객체지향프로그래밍

Lecture 7 : 접근지정자,const객체, static멤버

충북대 소프트웨어학부

이 태 겸(showm321@gmail.com)

목차

접근지정자

❖ 멤버에 대한 3가지 접근 지정자

- private(디폴트) :
  - 동일한 클래스의 멤버 함수에만 제한함.
  - 클래스 밖에서 직접 접근 불가
- protected
  - 클래스 자신과 상속받은 자식 클래스에만 허용
- public
  - 모든 다른 클래스에 허용. 클래스 밖에서 직접 접근 가능

접근지정자	해당 클래스의 멤버함수에서의 접근	파생클래스의 멤버함수에서의 접근	클래스 밖에서의 접근
private	O	X	X
protected	O	O	X
public	O	O	O

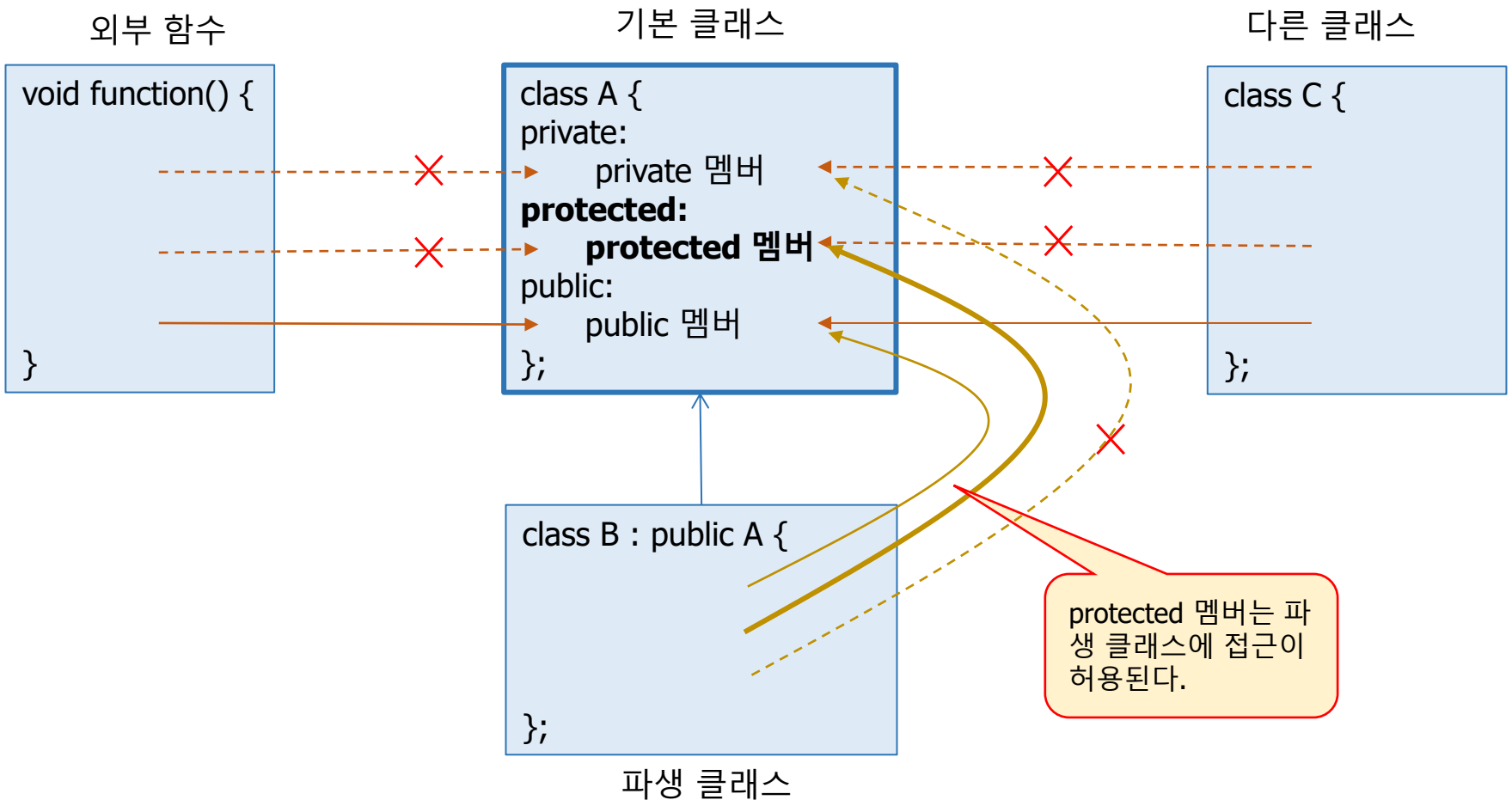
- 클래스 안 : 클래스의 내부와 멤버 함수
- 클래스 밖 : 전역 함수, 파생 혹은 다른 클래스, 클래스로 만들어진 객체

