Effective C++

# Chapter.1: C++의 규칙

## 1: C++은 단일 언어가 아닌 언어들의 연합체로 봐야 한다.

## 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

객체의 초기화에 있어서 C++에는 여러 방법이 있는데 저자는 이게 마음에 들지 않는다고 한다  
- 나도 그렇다

|  |
| --- |
| int x; |
| 이 경우에는 x의 값이 0으로 확실하게 초기화 된다 |
| class Point{  int x, y;  };  ..  Point p; |
| 이 경우에는 상황에 따라 초기화가 보장되기도 하고 되지 않기도 한다 |

**초기화 되지 않은 값을 읽게 되면 정의되지 않은 동작이 그대로 흘러나오게 된다.  
-** 어떠한 플랫폼의 경우 초기화되지 않은 객체를 읽기만 해도 프로그램이 멈춘다  
- 일반적인 경우 무작위의 비트 값을 읽고 객체의 내부가 이상한 값을 갖게 된다

= 결국 프로그램이 정상적인 동작을 하지 못하고 디버깅의 시간을 가져야 한다

C++의 객체 초기화가 중구난방인 것은 아님  
- 언제 초기화가 보장되고 언제 그렇지 않은지에 대한 규칙이 명확하게 준비되어 있다  
- 규칙 자체는 조금 복잡한 편이다  
- 조건을 모두 기억할 필요하는 없다

**가장 좋은 방법은 모든 객체를 사용하기 전에 항상 초기화하는 것**

|  |  |
| --- | --- |
| int x = 0; | int 타입의 직접 초기화 |
| const char \*text = “A C-Style string” | 포인터의 직접 초기화 |
| double d;  std::cin >> d; | 입력 스트림에서 읽어 초기화를 수행하는 경우 |
| **기본 제공 타입으로 만들어진 비 멤버 객체는 초기화를 직접 해야 한다** | |

**기본 제공 타입을 제외한 C++의 초기화의 나머지 부분은 생성자로 귀결된다.  
-** 지켜야 할 규칙은 간단하다  
- **그 객체의 모든 내용을 초기화하면 된다**.  
- 이것만 지키면 모든 것이 해결된다.

\* 대입(Assignment)과 초기화(Initialization)를 헷갈리지 않는 것이 중요하다

|  |
| --- |
| class PhoneNumber { .. };  class ABEntry {  public:  ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones);  private:  std::string theName;  std::string theAddress;  std:list<PhoneNumber> thePhones;  int numTimesConsulted;  }; |
| ABEntry::ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones)  {  theName = name;  theAddress = address;  thePhones = phones;  numTimesConsulted = 0;  }; |
| 위의 과정은 초기화가 아닌 **대입**이다. |

- ABEntry의 생성자에서 theName, theAddress, thePhones는 초기화가 아닌 어떠한 값이 대입이 되고 있다.  
- 3개의 데이터 멤버의 기본 생성자가 호출된 것으로 초기화는 이미 지나감  
- 기본 제공 타입인 numTimesConsulted도 대입되기 전 초기화되었다고 보장할 수 없다

**해결 방법: 멤버 초기화 리스트**

|  |
| --- |
| ABEntry::ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones): theName(name), theAddress(address), thePhones(phones), numTimesConsulted(0) {} |
| 생성자 본문에 아무것도 없지만 멤버 초기화 리스트를 통해 대입이 아닌 초기화가 가능해진다 |

- 일반적인 대입 방법보다 멤버 초기화 리스트가 더 효율적일 수 있다.  
- 대입만 사용한 버전의 경우 string 타입의 기본 생성자를 호출하고 초기화를 미리 하고 생성자에서 대입을 하는 방식이기 때문으로 초기화는 헛 짓이 되버린다.  
- 이 문제를 멤버 초기화 리스트를 통해 피해갈 수 있는 것

초기화 리스트에 들어가는 인자는 바로 데이터 멤버에 대한 생성자의 인자로 쓰인다  
- theName은 name으로부터 복사 생성자에 의해 초기화 되는 식

대부분의 데이터 타입에 대해서는 기본 생성자 호출 후 복사 대입 연산자를 연달아 호출하는 이전의 방법보다 **복사 생성자를 한 번 호출하는 방법(초기화 리스트 사용)이 훨씬 효율적**이다.

