

# CH.9 클래스와 객체 마스터하기

전문가를 위한 C++

오기환

# friend

- 내 클래스의 모든 멤버에게  
접근 가능하도록 허용하기

```
1  class Bar
2  {
3  public:
4      void process();
5  };
6
7
8  void dump();
9
10 class Foo
11 {
12     friend class Bar;
13     friend void Bar::process();
14     friend void dump();
15 };
16
17
```

# 5의 규칙 (Rule of 5)

# 객체에 동적 메모리 할당하기

- 생성자에서 할당, 소멸자에서 해제
- 복제 생성자와 대입 연산자를 직접 정의

```
1  class Spreadsheet
2  {
3  public:
4      // 동적 할당 메모리에 저장된 데이터를 직접 복제
5      Spreadsheet(const Spreadsheet &src);
6
7      // 동적 할당 메모리에 저장된 데이터를 직접 대입
8      // 코드의 간결함과 안정성을 위해 복제 후 swap 사용
9      Spreadsheet &operator=(const Spreadsheet &rhs)
10     {
11         if (this == &rhs)
12         {
13             return *this;
14         }
15         Spreadsheet temp(rhs);
16         swap(*this, temp);
17         return *this;
18     }
19     friend void swap(Spreadsheet &first, Spreadsheet &second);
20 };
21
22 void swap(Spreadsheet &first, Spreadsheet &second)
23 {
24     // 모든 멤버 변수들 swap
25 }
```

# 대입/복제/값 전달 금지하기

- 가장 간단한 방법
  - 컴파일러가 기본적으로 생성하는 연산자를 명시적으로 delete 하면 됨



```
1 class SpreadCell
2 {
3 public:
4     SpreadCell(const SpreadCell &src) = delete;
5     SpreadCell &operator=(const SpreadCell &rhs) = delete;
6 };
```

# 이동 의미론과 이동 처리

## ravalue와 rvalue reference

- lvalue (좌측값) - 이름과 주소를 가진 대상 변수, 대입문의 좌측 변수
- rvalue (우측값) - 리터럴, 임시 객체, 값 등 좌측값이 아닌 모든 대상, 대입문의 우측 변수
- rvalue reference (우측 값 레퍼런스)
  - 우측 값이  $4 * 2$  와 같은 임시변수 일 경우 사용
  - `type&& name` 으로 우측값을 사용한다고 명시적으로 함수 선언 가능

# 좌측값 우측값 예시



```
1 // 좌측값 레퍼런스 매개변수
2 void greet(std::string &message)
3 {
4     std::cout << "left " << message << std::endl;
5 }
6
7 // 우측값 레퍼런스 매개변수
8 void greet(std::string &&message)
9 {
10     std::cout << "right " << message << std::endl;
11 }
12
13 std::string a = "hello";
14 std::string b = "world";
15 greet(a);           // 좌측값 전달
16 greet(a + b);       // 우측값 (임시변수) 전달
17 greet("hello me");  // 리터럴 (임시변수) 전달
```



```
1 // 우측값이 좌측값이 되는 경우
2 void helper(std::string &&message);
3
4 void greet2(std::string &&message)
5 {
6     // 에러, 함수내에서는 message 라는 이름으로 접근 가능한 주소가 있는 변수임
7     helper(message);
8
9     // std::move 사용으로 우측값으로 변환해주어야 함
10    helper(std::move(message));
11 }
12
13 int &e = 2; // 에러 , 2는 주소가 없는 임시 변수임
14 int &&i = 2;
15 int a = 2, b = 3;
16 int &&j = a + b;
```