❖ 캡슐화의 목적

- 객체 보호, 보안
- C++에서 객체의 캡슐화 전략
  - 객체의 상태를 나타내는 데이터 멤버(멤버 변수)에 대한 보호
  - 중요한 멤버는 다른 클래스나 객체에서 접근할 수 없도록 보호
  - 외부와의 인터페이스를 위해서 일부 멤버는 외부에 접근 허용

```
class Sample {
private:
  // private 멤버 선언 (디폴트)
public:
   // public 멤버 선언
protected:
   // protected 멤버 선언
};
```

멤버의 접근 지정에 따른 접근성



중복 접근 지정과 디폴트 접근 지정

접근 지정의 중복 사용 가능

```
class Sample {
private:
   // private 멤버 선언
public:
   // public 멤버 선언
private:
   // private 멤버 선언
};
```

접근 지정의 중복 사례

```
class Sample {
private:
   int x, y;
public:
   Sample();
private:
   bool checkXY();
};
```

디폴트 접근 지정은 private

```
class Circle {
   int radius;
public:
   Circle();
   Circle(int r);
   double getArea();
};
```

```
class Circle {
private:
   int radius;
public:
   Circle();
   Circle(int r);
   double getArea();
};
```

멤버 변수는 private 지정이 바람직함

```
class Circle {
public:
   int radius;
   Circle();
   Circle(int r);
   double getArea();
};


Circle::Circle() {
   radius = 1;
}
Circle::Circle(int r) {
   radius = r;
}
```

멤버 변수 보호받지 못함

노출된 멤버는 마음대로 접근. 나쁜 사례

```
int main() {
   Circle waffle;
   waffle.radius = 5;
}
```

멤버 변수를 public으로 선언한 나쁜 사례

```
class Circle {
private:
   int radius;
public:
   Circle();
   Circle(int r);
   double getArea();
};

Circle::Circle() {
   radius = 1;
}
Circle::Circle(int r) {
   radius = r;
}
```

멤버 변수 보호받고 있음

```
int main() {
   Circle waffle(5); // 생성자에서 radius 설정
   waffle.radius = 5; // private 멤버 접근 불가
}
```

멤버 변수를 private으로 선언한 바람직한 사례

```
#include <iostream>
using namespace std;

class PrivateAccessError {
private:
   int a;
   void f();
   PrivateAccessError();
public:
   int b;
   void g();
   PrivateAccessError(int x);
};

PrivateAccessError::PrivateAccessError() {
   a = 1;          // (1)
   b = 1;          // (2)
}

PrivateAccessError::PrivateAccessError(int x) {
   a = x;          // (3)
   b = x;          // (4)
}

void PrivateAccessError::f() {
   a = 5;          // (5)
   b = 5;          // (6)
}
```

다음 코드에서 컴파일 오류가 발생하는 곳은?

```
void PrivateAccessError::g() {
   a = 6;          // (7)
   b = 6;          // (8)
}

int main() {
   PrivateAccessError objA;          // (9)
   PrivateAccessError objB(100);     // (10)
   objB.a = 10;                      // (11)
   objB.b = 20;                      // (12)
   objB.f();                         // (13)
   objB.g();                         // (14)
}
```

```
#include <iostream>
using namespace std;

class PrivateAccessError {
private:
   int a;
   void f();
   PrivateAccessError();
public:
   int b;
   void g();
   PrivateAccessError(int x);
};

PrivateAccessError::PrivateAccessError() {
   a = 1;          // (1)
   b = 1;          // (2)
}

PrivateAccessError::PrivateAccessError(int x) {
   a = x;          // (3)
   b = x;          // (4)
}

void PrivateAccessError::f() {
   a = 5;          // (5)
   b = 5;          // (6)
}
```