대부분의 타입에 포함되지 않는 타입이 기본 제공 타입으로 기본 제공 타입은 초기화와 대입에 걸리는 비용의 차이가 없다  
- 그래도 그냥 멤버 초기화 리스트에 모두 넣어서 사용한다

데이터 멤버를 기본 생성자로 초기화하고 싶어도 멤버 초기화 리스트를 사용하도록 하자  
- 생성자 인자로 아무것도 주지 않아도 된다

|  |
| --- |
| ABEntry::ABEntry() : theName(), theAddress(), thePhones(), numTimesConsulted(0) {} |
| string 타입에 대해서는 기본 생성자를 호출한다  기본 제공 타입은 명시적으로 초기화해준다. |

기본 생성자이든 아니든 클래스 데이터 멤버는 모두 초기화 리스트에 항상 올려 두는 것이 리스트에서 어떤 멤버를 빼먹었을 때 어떤 멤버가 초기화되지 않을 수 있다는 사실을 끌고가야 하는 부담이 없어진다.  
- 기본 제공 타입도 초기화 리스트에 넣어서 사용하는 것을 추천한다고 함

**기본 제공 타입의 멤버를 초기화 리스트에 넣어야 하는 일이 선택이 아니라 의무가 될 수도 있다**.  
- 해당 값이 상수이거나 참조자로 되어 있는 데이터 멤버의 경우 반드시 초기화가 선행되어야 한다  
- **상수와 참조자는 대입 자체가 불가능하다**

이런 것을 일일이 외우는 것보다 그냥 통짜로 멤버 초기화 리스트를 사용하는 것을 추천하는 것

여러 개의 생성자를 가지고 있는 클래스의 경우 각 생성자마다 초기화 리스트가 붙는다  
- 이러한 클래스에 데이터 멤버와 기본 클래스가 적지 않다면 초기화 리스트가 생성자마다 늘어지고 코드가 더러워진다.  
- 이러한 경우 대입으로도 초기화가 가능한 데이터 멤버들을 초기화 리스트에서 빼내고 별도의 함수로 옮기는 것도 가능하다  
- 이들에 대한 대입 연산을 하나의 함수에 뭉치고 모든 생성자에서 이 함수를 호출하게 하는 방법(Getter Setter 식인 것 같음)

이러한 방법은 데이터 멤버의 진짜 초기값을 파일에서 불러오거나 데이터 베이스에서 찾아오는 경우에 특히 유용하다

**요약  
- 일반적인 경우라면 대입을 통한 가짜 초기화 방식보다는 진짜 멤버 초기화(초기화 리스트)를 사용하는 것이 좋다.**

**객체를 구성하는 데이터의 초기화 순서**

\*이 순서는 어떠한 컴파일러든 항상 동일하다

|  |
| --- |
| 기본 클래스는 파생 클래스보다 먼저 초기화된다 |
| 클래스 데이터 멤버는 그들이 선언된 순서대로 초기화된다 |
| ex) ABEntry 클래스에서 theName이 항상 첫번째로, theAddress가 두번째로 초기화 되는 것 |
| 멤버 초기화 리스트에 넣어진 순서가 달라도 초기화 순서는 선언 순으로 컴파일된다. |

- 안타깝게도 초기화 리스트의 순서가 섞여도 컴파일 된다  
- 혼란을 막고 찾아내기 힘든 동작 버그도 피하기 위해서 **초기화 리스트의 멤버 순서도 클래스 선언 순서와 동일하게 맞춰야 한다**.

**비정지역 정적 객체의 초기화 순서는 개별 번역 단위에서 정해진다.**

**정적 객체**는 자신이 생성된 시점부터 프로그램이 끝날 때까지 살아 있는 객체  
- 스택 객체 및 힙 기반 객체는 애초부터 정적 객체가 될 수 없다

정적 객체의 종류

|  |
| --- |
| 전역 객체 |
| 네임 스페이스 유효 범위에서 정의된 객체 |
| 클래스 안에서 static으로 선언된 객체 |
| 파일 유효 범위에서 static으로 선언된 객체 |
| 함수 안에서 static으로 선언된 객체 |

- 이 중에서 함수 안에 있는 정적 객체를 **지역 정적 객체**라고 한다  
- 나머지는 **비지역 정적 객체**라고 한다.

정적 객체는 프로그램이 종료될 때 자동으로 소멸된다  
- main 함수가 끝나면 정적 객체의 소멸자가 호출된다.

**번역 단위**는 컴파일을 통해 하나의 목적 파일을 만드는 바탕이 되는 소스 코드  
- 번역: 소스의 언어를 기계어로 옮기는 것  
- 번역 단위는 기본적으로 소스 파일 하나가되는데 그 파일이 #include를 하는 파일들까지 합쳐서 하나의 번역 단위가 된다.

