Effective C++

# Chapter.1

## 2: #define 대신 const, enum, inline

**선행 처리자(매크로)보다 컴파일러를 더 가까이 하는 것이 좋다**

#define은 C++ 언어 자체의 일부가 아닌 것으로 취급될 수 있다

|  |
| --- |
| #define ASPECT\_RATIO 1.653 |

- ASPECT\_RATIO가 기호식 이름으로 보이지만 컴파일러에서는 보이지 않는다  
-> 컴파일러에 넘어가기 전 선행 처리자가 숫자 상수로 바꿔버린다

= ASPECT\_RATIO는 컴파일러가 쓰는 기호 테이블에 들어가지 않게 된다  
= 숫자 상수로 대체된 코드에서 컴파일 에러가 발생한다면 헷갈릴 수 있다

에러 메시지에는 ASPECT\_RATIO가 아닌 1.653이 나온다  
- ASPECT\_RATIO를 작성한 사람이 본인이 아닌 경우 더 곤란해진다

**해결 방법: 매크로 대신 상수를 쓴다**

ex) const double AspectRatio = 1.653;

const로 작성할 경우 언어 차원에서 지원하는 상수 타입의 데이터기 때문에 당연히 컴파일러에도 보이며 기호 테이블에도 들어간다.

상수가 부동 소수점 실수 타입일 경우 컴파일을 거친 최종 코드의 크기가 #define을 쓴 경우보다 작게 나올 수 있다.  
- 매크로를 쓰는 경우 코드에 ASPECT\_RATIO가 등장하면 선행 처리자가 1.653으로 바꾸면서 코드 안에 1.653의 사본이 등장 횟수만큼 들어간다  
- 상수 타입을 쓰는 경우 AspectRatio는 여러 번 쓰여도 사본은 단 1개만 생긴다.

**매크로를 상수로 교체하는 경우 주의 사항**

**1: 상수 포인터를 정의하는 경우**

상수 정의는 일반적으로 헤더 파일에 넣는 것이 상례로 포인터는 꼭 const로 선언해줘야 한다  
- 추가로 포인터가 가키리는 대상가지 const로 선언하는 것이 보통  
- 헤더 파일 안에 char\* 기반의 문자열 상수를 정의한다면 2번 써야 하는 것

|  |
| --- |
| **const** char\* **const** authorName = "Scott Meyers"; |

- 문자열 상수를 쓰는 경우 char\* 기반의 구형 문자열 보다 string객체가 더 사용하기 좋다

|  |
| --- |
| **const** std::string authorName("Scott Meyers"); |

**2: 클래스 멤버로 상수를 정의하는 경우**

클래스 상수를 정의하는 경우로 어떤 상수의 유효범위를 클래스로 한정하고자 하는 경우 그 상수를 멤버로 만들어야 하는데 그 상수의 사본 개수가 한 개를 넘기지 않게 하려면 정적 멤버로 만들어야 한다

|  |
| --- |
| class GamePlayer  {  private:  static const int NumTurns = 5; // 상수 선언  int scores[NumTurns]; // 상수 사용  ….  }; |

- NumTurns는 선언(declaration)으로 정의가 아니다  
- C++에서는 사용하고자 하는 것에 대해 정의가 있어야 하는 것이 일반적  
- 정적 멤버로 만들어지는 정수 타입의 클래스 내부 상수는 예외  
- 정수 타입의 클래스 내부 상수는 주소를 사용하지 않는 한 정의 없이 선언만 해도 문제없다

\* 클래스 상수의 주소를 구하거나 컴파일러의 문제로 정의를 요구하는 경우 별도의 정의를 제공해야 한다

|  |
| --- |
| const int GamePlayer:;NumTurns; |

- 클래스 상수의 정의는 구현 파일에 둔다(헤더 파일에는 두지 않는다)  
- 정의에는 상수의 초기 값이 있어서는 안된다  
- 클래스 상수의 초기 값은 해당 상수가 선언된 시점에서 바로 주어진다  
= NumTurns는 선언될 당시에 바로 초기화된다.