다음 코드에서 컴파일 오류가 발생하는 곳은?

```
void PrivateAccessError::g() {
   a = 6;          // (7)
   b = 6;          // (8)
}

int main() {
   PrivateAccessError objA;          // (9)
   PrivateAccessError objB(100);     // (10)
   objB.a = 10;                      // (11)
   objB.b = 20;                      // (12)
   objB.f();                         // (13)
   objB.g();                         // (14)
}
```

(9) 생성자 PrivateAccessError()는 private 이므로
main()에서 호출할 수 없다.
(11) a는 PrivateAccessError 클래스의 private 멤버이므로
main()에서 접근할 수 없다.
(13) f()는 PrivateAccessError 클래스의 private 멤버이므로
main()에서 호출할 수 없다.

- 생성자도 private으로 선언할 수 있음.
- 생성자를 private으로 선언하는 경우는 한 클래스에서 오직 하나의 객체만 생성할 수 있도록 하기 위한 것임.

C++ 구조체

❖C++ 구조체
- 상속, 멤버, 접근 지정 등 모든 것이 클래스와 동일
- 클래스와 유일하게 다른 점
  - 구조체의 디폴트 접근 지정 – public
  - 클래스의 디폴트 접근 지정 – private

```
struct StructName {
private:
   // private 멤버 선언
protected:
   // protected 멤버 선언
public:
   // public 멤버 선언
};
```

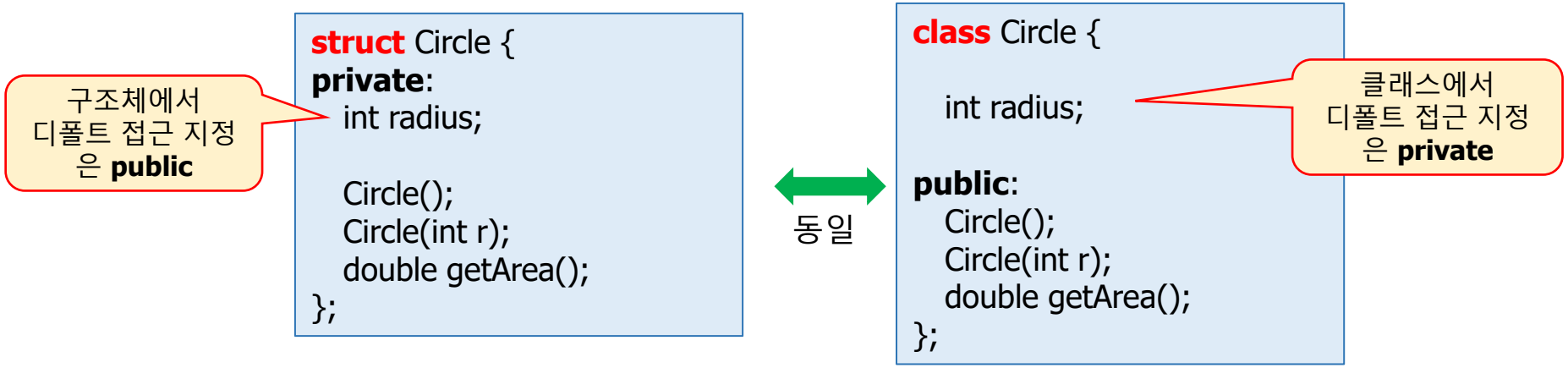
❖C++에서 구조체를 수용한 이유?
- C 언어와의 호환성 때문
  - C의 구조체 100% 호환 수용
  - C 소스를 그대로 가져다 쓰기 위해 (extern "C" { #include "your_c_header.h"})

```c
#include <stdio.h>

int main() {
    printf("Hello, C code in C++!\n");
    return 0;
}
```

❖구조체 객체 생성
- struct 키워드 생략

```
structName stObj;          // (0), C++ 구조체 객체 생성
~~struct~~ structName stObj; // (X), C 언어의 구조체 객체 생성
```