**별개의 번역 단위에서 정의된 정적 객체들의 초기화 순서는 정해져 있지 않다**

|  |
| --- |
| claass FileSystem{  public:  ...  std::size\_t numDisks() const;  ...  }; |
| extern FileSystem tfs; // 사용자가 쓰게 될 객체 |

|  |
| --- |
| class Directory{  public:  Directory( params );  ...  }; |
| Directory::Directory( params )  {  ...  std::size\_t disks = tfs.numDisks();  ...  } |

사용자가 Directory 클래스를 사용해서 임시 파일을 담는 디렉토리 객체를 하나 생성한다고 가정한다

|  |
| --- |
| Directory tempDir { params }; |

정적 객체의 초기화 순서 때문에 문제가 심각해질 수 있다.  
- tfs가 tempDir보다 선행되어 초기화되지 않으면 tempDir의 생성자는 tfs가 초기화되지 않았는데 tfs를 사용하게 된다.  
- tfs와 tempDir은 제작자, 제작 시기도 다르고 소스 파일도 다르다.

**둘은 다른 번역 단위 안에서 정의된 비지역 정적 객체에 해당**한다

그리고, **별개의 번역 단위에서 정의된 정적 객체들의 초기화 순서는 정해져 있지 않다**

일반적인 방법으로 이 문제를 해결할 수는 없고 설계 측면에서 약간의 변화가 필요하다.

해결 방법

비지역 정적 객체로 선언하고, 그 함수에서는 이들에 대한 참조자를 반환하게 만든다  
- 사용자 쪽에서는 비지역 정적 객체를 직접 참조하는 형식이 아닌 함수 호출로 대신한다.  
- 즉, **비지역 정적 객체를 지역 정적 객체로 바꾸는 것**이다.  
- **디자인 패턴에서 싱글톤 패턴(Singleton Pattern)의 전형적인 구현 방식**이다.

지역 정적 객체는 함수 호출 중에 그 객체의 정의에 최초로 닿았을 때 초기화되도록 만들어져 있다.  
- C++이 보장한다  
- 위의 방법이 이 사실을 이용한 방법  
- 비지역 정적 객체를 직접 접근하지 않고 지역 정적 객체에 대한 참조자를 반환한 쪽으로 바꾼 경우, 얻어낸 참조자는 반드시 초기화된 객체를 참조하도록 맞춰야 한다

비지역 정적 객체 대신에 준비한 지역 객체 참조자 반환 함수를 호출할 일이 없다면 해당 객체의 생성/소멸 비용도 생기지 않게 막아야 한다.  
- 진짜 비지역 정적 객체의 경우 막고 말고 할 것도 없긴 하다

|  |  |
| --- | --- |
| class FileSystem { .. }; |  |
| FirleSystem& tfs()  {  static FileSystem fs;  reutnr fs;  } | tfs 객체를 이 함수로 대신한다  - 지역 정적 객체를 정의하고 초기화하고 이 객체에 대한 참조자를 반환한다. |
| class Directory { ... }; |  |
| Directory::Directory (params)  {  std::size\_t disks = tfs().numDisks();  } |  |
| Directory& tempDir()  {  static Directory td;  return td;  } | tempDir 객체를 이 함수로 대신한다  - 지역 정적 객체를 정의하고 초기화하고 이 객체에 대한 참조자를 반환한다. |

- tfs와 tempDir 대신에 tfs(), tempDir()을 참조하는 것으로 바뀌기만 한 것  
- 정적 객체 자체를 직접 사용하지 않고 그 객체에 대한 참조자를 반환하는 함수를 사용하는 것

참조자 반환 함수는 어느 경우든 복잡하게 구현될 일이 없다  
- 처음에 지역 정적 객체 정의 및 초기화  
- 그 객체의 참조자를 반환하면 된다.  
- 함수 본문이 단순하여 인라인으로 만들어도 좋을 수 있지만 다중 스레드 시스템에서는 동작에 문제가 생길 수 있다.  
- 다중 스레드 시스템 상에서는 비상수 정적 객체(지역, 비지역 모두)는 큰 문제가 된다

다중 스레드 시스템 상에서 비상수 정적 객체의 문제를 해결 방법 중 하나로 프로그램이 다중 스레드로 돌입하기 전 단계에서 참조자 반환 함수를 전부 손으로 호출한다  
- 초기화에 관계된 경쟁 상태를 해결해줄 수 있다.

