 inline 함수의 경우 헤더 선언에 inline 을 표기하지 않는다.

(헤더 파일에 inline으로 명기하여 선언을 하는 것과 별개로 실제 구현에 inline으로 작성하면 함수는 inline으로 작동된다. 클래스 헤더를 잘 알아볼

수 없게 하는 inline은 굳이 사용 할 필요성이 없다.)

예)
```
class CSound
{
올바른 예)      BOOL Play();
잘못된 예)      inline BOOL Play();
};
```

3.4 inline 함수의 경우 선언구문 1라인 아래 줄에 입력한다.

(inline 함수는 헤더 파일의 하단에 선언하고 너무 많은 inline 함수가 존재한다면 *.ini 파일을 작성하여 헤더에서 include 한다.)

예)
```
class CSound
{
public:
    BOOL CSound ::Play();
};

inline BOOL CSound ::Play()
{
...
}
```

또는

```
class CSound
{
public:
    BOOL CSound ::Play();
};
```

```
#include "Sound.inl"
```

3.5 클래스 선언문에 어떠한 함수라도 구현을 하지 않는다.

(클래스의 선언문은 매우 중요한 공간이다. 그 공간에 함수의 구현이 들어간다면 함수의 이름들을 한눈에 볼 수 없어 가독성이 매우 떨어지게 된다. 따라서 모든 구현은 cpp 파일 및 별개의 인라인 함수로 구현한다.)

```
잘못된 예)
class CTest
{
BOOL Play()
{
...
}
};
```

3.6 클래스의 생성자 및 종결자는 inline함수로 구현하지 않는다.

(성능향상을 위한 몇몇 경우를 제외하면 생성자와 종결자는 호출되는 회수는 매우 적다. 따라서 inline으로 구현하지 않는다.)

3.7 종결자는 일반적으로 virtual로 선언한다.

(종결자가 virtual이어야 상속될 경우 등에서 올바르게 종결자가 작동된다. 따라서 상속이 되지 않기를 바라는 경우와 클래스 크기를 줄이기 위한 경우 및 성능향상을 위한 경우를 제외하고 가상함수로 선언한다.)

4.1 클래스명과 파일명은 'C'를 제외한 동일한 이름을 가진다.

예)클래스명 : CFileName
파일명 : FileName.h, FileName.cpp

4.2 1클래스 2파일(.h .cpp)을 가진다.

4.3 클래스의 복잡도에 따라서 여러 개의 cpp 파일을 가질 수 있다.

(종종 멤버 함수가 많다면 함수의 검색 및 수정 등이 불편하여 관리가 어려워진다. 그러한 경우 비슷한 함수 별로 파일을 만든다면 쉽게 관리가 가능하다.)

예) 클래스명 : CSurfaceGdi
파일명 : SurfaceGdi.h, SurfaceGdiPut.cpp SurfaceGdiInit.cpp, SurfaceGdiEffect.cpp

4.5 매우 간단하면서 비슷한 여러 개의 클래스 및 구조체의 경우 1개의 파일에 포함될 수 있다.

(작은 클래스, 구조체마다 각각의 파일을 가진다면 파일개수의 복잡성이 증가한다.)

4.6 문서(기획서, UML등) 파일은 소스파일과 다른 특정 디렉토리를 사용한다. (예:Document)

(각종 문서들을 소스파일과 함께 관리한다면 더욱더 복잡한 파일목록들이 만들어질 것이다.)

4.7 일정 규모이상의 프로젝트의 시작은 폴더구조를 설계한 이후 시작한다.

(프로젝트 시작 시 폴더의 구조와 각 폴더 별 연관성 등을 고려하여 폴더구조를 정의하고 상호 참조 관계 등을 정의하는 것으로 프로젝트를 시작해야 한다.)

예)
- [App]
- [Common]
- [Core]
- [Sound]
- [Image]
- [Document]

4.8 #include 시 상대주소를 사용한다. 예) #include '../BugGame/BugMain.h'

(절대주소를 사용한다면 다른 프로그래머들은 컴파일이 안될 수도 있기 때문이다.)

4.9 #include 는 시스템 헤더 -> 라이브러리 -> 자체 코드 등의 순서로 작성한다.

(#include 또한 상위 -> 하위 관계를 지키도록 하면 가독성이 높아진다.)

예)
```
#include 'stdio.h'
#include 'afxinet.h'
#include 'MyImageEngine.h'
#include 'MyFile.h'
```

4.10 프로그램 코드, 이미지, 사운드를 포함한 모든 소스는 Github를 사용하여 관리한다.

(다른 프로그래머가 소스 관리을 이용하더라도 컴파일과 실행을 할 수 있게 한다.)

∴ ∴

5.1 TAB의 크기는 4글자로 한다.

5.2 함수마다 1라인을 띄운다.