구조체와 클래스의 디폴트 접근 지정 비교



접근자 함수와 수정자 함수

❖ 접근자 함수
- public 접근 권한을 갖는 멤버 함수로
- 클래스의 멤버 변수(private)에 대한 접근을 도와주는 기능 제공
- 접근자 함수는 보통 **get** 함수와 **set** 함수로 구현됨
- getter(), setter()

```
class Circle {
private:
   int radius;

public:
   Circle();
   Circle(int r);
   double getArea();

// 접근자함수
   int getRadius();
   bool setRadius(int r);
};

int Circle::getRadius() {
   return radius;
}

bool Circle::setRadius(int r) {
   if (r > 30) return false;
   radius = r;
   return true;
}
```

```
int main() {
   Circle waffle(5); // 생성자에서 radius 설정
   waffle.radius = 5; // private 멤버 접근 불가
    waffle.setRadius(5);

    int radius;
    radius = waffle.radius; // private 멤버 접근 불가
    radius = waffle.getRadius();
}
```

접근자 함수의 장점

❖ 다양한 레벨의 접근 권한 설정 가능
- 멤버 변수를 public으로 지정하면 클래스 밖에서 멤버 변수에 대한 모든 접근을 허용하게 된다.
- 접근자 함수를 사용할 때는 get 함수만 정의할 수도 있고, set 함수만 정의할 수도 있고, get/set 함수를 모두 정의할 수도 있고, get/set 함수를 모두 정의하지 않을 수도 있다.
- 문서 읽기, 읽기/쓰기

❖ 멤버 변수의 값을 변경하거나 읽어올 때, 멤버 변수의 값에 대한 유효성 검사 가능

```
bool Circle::setRadius(int r) {
   if (r > 30) return false;
   radius = r;
   return true;
}
```

목차

const 객체

❖ const 객체

- **멤버 변수의 값을 변경할 수 없음**
  const 객체에는 대입할 수 없음

```
Circle pizza1(5);
const Circle pizza2; // 디폴트 생성자로 초기화
// const Circle pizza2(5); // 매개변수가 있는 생성자를 통해서도 초기화 가능
pizza2 = pizza1;     // const 객체에는 대입할 수 없으므로 컴파일 에러
```