# 이동 의미론 구현

- 이동 생성자와 이동 대입 연산자를 구현하고

```
1 class Movable
2 {
3 public:
4     Movable(Movable &&src) noexcept;
5     Movable &operator=(Movable &&rhs) noexcept;
6
7 private:
8     void cleanup() noexcept;
9     void moveFrom(Movable &src) noexcept;
10 }
```

```
1 void Movable::cleanup() noexcept
2 {
3     // 기존 메모리 해제 (생략)
4
5     // 동적 할당된 주소를 nullptr 로 변경해서 소멸자에서 중복 free 방지
6     memory = nullptr;
7 }
8
9 void Movable::moveFrom(Movable &src) noexcept
10 {
11     // 얽은 복제 하기 (생략), 객체는 std::move 사용
12     someString = std::move(src.someString);
13
14     // 동적 할당된 주소를 nullptr 로 변경해서 소멸자에서 중복 free 방지
15     memory = nullptr;
16 }
17
18 Movable &Movable::operator=(Movable &&rhs) noexcept
19 {
20     if (this == &rhs)
21     {
22         return *this;
23     }
24
25     cleanup();
26     moveFrom(rhs);
27     return *this;
28 }
```



# swap 으로 이동생성자 구현하기



```
1 // swap 으로 이동생성자 구현
2 class MovableWithSwap
3 {
4     private:
5         MovableWithSwap() = default;
6     public:
7         MovableWithSwap(MovableWithSwap&& src) noexcept;
8         MovableWithSwap& operator=(MovableWithSwap&& rhs) noexcept;
9 }
10
11 MovableWithSwap::MovableWithSwap(MovableWithSwap &&src) noexcept
12     :MovableWithSwap() {
13     swap(*this, src);
14 }
15
16 MovableWithSwap& MovableWithSwap::operator=(MovableWithSwap &&rhs) noexcept {
17     MovableWithSwap temp(std::move(rhs));
18     swap(*this, temp);
19     return *this;
20 }
```

# 이동 의미론이 적용된 swap



```
1 void nonMoveSwap(T &a, T &b)
2 {
3     T temp(a);
4     a = b;
5     b = temp;
6 }
```



```
1 void moveSwap(T &a, T &b)
2 {
3     T temp(std::move(a));
4     a = std::move(b);
5     b = std::move(temp);
6 }
```

# Rule of Zero (0의 규칙)

- 앞의 예제는 직접 동적 할당/해제를 사용하는 경우
- 직접 메모리 동적 할당/해제 하지 말고 `std::vector`와 같은 컨테이너를 적극 활용 하자

# 메서드의 종류

# static

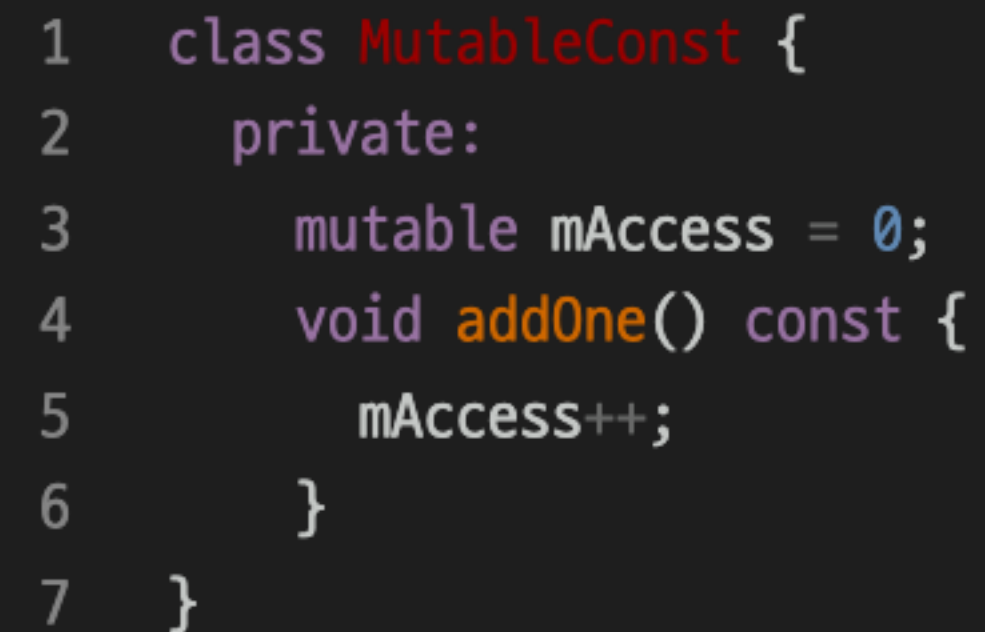
- 클래스 단위로 적용되는 메서드
- static 키워드를 앞에 붙여서 정의
  - 하나의 객체가 아닌 클래스 전체에 적용되기 때문에 this 키워드 사용 불가