클래스 상수는 #define으로 만들 수 없다

- #define의 유효 범위를 알 수 없다  
- 일단 매크로는 정의되면 컴파일이 종료될 때까지 유효하다  
- 단 #undef가 호출되면 거기까지만 유효하다

**결론: #define은 클래스 상수를 정의하는데 쓸 수 없으며 어떠한 형태의 캡슐화 혜택도 볼 수 없다**.  
- private 성격의 #define은 존재하지 않는다.

**\* 상수 데이터 멤버는 캡슐화가 가능하다**

오래된 컴파일러의 경우 상수의 선언만으로는 사용할 수 없는 경우가 있다  
- 정적 클래스 멤버가 선언된 시점에 초기 값을 주는 것이 맞지 않다고 판단하기 때문  
- 클래스 내부 초기화를 허용하는 경우가 정수 타입의 상수에만 국한되어 있기 때문이기도 함

이럴 경우 아래와 같이 작성해야 한다.

|  |
| --- |
| class CostEstimate  {  private:  static const double FudgeFactor; // 정적 클래스 상수의 선언  }; |
| const double CostEstimate::FudgeFactor = 1.35; // 정적 클래스 상수의 정의 |

- 일반적인 경우라면 이것으로 대부분 통과된다

**단 하나의 예외사항: 클래스를 컴파일하는 도중에 클래스 상수의 값이 필요할 때**

GamePlayer::Scores 등의 배열 멤버를 선언할 대가 대표적인 예  
- 컴파일러는 컴파일 과정에서 이 배결의 크기를 알아야 한다면서 오류를 내보낸다

**해결 방법: 나열자 해킹 기법**

원리는 나열자 타입의 값은 int가 놓일 곳에도 쓸 수 있다

|  |
| --- |
| class GamePlayer  {  private:  enum {NumTurns = 5}; // 나열자 해킹 기법으로 NumTurns를 기호식 이름으로 만든다  int scores[NumTurns]; // 동작한다  }; |

나열자 해킹 기법은 여러 이유로 도움이 된다

**1: 나열자 해킹 기법은 동작 방식이 const보다는 #define에 더 가깝다**

- const의 주소를 잡아내는 것은 합당하지만 enum의 주소를 취하는 일은 불법이고 #define의 주소를 얻는 것 역시 맞지 않다  
- 선언한 정수 상수를 가지고 다른 사람의 주소를 얻거나 참조자를 쓰는 것이 싫다면 enum을 사용하는 것이 도움이 된다.  
- 제대로 만들어진 컴파일러는 정수 타입의 const 객체에 대해 저장공간을 준비하지 않지만 반대로 동작하는 컴파일러도 존재하기 때문에 두 컴파일러 모두 안전하게 const 객체에 대한 메모리를 만들지 않는 방법으로 만들고자 할 때  
- enum은 #define처럼 불필요한 메모리 할당을 하지 않는다

**2: 상당히 많은 코드에서 이 기법을 사용한다.**

이러한 경우 쉽게 알아보기 위해 눈에 익혀두는 것

+ 나열자 해킹 기법은 템플릿 메타 프로그래밍의 핵심 기법이기도 함

**#define 지시자의 오용 사례: 매크로 함수**

함수처럼 보이지만 함수 호출 오버헤드를 일으키지 않는 매크로를 구현하는 것

|  |
| --- |
| #define CALL\_WITH\_MAX(a,b) f((a) > (b) ? (a) : (b)) |

- 매크로 함수는 많은 단점이 존재한다

매크로를 작성할 때는 매크로 본문에 있는 인자마다 반드시 괄호를 씌워줘야 한다  
- 이것이 되어있지 않다면 표현식을 매크로에 넘길 때 골치 아픈 일이 발생한다

|  |
| --- |
| int a = 5, b= 0;  CALL\_WITH\_MAX(++a, b); // a가 2번 증가  CALL\_WITH\_MAX(++a, b+10); //a가 1번 증가 |