초기화 순서 문제를 방지하기 위해 참조자 반환 함수를 사용하는 것은 객체들의 초기화 순서를 제대로 맞춰 둔다는 전제 조건이 있다.  
- 객체 B가 초기화 되기 전에 객체 A가 초기화되어야 하는데 A의 초기화가 B의 초기화에 의존하도록 되어 있다면 문제가 발생한다  
- 이러한 문제점을 미리 고려한다면 최소한 단일 스레드 상에서는 확실하다.

어떤 객체가 초기화되기 전에 그 객체를 사용하는 일이 생기지 않도록 하기 위해서는

|  |
| --- |
| 멤버가 아닌 기본 제공 타입 객체는 직접 초기화한다 |
| 객체의 모든 부분에 대한 초기화에는 멤버 초기화 리스트를 사용한다 |
| 별개의 번역 단위에 정의된 비지역 정적 객체에 대한 영향을 끼치는 불확실한 초기화 순서를 고려하여 불확실성을 피해서 프로그램을 설계한다. |

정리

|  |
| --- |
| 기본 제공 타입의 객체는 직접 손으로 초기화한다  - 경우에 따라 혼자 되기도 하고 안되기도 하니까 직접 한다 |
| 생성자에서는 데이터 멤버에 대한 대입문을 생성자 본문 내부에 넣는 방법으로 멤버를 초기화하지 말고 멤버 초기화 리스트를 사용한다  - 초기화 리스트에서 데이터 멤버를 나열할 때는 클래스에서 각 데이터 멤버가 선언된 순서와 똑같이 나열한다 |
| 여러 번역 단위에 있는 비지역 정적 객체들의 초기화 순서 문제는 피해서 설계한다 - 비지역 정적 객체를 지역 정적 객체로 바꾼다. |

# Chapter.2: 생성자, 소멸자, 대입 연산자

대부분의 C++ 클래스에 한 개 이상 꼭 들어있는 것이 생성자와 소멸자, 대입 연산자  
- 생성자는 새로운 객체를 메모리에 만드는 데 필요한 과정을 제어하고 객체의 초기화를 맡는 함수  
- 소멸자는 객체를 제거하고 동시에 그 객체가 메모리 상에서 적절히 사라지는 과정을 제어하는 함수  
- 대입 연산자는 기존의 객체에 다른 객체의 값을 줄 때 사용하는 함수

클래스를 제대로 사용하기 위해서는 이것들을 잘 만들어야 한다

## 5: C++가 호출해버리는 함수들에 주의한다.

클래스가 비어 있지만 비어 있는 것이 아닌 때가 있다  
- C++가 빈 클래스를 훑고 지나가는 경우  
- C++의 어떠한 멤버 함수는 클래스에 안에 직접 선언해 넣지 않으면 컴파일러가 자동으로 선언하도록 되어 있다  
- 복사 생성자, 복사 대입 연산자, 소멸자가 이에 해당한다

컴파일러가 자동으로 선언하는 함수의 형태는 모두 기본형이다  
- 생성자도 선언되어 있지 않다면 역시 컴파일러가 자동으로 기본 생성자를 만들어 놓는다.  
- 모두 public 멤버에 inline 함수이다.

|  |  |
| --- | --- |
| class Empty {}; | class Empty{  public:  Empty() { ... }  Empty(const Empty& rhs) { ... }  ~Empty() { ... }  Empty& operator=(const Empty& rhs) {..}  }; |
| 두 클래스는 별 차이가 없다 | |

컴파일러가 자동으로 생성하는 함수들은 반드시 필요하다고 판단할 때만 만들어지도록 되어 있다.  
- 필요한 조건이 그렇게 대단한 것은 아니다

|  |  |
| --- | --- |
| Empty e1; | 기본 생성자와 소멸자 |
| Empty e2(e1); | 복사 생성자 |
| e2 = e1; | 복사 대입 연산자 |

**기본 생성자와 소멸자**

기본적으로 컴파일러에게 ‘배후의 코드’를 작성할 수 있는 자리를 마련하는 것이 그 역할이다  
- 기본 클래스 및 비정적 데이터 멤버의 생성자와 소멸자를 호출하는 코드가 여기서 생성되는 것이다.  
- 소멸자는 이 클래스가 상속한 기본 클래스의 소멸자가 가상 소멸자로 되어 있지 않다면 비가상 소멸자로 만들어진다

**복사 생성자와 복사 대입 연산자**

원본 객체의 비정적 데이터를 사본 객체 쪽으로 그냥 복사하는 것이 하는 일의 전부다

|  |
| --- |
| template<typename T>  class NamedObject {  public:  NamedObject(const char \*name, const T& value);  NamedObject(const std::string& name, const T& value);  private:  std::string nameValue;  T objectValue;  }; |