# const

- 메서드 내부에서 절대로 데이터 멤버를 변경하지 않는 메서드
  - const 키워드를 **뒤**에 붙여서 선언
  - static 메서드는 애초에 멤버 접근 못하므로 const 선언 불필요
  - const 메서드 안에서는 const 메서드만 호출 가능

# const 메서드 에서 데이터 수정하기

- const 메서드 내에서도 수정가능하게 하기 위해 mutable 키워드로 선언하면 됨



```
1 class MutableConst {  
2     private:  
3         mutable mAccess = 0;  
4         void addOne() const {  
5             mAccess++;  
6         }  
7     }
```

# 메서드 오버로딩

- 이름은 같고 매개변수의 타입이나 개수만 다르게 지정 가능
- 리턴타입에 대한 오버로딩은 지원하지 않음
- const를 기준으로도 오버로딩이 됨 - const 객체는 const 함수만 호출하게 됨
  - 구현이 동일한 경우에는 const\_cast 사용

```
1  class ConstOverloading {
2      public:
3          const ConstOverloading& getIt() const {
4              return someConstThing;
5          };
6          ConstOverloading& getIt() {
7              // std::as_const로 const 형변환
8              // const_cast 로 const 제거
9              return const_cast<ConstOverloading&>(std::as_const(*this).getIt());
10         }
11     }
```



# 오버로딩 삭제하기



```
1  class NotForDouble
2  {
3  public:
4      void foo(int i);
5      void foot(double d) = delete;
6  }
```

# 인라인 메서드

- 함수/메서드 호출 부분에 구현을 직접 주입하는 형태
- inline 키워드를 선언부 앞에 붙여서 사용
- 필요없는 경우, 성능에 문제가 생길 수 있는 경우 컴파일러가 무시하고 일반 메서드로 컴파일 할 수도 있음
- 클래스 정의 부에서 바로 구현하면 자동으로 inline 처리

# 디폴트 인수

- 함수나 메서드의 매개변수에 대한 기본값을 설정
- 오른쪽 끝의 매개변수부터 시작해서 중간에 건너뛰지 않고 연속으로 나열해야 함
- 매개변수만 다른 메서드 오버로딩을 디폴트 인수를 사용하여 1개의 메서드로 구현 가능

# 데이터 멤버의 종류

# 생략

- 기본적으로 아는 거랑 똑같으므로 생략

기타

# 중첩 클래스



```
1 class Outer {  
2     public:  
3         class Inner {  
4             void do();  
5         }  
6     }  
7     void Outer::inner::do() {  
8         return;  
9     }
```



```
1 class SimpleOuter {  
2     public:  
3         class Inner;  
4     }  
5  
6     class SimpleOuter::Inner {  
7         public:  
8             void do();  
9     }
```

# 열거(enum) 사용



```
1  class EnumMember {  
2      public:  
3          enum class Color {Red = 1, Green, Blue, Yellow};  
4      private:  
5          Color mColor = Color.red;  
6  }
```



# 연산자 오버로딩

- 기본 오버로딩과 같음

안정적인 인터페이스 만들기

# 인터페이스 클래스와 구현 클래스

- c++ 언어적 특성으로 인해 private과 protected 도 노출됨
- 인터페이스 클래스와 구현 클래스를 따로 정의하면 이러한 단점을 줄일 수 있음
  - 펌플 이디엄 (pimpl idiom), 브릿지 패턴 (bridge pattern)
  - 정말로 필요한 public 객체만 선언, 나머지는 구현 클래스에 대한 레퍼런스만 선언