- f가 호출되기 전에 a가 증가하는 횟수가 달라진다  
- 비교를 통해 처리한 결과가 어떤 것이냐에 따라 달라지는 것

**인라인 함수에 대한 템플릿**

기존 매크로의 효율을 그대로 유지함은 물론 정규 함수의 모든 동작방식 및 타입 안전성까지 완벽하게 만들 수 있는 방법

|  |
| --- |
| template<typename T>  inline void callWithMax(const T& a, const T& b)  {  f(a > b ? a : b);  } |

- 템플릿 함수이기 떄문에 동일 계열 함수군을 만들어낸다  
- 동일한 타입의 객체 두 개를 인자로 받고 둘 중 큰 것을 f에 넘겨서 호출하는 구조  
- 함수 본문에 여러 개의 괄호를 사용할 필요 없고 인자를 여러 번 평가할지 모르는 걱정도 필요 없다  
- 임의의 클래스 안에서만 쓸 수 있는 인라인 함수에 대해서도 동작할 수 있다  
= 매크로의 경우에는 이럴 수 없다

정리

1: 단순한 상수를 쓸 때는 #define보다 const 객체 혹은 enum을 우선시 한다

2: 함수처럼 쓰이는 매크로를 만들려면 #define 매크로보다 인라인 함수를 우선 시 한다.

## 3: const를 자주 사용한다

const는 의미적인 제약을 소스 코드 수준에서 붙인다는 점과 컴파일러가 이 제약을 단단하게 지켜준다는 장점이 있다  
- const 키워드가 붙은 객체는 외부 변경을 불가능하게 한다

어떤 값이 불변이어야 한타는 제작자의 의도를 컴파일러 및 다른 프로그래머와 나눌 수 있는 수단이 된다  
- 값이 변하면 안 되는 것이 맞다면 이는 지켜져야 한다  
- 컴파일러가 제약 위반을 막는 일에 거들 수 있도록 const를 사용하는 것

const의 기능  
- 클래스 외부에서 전역 또는 네임 스페이스 유효 범위의 상수를 선언(정의)하는데 사용할 수 있다.  
- 파일, 함수, 블록 유효 범위에서 static으로 선언한 객체도 const를 붙일 수 있다.  
- 클래스 내부에서 정적 멤버, 비정적 멤버 모두를 상수로 선언할 수 있다.  
- 포인터의 경우 포인터 자체를 상수로 만들거나 포인터가 가리키는 데이터를 상수로 지정할 수 도 있으며 둘 다 지정할 수 도 있다.

|  |  |
| --- | --- |
| char greeting[] = "Hello"; |  |
| char \*p = greeting; | 비상수 포인터와 비상수 데이터 |
| const char \*p = greeting; | 비상수 포인터, 상수 데이터 |
| char \* const p = greeting; | 상수 포인터, 비상수 데이터 |
| const char \* const p = greeting; | 상수 포인터, 상수 데이터 |

- const가 **\*의 왼쪽**에 있으면 **포인터가 가리키는 대상이 상수**  
- const가 **\*의 오른쪽**에 있으면 **포인터 자체가 상수**  
- const가 \*의 양쪽에 있으면 포인터와 포인터가 가리키는 대상이 상수

|  |
| --- |
| void f1(const Widget \*pw); // f1은 상수 Widget 객체에 대한 포인터를 매개변수로 한다  void f2(Widget const \*pw); // f2도 동일하다 |

- **\*의 왼쪽에만 있으면 const가 타입의 어디에 위치해도 의미는 동일하다**.  
- 쓰는 사람 마음이지만 의미는 동일하므로 눈에 익혀둔다.