5.3 소괄호의 사용은 다음과 같이 공백입력을 한다. (함수, if, while, for, switch, 형 변환 등 모두 동일함)

```
예)
if( bSuccess == TRUE ) …
int DoSomething( int nParam1, int nParam2 );
for( int I = 0; I <nCount; I++ )
int nIntSize = sizeof( int );
int nValue = ( int ) byValue;
```

5.4 중괄호의 사용은 다음과 같은 형식으로 한다. (함수, if, while, for, switch 모두 동일함)

```
올바른 예)
if( bSuccess == TRUE )
{
```

```
    DoSomething();
    ...
}

int DoSomething()
{
    ...
}

잘못된 예)
if( bSuccess == TRUE ){
    DoSomething();
    ...
}

if( bSuccess == TRUE )
    {
    DoSomething();
    ...
    }
```

5.5 한 줄 조건문 및 반복문은 다음과 같이 모두 사용 할 수 있다. 하지만 동일 구문에 여러 스타일을 혼용하여 사용하면 안된다.

```
올바른 예)
if( bSuccess == TRUE )
{
    DoSomething();
}
else
{
    DoSomething();
}
```

잘못된 예)

```
if( bSuccess == TRUE ) DoSomething();
else
{
    DoSomething();
}

if( bSuccess == TRUE ) DoSomething();
else
    DoSomething();

if( bSuccess == TRUE )
{
    DoSomething();
}
else    DoSomething();
```

5.6 'return' 의 윗라인은 1줄 여백으로 한다.

올바른 예)

```
BOOL IsSuccess()
{
    BOOL bRtn = m_bRtn;

    return bRtn;
}
```

잘못된 예)

```
BOOL IsSuccess()
{
    BOOL bRtn = m_bRtn;
    return bRtn;
```

```
}
```

5.7 중괄호를 닫는 부분과 if, for등의 조건, 반복 구문에서는 1줄 여백을 가진다.

(적절한 줄 여백은 코드를 읽기 쉽게 한다. 참고로 if-else구문은 else에만 적용한다.)

```
올바른 예)
if( bSuccess == TRUE )
{
    ...
}
else
{
    ...
}

nSum += nCount;

잘못된 예)
if( bSuccess == TRUE )
{
    ...
}
nSum += nCount;
```

: :

6.1 무한 루프 구문은 while(TRUE)를 사용 한다.

```
올바른 예)
```

```
while( TRUE )
{
}
```

6.2 BOOL및 HRESULT 등의 리턴 값을 TRUE, FALSE, SUCCEED(), FAILED()등 과 비교한다.,

(코드의 간소함보다 '!' 이 눈에 잘 띄지 않으므로 오히려 가독성을 저해하고 코드 실수를 유발할 가능성이 크므로 이와 같이 변경하였다.)

올바른 예)

```
if( IsMain() == TRUE ) …
if( IsMain() == FALSE ) …
```

잘못된 예)

```
if( IsMain()) …
if( !IsMain()) …
```

7.1 주석은 모국어(한국어)로 작성한다.

7.2 주석은 Doxygen 표준을 사용한다.

7.3 복잡성에 따라서 세부적인 설명 및 예제를 포함할 수 있다.

7.4 일반적인 주석의 경우 /*, */ 보다는 가능한 // 으로 주석을 사용한다.

(Doxygen 의 경우를 예외로 한다.)

7.5 주석은 내용의 앞 라인과 같은 컬럼에 기록한다.

(몇몇 특수한 상황에서 라인 뒤에 입력하는 경우도 있다.)

올바른 예)

```
// 성공했다면…
if( bSuccess == TRUE )
{
    // 어떤일을 한다.
    DoSomethinh();
}
```

잘못된 예)

```
// 성공했다면…
if( bSuccess == TRUE )
{
// 어떤일을 한다.
    DoSomethinh();
}

if( bSuccess == TRUE ) // 성공했다면…
{
    DoSomething(); // 어떤일을 한다. (X)
}
```

일반적으로 생성, 초기화하는 부분은 BOOL 을 리턴 하고 파괴, 제거, 닫는 부분은 void 형 함수로 처리한다.

권장하는 함수명 뒤에 CreateImage, OpenFile과 같이 각종 명사를 붙여서 사용 할 수 있다.

함수명	용도
Create / Destroy	생성 / 파괴
Open / Close	열다 / 닫다
Init / Final	초기화 / 마지막
Play / Stop	시작 / 정지
Get / Set	얻음 / 넣음
Add / Remove	추가 / 삭제 (일반적으로 순서는 상관 없다)
Insert / Delete	(중간에) 삽입 / (중간에) 삭제
Start / Stop	시작 / 정지
Suspend / Resume	일시 정지 / 재 시작
Begin / End	시작 / 끝
First / Last	처음 / 끝
Next / Previous	이전 / 다음
Increment / Decrement	증가 / 감소
Old / New	이전 / 새로
Up / Down	상 / 하
Min / Max	최소 / 최대
Show / Hide	보여줌 / 가림