- 이 템플릿 안에는 생성자가 선언되어 있다 = 컴파일러가 기본 생성자를 만들지 않는다  
- 생성자 인자가 꼭 필요한 클래스를 만드는 것으로 결정하고 만들었다면 컴파일러는 기본 생성자를 만들지 않는다  
- 복사 생성자와 복사 대입 연산자는 선언되어 있지 않으므로 컴파일러가 기본형을 만든다.

|  |
| --- |
| NamedObject<int> no1(“Smallest Prime Number”, 2);  NamedObject<int> no2(no1); // 복사 생성자가 호출된다. |

- 컴파일러가 자체적으로 만든 복사 생성자는 no1.nameValue와 no1.objectValue를 사용하여 no2.nameValue와 no2.objectValue를 각각 초기화한다  
- string 타입은 자체적으로 복사 생성자를 가지고 있어 no2.nameValue의 초기화는 string의 복사 생성자에서 no1.nameValue를 인자로 받아 호출하여 이뤄진다.  
- int 타입은 기본 제공 타입으로 no2.objectValue의 초기화는 no1.objectValue의 각 비트를 그대로 복사해서 가져오는 것으로 끝난다.

컴파일러가 생성하는 NamedObject<int>의 복사 대입 연산자도 동작 원리가 근본적으로 같다.  
- 일반적인 것들만 본다면 이 복사 대입 연산자의 동작이 제대로 되기 위해서는 최종 결과 코드가 적법해야 하고 합리적이어야 한다.  
- 어느 검사도 통과하지 못하면 컴파일러는 자동 생성을 거부한다.

|  |
| --- |
| template<class T>  class NamedObject {  public:  NamedObject(std::string& name, const T& value);  private:  std::string& nameValue;  const T objectValue;  }; |
| std::string newDog(“Persephone”);  std::string oldDog(“Satch”);  NamedObject<int> p(newDog, 2);  NamedObject<int> s(oldDog, 36);  p = s; |

- 대입 연산이 일어나기 전, p.nameValue와 s.nameValue는 string 객체를 참조한다  
- 두 객체는 동일한 string 객체가 아니다.  
- 이 때 대입 연산이 발생하면 p.nameValue가 s.nameValue가 참조하는 string을 가리키는, 참조자 자체가 변경되지는 않는다.  
- **C++의 참조자는 원래 자신이 참조하고 있는 것과 다른 객체를 참조할 수 없다**.

위의 코드에 대해서 C++은 컴파일 에러를 던진다

참조자를 데이터 멤버로 갖는 클래스에 대입 연산을 지원하기 위해서는 직접 복사 대입 연산자를 정의해야 한다  
- 데이터 멤버가 상수 객체인 경우에도 C++이 위와 같은 동작을 뱉는다  
-> 상수 멤버를 수정하는 것은 문법에 어긋나는 행동으로 컴파일러가 자동으로 만든 복사 대입 연산자 내부에서 어떻게 동작할지 애매해 진다.

**정리**

|  |
| --- |
| 컴파일러는 경우에 따라 클래스에 대해 기본 생성자, 복사 생성자, 복사 대입 연산자, 소멸자를 암시적으로 만든다. |

## 6: 컴파일러가 만든 함수가 필요 없으면 이것들의 사용을 막는다

어떠한 경우에 클래스의 특정한 종류의 기능을 지원하지 않았으면 하는 의도를 반영하는 방법은 그런 기능을 제공하는 함수를 선언하지 않는 것이다  
- 이 방법은 복사 생성자와 복사 대입 연산자에 대해서는 동작하지 않는다  
- 복사 생성자와 복사 대입 연산자는 선언하지 않고 외부에서 호출할 경우 컴파일러가 이들을 선언한다(2-5에서 했던 것)

컴파일러가 자동으로 만드는 함수는 모두 공개된다. 즉 public 멤버가 되는 것  
- 복사 생성자와 복사 대입 연산자가 저절로 만들어지는 것을 막기 위해 직접 선언해야 하는 것은 동일함  
- 이것들을 public 멤버가 아닌 private 멤버로 선언하면 된다.  
- 이 함수들이 비공개의 접근성을 가지므로 외부로부터의 호출을 차단할 수 있다.