STL 반복자는 포인터를 본 뜬 것으로 기본적인 동작 원리가 T\* 포인터와 비슷하다  
- 어떤 반복자를 const로 선언하는 일은 포인터를 상수로 선언하는 것 즉, T\* const 포인터와 같다  
- 반복자는 자신이 가리키는 대상이 아닌 것을 가리키는 경우가 허용되지 않는다  
- 반복자가 가리키는 대상 자체는 변경이 가능하다  
= **변경이 불가능한 객체를 가리키는 반복자가 필요하다면 const\_iterator를 쓰면 된다.**

|  |
| --- |
| const std::vector<int>::iterator iter = vec.begin();  // iter은 T\* const와 동일하게 동작한다(포인터의 상수화)  \*iter = 10;  **++iter; // iter은 상수로 변경이 불가능하다** |
| std::vector<int>::const\_iterator cIter = vec.begin();  // cIter는 const T\* 처럼 동작한다(데이터의 상수화)  **\*cIter = 10; // \*cIter는 상수로 변경이 불가능하다**  ++cIter; // cIter 자체가 변경되는 것으로 이것은 가능하다. |

**가장 강력한 const의 용도는 함수 선언에 사용하는 것**

함수 선언문에 있어서 const는 함수의 반환 값, 각 매개변수, 멤버 함수 앞에 붙을 수 있으며 함수 전체에 대해 const의 성질을 붙일 수 있다.

|  |
| --- |
| 함수 반환 값을 상수로 지정 |
| class Rational { … };  const Rational operator\* (const Rational& lhs, const Rational& rhs); |

- 함수 반환 값을 상수로 정해주면 안전성이나 효율을 포기하지 않고도 사용자 측의 에러 돌발 상황을 줄이는 효과를 꽤 자주 볼 수 있게 된다.

|  |
| --- |
| Rational a, b, c;  …  (a \* b) = c; // a\*b의 결과에 =를 호출하게 된다 |
| if (a \* b = c) … |
| 기본 타입이면 문법 위반에 걸리지만 사용자 정의 타입들은 이것을 잡아주지 않는다 |

- operator\* 의 반환 값을 const로 지정해 놓으면 사용자 정의 타입들에 대해 나타나는 문법 오류를 조금 줄여줄 수 있다.  
- "=="를 잘못쳐서 "="로 입력한 경우와 같은 컴파일 에러를 줄여줄 수 있게 된다.

const 매개변수는 그냥 const 타입의 지역 객체와 특성이 동일하다  
- 가능한 많이 사용하면 된다.

**상수 멤버 함수**

멤버 함수에 붙는 const 키워드의 역할은 **해당 멤버 함수가 상수 객체에 대해 호출될 함수**라는 것을 알려주는 것

중요성

**1: 클래스의 인터페이스를 이해하기 좋게 하기 위해서**

클래스로 만들어진 객체를 변경할 수 있는 함수는 무엇이고 변경할 수 없는 함수는 무엇인가를 사용자 쪽에서 알고 있어야 하는 것

**2: 키워드를 통해 상수 객체를 사용할 수 있게 하기 위해서**

코드의 효율을 위해 아주 중요한 부분  
- C++ 프로그램의 실행 성능을 높이는 핵심 기법 중 하나가 객체 전달을 **상수 객체에 의한 참조자**(reference-to-const)로 진행하는 것  
- 이 기법이 제대로 동작하려면 상수 상태로 전달된 객체를 조작할 수 있는 const 멤버 함수, 상수 멤버가 준비되어 있어야 한다

**const 키워드가 있고 없고의 차이만 있는 멤버 함수들은 오버로딩이 가능하다  
- C++의 아주 중요한 성질**

|  |
| --- |
| class TextBlock {  public:  const char& operator[] (std::size\_t position) const // 상수 객체에 대한 operator[]  { return text[position]; }  char& operator[] (std::size\_t position) // 비상수 객체에 대한 operator[]  { return text[position]; }  private:  std::string text; }; |
| TextBlock tb("Hello");  std::cout << tb[0]; // TextBlock::operator[]의 비상수 멤버를 호출한다 |
| const TextBlock ctb("World");  std::cout << ctb[0]; // TextBlock::operator[]의 상수 멤버를 호출한다 |