- **const 멤버 함수만 호출 가능**

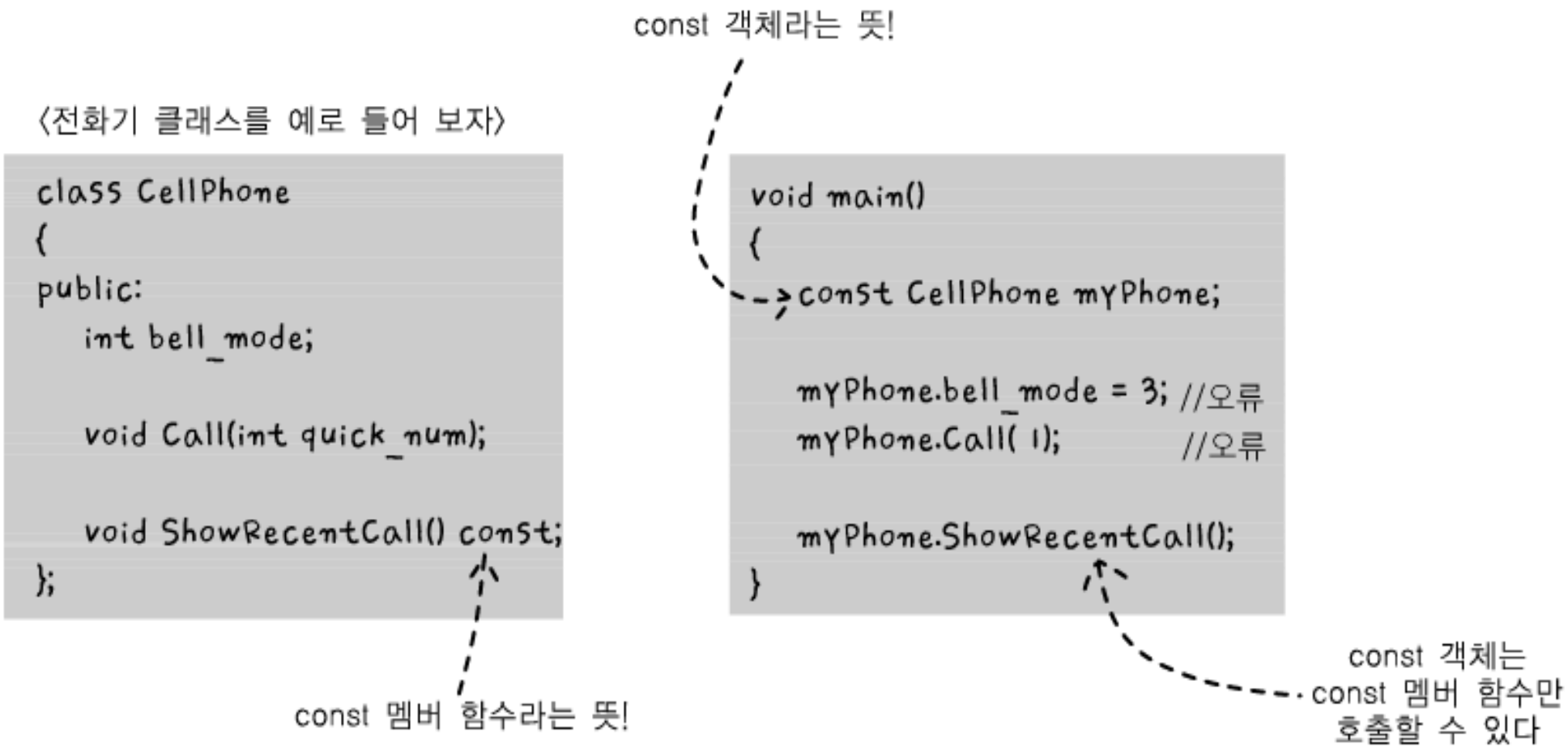
➔ **const 멤버 함수를 지정할 필요가 있다!!!**

❖ const 객체의 사용

- 객체를 함수의 입력 인자로 전달할 때 주로 사용
- Call by value로 객체를 전달하는 경우, 많은 메모리공간 필요 -> 메모리 효율성 감소
- 따라서, 객체와 같이 크기가 큰 것을 인자로 넘길 땐, 참조형식으로 전달하는게 효율적
- 참조 객체 전달 시 인자로 전달된 원본 객체의 데이터의 변경 가능성 존재
- const 객체 사용을 통해 방지 가능

```
void ShowData(Circle& s);
void ShowData(const Circle& s);
                    // s는 입력 인자, 원본 데이터를 변경할 수 없음을 보장
```

const 객체의 사용 예



const 멤버 함수

❖ 객체의 값을 변경하지 않기로 약속하는 멤버 함수
- "**이 함수에서는 멤버 변수의 값을 변경하지 않으므로, const 객체가 이 멤버함수를 호출해도 안전하다.**"는 의미
- 클래스의 사용자에게 멤버 함수에 대한 명확한 정보를 제공

❖ const 멤버 함수로 지정하는 방법
- 함수의 선언과 정의 양쪽 모두에 **const** 키워드 지정

```
class 클래스이름 {
    멤버 함수 선언 const;
};
반환형 클래스이름::멤버함수이름(인자리스트) const {
}
```

❖ const 멤버 함수 안에서는
- 멤버 변수의 값 변경 불가능 / 다른 일반 멤버 함수 호출 불가능

멤버 함수를 const로 만드는 것의 의미

· 다른 개발자가 "아 이 함수는 멤버 변수의 값을 변경하지 않는구나"라고 생각하게 만든다.
· 실수로 멤버 변수의 값을 바꾸려고 하면 컴퓨터가 오류 메시지를 통해서 알려준다.
· const 객체를 사용해서 이 함수를 호출할 수 있다.

목차

static 멤버와 non-static 멤버의 특성

❖ static
- 변수와 함수에 대한 기억 부류의 한 종류
  - 지금까지 우리가 알던 전역, 지역 멤버들은 생존기간과 접근범위가 대부분 일치함
  - static 멤버의 생명 주기 – 프로그램이 시작될 때 생성, 프로그램 종료 시 소멸 **(전역 멤버와 같음)**
  - static 멤버의 사용 범위 – 선언된 범위, 접근 지정에 따라 다름 **(전역 멤버와 다름)**

구분	예시	접근 범위	생존 기간(메모리)
`static` 전역 변수	`static int x;`	해당 **파일 내부**에서만	프로그램 종료 시까지
`static` 지역 변수	함수 안 `static int`	해당 **함수 내부**에서만	프로그램 종료 시까지
`static` 클래스 멤버	`static int count;`	클래스명으로 접근 가능	프로그램 종료 시까지