private 멤버는 여전히 friend함수가 호출할 수 있다는 문제점이 존재한다  
- 이것까지 막기 위해서는 define 자체를 해버리지 않는 방법도 있다  
- 정의되지 않은 함수가 어쩌다가 실수로 호출하려 한 경우 링크 시점에서 에러를 보게 될 것이니 상관없다  
- 실제로 이 방법은 하나의 기법으로 굳어지고 있다  
- C++의 iostream 라이브러리에 속한 몇 클래스에도 복사 방지책으로 사용되고 있다

|  |
| --- |
| class HomeForSale {  public:  ....  private:  ...  HomeForSalse(const HomeForSale&);  HomeForSale& operator=(const HomeForSale&);  }; |

- 매개변수의 이름을 굳이 적을 필요는 없다  
- 앞으로 구현될 일도 없으며 사용될 일도 없으므로 더더욱 매개변수의 이름을 적을 필요가 없다

사용자가 HomeForSale 객체의 복사를 시도하려고 하면 컴파일러가 막을 것이고 멤버 함수 또는 프렌드 함수 안에서 복사를 시도할 경우 링커가 막아줄 것이다.

**링크 시점의 에러를 컴파일 시점의 에러로 옮길 수도 있다**  
- 에러 탐지는 미리 하는 것이 좋으므로 이 방법이 추천된다.

복사 생성자와 복사 대입 연산자를 private로 선언하되 이것을 HomeForSale 자체에 넣지 말고 별도의 기본 클래스에 넣고 이것으로부터 HomeForSale을 파생시키는 방법으로 사용하는 것  
- 별도의 기본 클래스는 복사 방지만 막는 특별한 의미가 부여된다.

|  |
| --- |
| class Uncopyable {  protected:  Uncopyable() {}  ~Uncopyable() {}  private:  Uncopyable(const Uncompable&); // 복사 생성은 방지한다.  Uncopyable& operator=(const Uncopyable&); // 복사 대입 연산자도 방지한다  }; |
| 파생된 객체에 대해서 생성과 소멸을 허용한다  하지만 복사는 방지한다. |
| class HomeForSale : private Uncopyable {  ...  }; |
| 복사를 막고 싶은 클래스는 위의 클래스를 상속받아 사용하는 식 |

HomeForSale 객체의 복사를 외부에서 시도하려고 하면 HomeForSale 클래스 만의 복사 생성자와 복사 대입 연산자를 만들려고 한다  
- 컴파일러가 생성한 복사 함수는 기본 클래스의 대응 버전을 호출하게 되어 있다.  
- 이런 호출은 위처럼 코드를 작성할 시 호출하지 못하게 된다  
- 복사 함수들이 기본 클래스에서 공개되어 있지 않기 때문

Uncopyable의 상속은 public일 필요가 없으며, 소멸자는 가상 소멸자가 아니어도 된다.  
- Uncopyable 클래스는 데이터 멤버가 아예 없다.  
- 공백 기본 클래스 최적화 기법을 사용할 여지도 있지만 기본 클래스로 다중 상속의 여지도 존재한다.  
- 다중 상속을 할 경우 공백 기본 클래스 최적화 기법을 사용하지 못할 때가 종종 있다  
- 어지간한 경우 이런거 무시해도 상관없으며 부스트 라이브러리 안에 위의 Uncopyable과 동일한 클래스도 존재하여 이것을 사용해도 상관없다

정리

|  |
| --- |
| 컴파일러에서 자동으로 제공하는 기능을 허용하지 않으려면 대응되는 멤버 함수를 private로 선언하고 구현을 하지 않은 채로 남겨두거나 기본 클래스를 만들어 클래스를 파생시킨다. |

## 7: 다형성을 가진 기본 클래스에서는 소멸자를 반드시 가상 소멸자로 선언한다.

특정 기능을 가진 클래스를 기본 클래스로 만든 후 적절한 용도에 따라 파생시키면 유리한 클래스들이 존재한다

|  |
| --- |
| class TimeKeeper {  public:  TimeKeeper();  ~TimeKeeper();  }; |
| class AtomicClock: public TimeKeeper { ... }; |
| class WaterClock: public TimeKeeper { ... }; |
| class WristWatch: public TimeKeeper { ... }; |

- 클래스의 혜택을 받는 사용자들은 중간의 계산 과정에 상관없이 정보에 접근하려고 한다  
- 위의 예시에서 시간 기록 객체에 대한 포인터를 손에 넣는 용도로 팩토리 함수를 만들어 놓으면 편리하다  
- 팩토리 함수: 새로 생성된 파생 클래스 객체에 대한 기본 클래스 포인터를 반환하는 함수

|  |
| --- |
| TimeKeeper\* getTimeKeeper(); |
| TimeKeeper에서 파생된 클래스를 통해 동적으로 할당된 객체의 포인터를 반환함 |