실제 프로그램에서 상수 객체가 생기는 경우  
- 상수 객체에 대한 포인터  
- 상수 객체에 대한 참조자로 객체가 전달될 때

|  |
| --- |
| void print(const TextBlock& ctb) // 이 함수에서 ctb는 상수 객체로 쓰인다.  { std::cout << ctb[0]; } |

operator[] 를 오버로드 해서 각 버전마다 반환 타입을 다르게 만들었다면 TextBlock의 상수 객체와 비상수 객체의 쓰임이 달라진다.

|  |  |
| --- | --- |
| std::cout << tb[0]; | 비상수 버전의 TextBlock 객체를 읽는다. |
| tb[0] = 'x'; | 비상수 버전의 TextBlock 객체를 쓴다 |
| std::cout << ctb[0]; | 상수 버전의 TextBlock 객체를 읽는다 |
| ctb[0] = 'x'; | 컴파일 에러 (상수 TextBlock 객체에 대해 쓰기 불가) |

- 마지막의 컴파일 에러는 operator[]의 반환타입으로 인해 생긴 컴파일 에러다  
- operator[] 호출 자체가 잘못된 것은 없고 const char& 타입에 대입 연산을 시도하였기 때문에 발생한 에러

operator[]의 비상수 멤버는 char의 참조자를 반환한다

|  |
| --- |
| char& operator[] (std::size\_t position)  { return text[position]; } |

- char 하나만 쓰면 안 되고 char&를 반환해야 한다.  
- char만 반환할 경우 tb[0] = 'x';가 컴파일될 수 없다  
= **기본제공 타입을 반환하는 함수의 반환 값을 수정하는 것은 불가능하다**

기본 제공 타입을 반환하는 함수의 반환 값을 수정할 수 있어도 C++의 반환 시 값에 의한 반환을 수행해야 하는 성질로 인해 불가능하다  
- 결국 수정되는 값은 tb.text[0]의 사본  
- tb.text[0] 자체가 수정될 수는 없다

**비트 수준 상수성(물리적 상수성)**

어떤 멤버 함수가 그 객체의 어떤 데이터 멤버도 건드리지 않아야 const로 인정하는 개념  
- 정적 멤버는 제외한다  
- 그 객체를 구성하는 비트들 중 어떤 것도 변경해서는 안 된다는 것

비트 수준 상수성을 사용하면 상수성 위반을 발견하는 것이 힘들지 않다  
- 컴파일러는 데이터 멤버에 대해 대입 연산이 수행되었는지 보면 된다  
- C++에서 정의하고 있는 상수성이 비트 수준 상수성

상수 멤버 함수는 그 함수가 호출된 객체의 어떤 비정적 멤버도 수정할 수 없게 되어 있다.

제대로 const로 동작하지 않는데 비트 수준 상수성 검사를 통과하는 멤버 함수들이 적지 않다  
- 어떤 포인터가 가리키는 대상을 수정하는 멤버 함수들 중 상당 수가 이러한 경우에 속함  
- 포인터가 객체의 멤버로 들어있는 한 이 함수는 비트 수준 상수성을 갖는 것으로 판별된다.  
- 컴파일러가 이에 대해서 오류를 출력해주지 않는다

이로 인해 상식적으로 이해할 수 없는 동작이 생길 수 있다.

|  |
| --- |
| class CTextBlock {  public:  char& operator[] (std::size\_t position) const // 부적절한 operator[] 선언  { return pText[position]; } // 비트 수준 상수성이 있어서 부적절하지만 허용된다.  private:  char\* pText;  } |