❖ 클래스의 멤버
- static 멤버
  - 프로그램이 시작할 때 생성
  - 클래스 당 하나만 생성, 클래스 멤버라고 불림
  - 클래스의 모든 인스턴스(객체)들이 공유하는 멤버
- non-static 멤버
  - 객체가 생성될 때 함께 생성
  - 객체마다 객체 내에 생성
  - 인스턴스 멤버라고 불림

클래스에서 static 멤버 선언

❖ 멤버의 static 선언

```
class Circle {
public:
    static string name;
    Circle();
};

Circle::Circle() {
    name = "pizza";
}

string Circle::name;

int main() {
    Circle c = Circle();
    cout << c.name;
}
```

Static 멤버변수 선언

Static 멤버변수 초기화

Static 멤버변수 외부선언

❖ static 멤버 변수 생성

- **전역 변수로 생성**. 전체 프로그램 내에 **한 번만 생성**

❖ static 멤버 변수에 대한 **외부 선언**이 없으면 다음과 같은 링크 오류

컴파일 성공

링크 오류

```
1>------ 빌드 시작: 프로젝트: StaticSample1, 구성: Debug Win32 ------
1>  StaticSample1.cpp
1>StaticSample1.obj : error LNK2001: "public: static int Person::sharedMoney" (?sharedMoney@Person@@2HA) 외부 기호를 확인할 수 없습니다.
1>C:₩C++₩chap6₩defaultParameter₩Debug₩StaticSample1.exe : fatal error LNK1120: 1개의 확인할 수 없는 외부 참조입니다.
========== 빌드: 성공 0, 실패 1, 최신 0, 생략 0 ==========
```