- 팩토리 함수의 기존 규약을 따르면 getTimeKeeper 함수에서 반환되는 객체는 힙에 존재한다  
- 메모리 및 기타 자원의 누출을 막기 위해 적절히 삭제해야 한다

|  |
| --- |
| TimeKeeper\* ptk = getTimeKeeper();  ...  delete ptk; |
| 자원 누출을 막기 위해 삭제가 필요하다. |

- 객체 삭제를 사용자에게 전가하는 것은 에러 발생에 노출될 여지가 존재한다  
- 팩토리 함수의 인터페이스를 수정하면 발생할 수 있는 사용자 에러를 방지할 수 있기는 하다

getTimekeepr 함수가 반환하는 포인터가 파생 클래스 객체에 대한 포인터이고 이 포인터가 가리키는 객체가 삭제될 때는 기본 클래스 포인터를 통해 삭제된다는 점, 기본 클래스에 들어있는 소멸자가 비가상 소멸자라는 점에 주의해야 한다.  
- C++ 규정에 따르면 기본 클래스 포인터를 통해 파생 클래스 객체가 삭제될 때 **그 기본 클래스에 비가상 소멸자가 들어 있다면 프로그램 동작은 미정이다.**- 일반적으로 그 객체의 파생 클래스 부분이 소멸되지 않는다.

getTimekeeper 함수에서 포인터를 통해 얻는 AtomicClock 객체는 기본 클래스 포인터를 통해 삭제될 때 AtomicClock 부분이 제거되지 않고 AtomicClock의 소멸자도 제대로 실행되지 않는다  
- 기본 클래스 부분은 소멸 과정이 제대로 끝나므로 결국 **반쪽짜리 부분 소멸 객체가 된다**.  
- 이로 인해 자원, 메모리의 누수가 발생하며 자료 구조가 오염되고 무한한 디버깅 과정을 거치게 된다.

**해결 방법:** 기본 클래스에 가상 소멸자를 넣는다.

파생 클래스 객체를 기본 클래스 포인터로 삭제할 때 원하는 동작을 수행할 수 있게 만든다.  
- 기본 클래스의 소멸자 앞에 virtual 키워드를 붙이면 된다.

|  |
| --- |
| class TimeKeeper {  public:  Timekeeper();  virtual ~TimeKeepr();  ....  }; |
| TimeKeeper \*ptk = getTimeKeeper()’  ...  delete ptk; |

- Timekeeper와 비슷한 기본 클래스에는 소멸자 외에도 가상 멤버 함수들이 있기 마련이다.  
- 파생 클래스를 구현할 때 해당 함수를 역할에 따라 맞추는 작업을 허용한다는 뜻  
- **가상 함수를 하나라도 가진 클래스는 가상 소멸자를 가져야 하는 것이 대부분 맞다**

기본 클래스를 의도하지 않은 클래스에 대해 소멸자를 가상으로 선언하는 것은 좋지 않다

|  |
| --- |
| class Point {  public:  Point(int xCoord, int yCoord);  ~Point();  private:  int x, y;  }; |

|  |
| --- |
| 가상 함수에 대해서 |
| 가상 함수를 C++에서 구현하려면 클래스에 별도의 자료구조가 하나 들어가야 한다. - 이 자료구조는 프로그램 실행 중에 주어진 객체에 대해 어떤 가상 함수를 호출해야 하는지를 결정하는데 쓰이는 정보 - 실제로는 포인터의 형태를 취하는 것이 대부분 - vptr(가상 함수 테이블 포인터)라는 이름으로 불린다. - vptr은 가상 함수의 주소인 포인터들의 배열을 가리키고 있으며 가상 함수 테이블 포인터의 배열은 vtbl(가상 함수 테이블)이라고 불린다. - 가상 함수를 하나라도 가지고 있는 클래스는 반드시 그와 관련된 vtbl을 갖고 있다. - 어떤 객체에 대해 어떤 가상 함수가 호출되려고 하면 호출되는 실제 함수는 그 객체의 vptr이 가리키는 vtbl에 따라 결정된다. - vtbl에 있는 함수 포인터들 중 적절한 것이 연결되는 것 |

어느 경우를 막론하고 소멸자를 전부 virtual로 선언하는 것은 절대로 virtual로 선언하지 않는 것만큼 안좋은 일이다.  
- **가상 소멸자를 선언하는 것은 그 클래스에 가상 함수가 하나라도 들어 있는 경우에만 한정하면 된다.**

가상 함수가 하나도 없어도 비가상 소멸자로 인한 문제가 발생하는 경우도 있다.  
- 대표적인 예가 표준 string 타입으로 가상 함수를 가지고 있지 않지만 기본 클래스 타입으로 잡아버리는 경우가 존재한다.