- operator[] 함수가 상수 멤버 함수로 선언되어 있다(부적절한 operator[] 선언)  
- 부적절하지만 해당 객체의 내부 데이터에 대한 참조자를 반환한다  
- pText 자체는 건드리지 않아 컴파일러가 이 operator[]를 오류로 잡지 않는 것

|  |
| --- |
| const CTextBlock cctb("Hello"); // 상수 객체 선언  char \*pc = &cctb[0];  // 상수 버전의 operator[]를 호출하여 cctb의 내부 데이터에 대한 포인터를 얻는다.  \*pc= 'J'; // cctb가 "Jello"가 된다. |

- 어떠한 값으로 초기화된 상수 객체를 하나 만들어 놓고 이것에 상수 멤버 함수를 호출하니 값이 변한 것

**논리적 상수성**

비트수준 상수성을 보완하는 대체 개념으로 나오게 된 상수성의 개념  
- 객체의 일부 비트 정도는 바꿀 수 있되 사용자 측에서 알아채지 못하게만 한다면 상수 멤버 자격이 있다는 것

|  |
| --- |
| class CTextBlock {  public:  std::size\_t length() const;  private:  char \*pText;  std::size\_t textlength; // 직전에 계산한 텍스트 길이  bool lengthIsValid; // 텍스트 길이가 유효한지 판정하는 bool 값  }; |
| std::size\_t CTextBlock::length() const  {  if(!lengthIsValid) {  textLength = std::strlen(pText);  lengthIsValid = true;  }  return textLength;  } |

- 상수 멤버 함수 안에서는 textLength 및 lengthIsValid에 대입할 수 없다.

length() 함수의 구현은 비트 수준 상수성과는 거리가 멀다

CTextBlock의 상수 객체에 대해서는 당연히 문제가 없어야 할 코드지만 컴파일러는 에러를 출력한다  
- 컴파일러의 비트 수준 상수성 검사를 통과해야 하기 때문

**mutable: 컴파일러의 비트 수준 상수성을 통과하기 위한 방법**

|  |
| --- |
| class CTextBlock {  public:  std::size\_t length() const;  private:  char\* pText;  mutable std::size\_t textlength;  mutable bool lengthIsValid;  }; |
| std::size\_t CTextBlock::length() const  {  if(!lengthIsValid) {  textLength = std::strlen(pText);  lengthIsValid = true;  }  reutnr textLength;  } |

- mutable 선언을 할 경우 어떠한 순간에도 수정이 가능하다  
- 상수 멤버 함수 안에서도 수정이 가능해진다.

**상수 멤버, 비상수 멤버 함수에서 코드 중복 현상을 피하는 방법**

mutable 키워드가 비트 수준 상수성 문제를 해결해주는 좋은 방법이기는 하지만 const와 관련된 모든 문제를 해결해주지는 않는다.  
- 만약 operator[] 함수가 특정 문자의 참조자만 반환하는 것이 아닌 다른 일들을 하게 된다고 가정한다  
- 이런 저런 코드를 모조리 operator[] 함수의 상수/비상수 버전에 넣는다면 코드가 너무 복잡하고 길어지게 된다.

|  |
| --- |
| class TextBlock {  public:  const char& operator[] (std::size\_t position) const  {  … // 다른 동작 추가  return text[position];  }  char& operator[] (std::size\_t position)  {  … // 여기도 다른 동작 추가  return text[position];  }  private:  std::string text;  }; |
| 코드의 중복되는 부분이 많아진다 |

경계 검사 등의 이런 저런 코드들을 별도의 멤버 함수로 옮기고 이것들을 operator[]의 양 버전에서 호출하게 만들면 이러한 문제에 대응할 수 있겠지만 여전히 코드 중복 문제는 남는다.  
- 함수 호출 횟수 증가 + return도 중복 코드임

operator[]의 2개의 버전 중 하나만 제대로 구현하고 다른 것은 호출하는 식으로만 만들 수 있다  
- const 껍데기 캐스팅으로 날리면 된다  
- 캐스팅이 좋은 개념은 아니라고 저자는 생각한다고 한다

operator[]의 상수 버전과 비상수 버전은 하는 일이 동일하다  
- 다른 점은 반환 타입에 const의 존재 여부  
- 따라서 캐스팅을 사용하여 반환 타입으로 const 껍데기를 없애도 안전하다  
- 비상수 operator[] 함수를 호출하는 쪽이라면 호출부에는 비상수 객체가 우선적으로 들어있을 것이 분명하기 때문