|  |
| --- |
| class SpecialString: public std::string {  ...  }; |
| std::string은 가상 소멸자가 없다. |

- 이 경우에는 프로그램 어딘가에서 SpecialString의 포인터를 string의 포인터로 어떻게든 변환하고 그 string포인터에 delete를 적용하게 된다.  
- 이때부터 미정의 동작이 일어나게 된다.

|  |
| --- |
| SpecialString \*pss = new SpecialString(“Impeding Doom”);  std::string \*ps;  ...  ps = pss; // SpecialString\* 이 std::string\*으로 어떻게든 변환된다.  ...  delete ps; |
| delete ps에서 정의되지 않은 동작이 발생한다 - 실질적으로는 \*ps의 SpecialString 부분에 있는 자원이 누수된다. - SpecialString의 소멸자가 호출되지 않기 때문 |

위와 같은 문제는 가상 소멸자가 없는 클래스라면 어떤 것에든 전부 적용된다.  
- 가상 소멸자가 없는 클래스는 은근히 있다(ex: STL 컨테이너 타입 전부)

경우에 따라 순수 가상 소멸자를 사용하면 편리하게 사용할 수 도 있다.  
- 순수 가상 함수는 해당 클래스를 추상 클래스로 만든다  
- 추상 클래스: 그 자체로는 인스턴스를 못만드는 클래스

하지만 어떤 클래스가 추상 클래스였으면 좋겠지만 마땅히 넣을 만한 순수 가상 함수가 없을 때도 종종 있다.

추상 클래스는 본래 기본 클래스로 쓰일 목적으로 만들어진 것이고 기본 클래스로 쓰이려는 클래스는 가상 소멸자를 가져야 한다  
- 순수 가상 함수가 있으면 바로 추상 클래스가 된다

**결론  
- 추상 클래스로 만들고 싶은 클래스에 순수 가상 소멸자를 선언한다**

|  |
| --- |
| class AWOV {  public:  virtual ~AWOV() = 0;  }; |
| 순수 가상 소멸자를 선언한다. |

AWOV 클래스는 순수 가상 함수를 갖고 있으므로 추상 클래스다  
- 동시에 순수 가상 함수가 가상 소멸자로 소멸자 호출 문제로 고민할 필요 없다  
- 이 순수 가상 소멸자의 정의를 반드시 두어야 한다

|  |
| --- |
| AWOV::~AWOV() {} |

- 소멸자는 생성의 역순으로 실행된다  
- 상속 계통 구조에서 가장 아래 있는 파생 클래스의 소멸자가 가장 먼저 호출  
- 기본 클래스 쪽으로 거쳐 올라가며 각 기본 클래스의 소멸자가 하나씩 호출된다.  
- 컴파일러는 ~AWOV의 호출 코드를 만들기 위해 파생 클래스의 소멸자를 사용할 것이므로 이 함수의 본문이 필요한 것  
- 위의 순수 가상 소멸자의 정의가 없다면 링커 에러가 뜬다.

기본 클래스의 손에 가상 소멸자를 주는 규칙은 다형성을 가진 기본 클래스, 기본 클래스 인터페이스를 통해 파생 클래스 타입의 조작을 허용하도록 설계된 기본 클래스에만 적용된다.  
- TimeKeeper 클래스가 바로 이 경우  
- AtomicClock 등의 객체를 보면 TimeKeeper 포인터만 가지고 이것들을 조작할 수 있을 것이라 생각하게 되기 때문

모든 기본 클래스가 다형성을 갖도록 설계된 것은 아니다.  
- 표준 string 타입, STL 컨테이너 타입은 기본 클래스도, 다형성의 흔적도 찾을 수 없다

기본 클래스로는 쓰일 수 있지만 다형성을 갖지 않도록 설계된 클래스도 있다.  
- 이러한 클래스는 기본 클래스의 인터페이스를 통한 파생 클래스의 객체의 조작이 허용되지 않는다.

**정리**

|  |
| --- |
| 다형성을 가진 기본 클래스에는 반드시 가상 소멸자를 선언한다  - 어떤 클래스가 가상 함수를 하나라도 가진다면 이 클래스의 소멸자도 가상 소멸자여야 한다. |
| 기본 클래스로 설계되지 않았거나 다형성을 갖도록 설계되지 않은 클래스에는 가상 소멸자를 선언하지 말아야 한다. |

## 8: 예외가 소멸자를 떠나지 못하도록 묶어 놓는다.