결론적으로 캐스팅이 필요하지만 안전성도 유지하면서 코드 중복을 피하는 방법은 비상수 operator[]가 상수 버전을 호출하도록 구현하는 것

|  |
| --- |
| class TextBlock{  public:  …  const char& operator[] (std::size-t position) const  {  …  return text[position];  }  char& operator[] (std::size\_t position)  {  return const\_cast<char&>(static\_cast<const TextBlock&> (\*this)[position]);  }  }; |

- operator[]의 반환 타입에 캐스팅을 적용하고 const를 떼어낸다  
- \*this 타입에 const를 붙여 operator[]의 상수 버전을 호출한다.

캐스팅이 1회가 아닌 2회 일어난다  
- 무한 재귀호출을 피하기 위한 것 -> 비상수 operator[]에서 비상수 operator[]를 호출하면 무한히 반복되는 꼴이다.  
- 상수 operator[]를 호출하겠다는 것을 명시적으로 표현해야 하짐나 직접적인 방법이 없다  
- 그래서 (\*this) 타입으로 캐스팅하는 것  
- 원래의 타입인 TextBlock&에서 const TextBlock&으로 바꾸는 것

코드의 미적인 요소는 줄어들지만 코드의 중복을 피하는 효과는 얻을 수 있다  
- 프로그래머의 취향에 따라 코드의 중복을 피할지 중복을 할지는 달라진다

**중요한 것  
- 비상수 멤버 함수의 구현에 상수 멤버 쌍둥이를 사용하는 기법은 반드시 알아둬야 한다**

코드 중복 회피를 위해 상수 버전이 비상수 버전을 호출하게 만드는 것은 문제가 존재한다  
- 상수 멤버 함수는 해당 객체의 논리적인 상태를 변경하지 않겠다는 컴파일러와의 약속  
- 비상수 멤버 함수는 이러한 약속 따위 없다  
- 상수 멤버에서 비상수 멤버를 호출하게 되면 수정하지 않겠다는 약속을 어긴 것이고 객체는 변경될 위험에 빠진다

상수 멤버 함수에서 비상수 멤버 함수를 호출하면 안 되는 이유

실제로 상수 멤버 함수에서 비상수 멤버를 호출하는 코드를 어떻게든 컴파일하려면 const\_cast를 적용해서 \*this의 const를 떼어내는 과정은 안전하지 못하다  
- 이 과정의 역순(위에 적은 것)은 안전성에 문제가 없다  
- 비상수 멤버 함수에서는 객체를 바꾸든 말든 상관 없다  
- 비상수 멤버 함수에서 상수 멤버 함수를 호출한다고 해서 특별한 문제가 발생하지도 않는다  
-> \*this에서 static\_cast가 동작하는 것만 봐도 그렇다

**정리**

|  |
| --- |
| const를 붙여 선언하면 컴파일러가 사용상의 에러를 잡는 것을 도와준다  - const는 어떠한 유효범위에 있는 객체도 붙을 수 있고 함수 매개변수 및 반환 타입에도 붙을 수 있으며 멤버 함수에도 붙을 수 있다. |
| 컴파일러 쪽에서 보면 비트 수준의 상수성을 지켜야 하지만 프로그래머 쪽에서는 논리적인 상수성을 사용하여 프로그래밍을 해야 한다 |
| 상수 멤버 및 비상수 멤버 함수가 기능적으로 동일하게 구현되어 있을 경우 코드 중복을 피하는 것이 좋다.  - 이럴 때 비상수 버전이 상수 버전을 호출하도록 만들자 |

**4: 객체를 사용하기 전에 반드시 그 객체를 초기화하자**