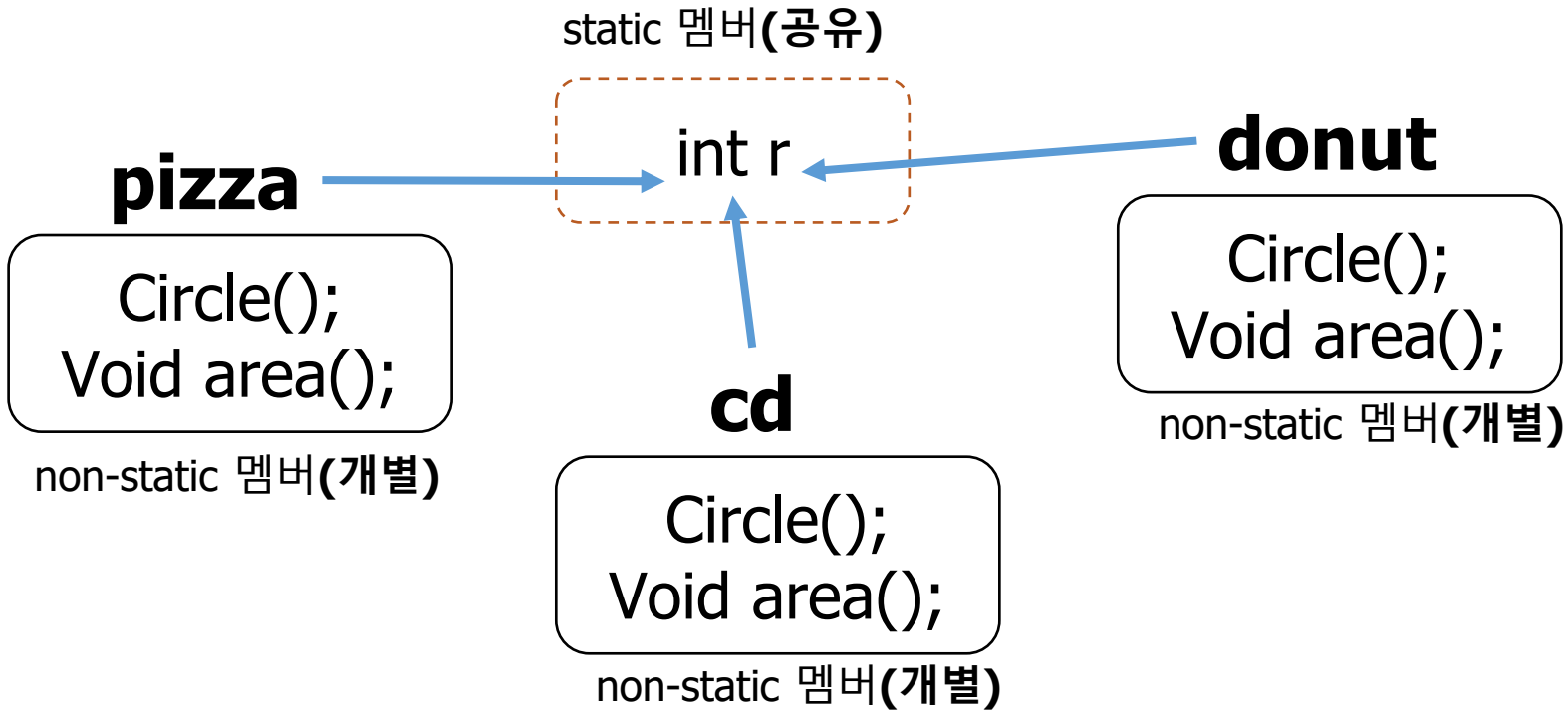
클래스에서 static 멤버와 non-static 멤버의 관계

클래스 선언

```
class Circle {
public:
    static int r;
    Circle();
    void area();
};
```

Circle pizza, cd, donut;

```
int main() {
    Circle pizza = Circle();
    Circle cd = Circle();
    Circle donut = Circle();
```



- 모든 객체가 static 멤버 r을 공유
- 모든 객체가 static 멤버 r에 대해 읽기, 쓰기 모두 가능

클래스에서 static 멤버 사용 : 클래스 이름을 통한 정적멤버 접근

❖ 정적 멤버 변수

- 같은 클래스의 객체들 사이에 공유되는 멤버 변수
- 보통 멤버처럼 접근할 수 있음
- 클래스 이름과 범위 지정자(::)로 접근 가능
  - 객체의 소유가 아니기 때문에!

```
strcpy(window::desktop,"HH");
window w1;
strcpy(w1.desktop,"HH");
```

❖ 정적 멤버 함수

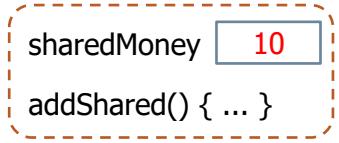
- 객체 없이 호출할 수 있는 멤버 함수
- 보통 멤버함수처럼 객체이름이나 포인터로 접근 가능
- 클래스 이름과 범위 지정자(::)로 접근 가능
  - 함수역시도 객체의 소유가 아니기 때문에!
- ~~this 포인터가 없음~~
  - ~~멤버 함수는 객체의 소유가 아님~~

```
window::closeAll();
window *w2;
w2 = &w1;
w2→closeAll();
w1.closeAll();
```

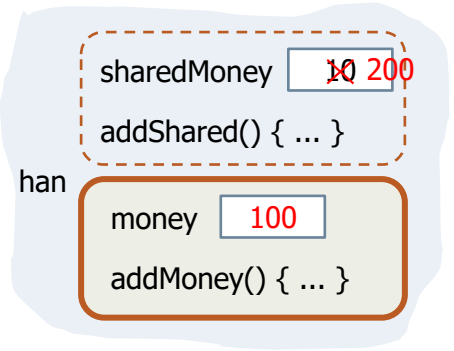
```
#include <iostream>
using namespace std;

class Person {
public:
    double money; // 개인 소유의 돈
    void addMoney(int money) {
        this->money += money;
    }

    static int sharedMoney; // 공금
    static void addShared(int n);
};

// static 변수 생성. 전역 공간에 생성
int Person::sharedMoney=10;  // 10으로 초기화
// static 함수 생성. 전역 공간에 생성, 일반적인 함수 초기화와 같음
void Person::addShared(int n){
        sharedMoney += n;
}

// main() 함수
int main() {
    Person han;
    han.money = 100; // han의 개인 돈=100
    han.sharedMoney = 200; // static 멤버 접근, 공금=200

    Person lee;
    lee.money = 150; // lee의 개인 돈=150
    lee.addMoney(200); // lee의 개인 돈=350
    lee.addShared(200); // static 멤버 접근, 공금=400

    cout << han.money << ' ' << lee.money << endl;
    cout << han.sharedMoney << ' ' << lee.sharedMoney << endl;
}
```

```
100 350
400 400
```

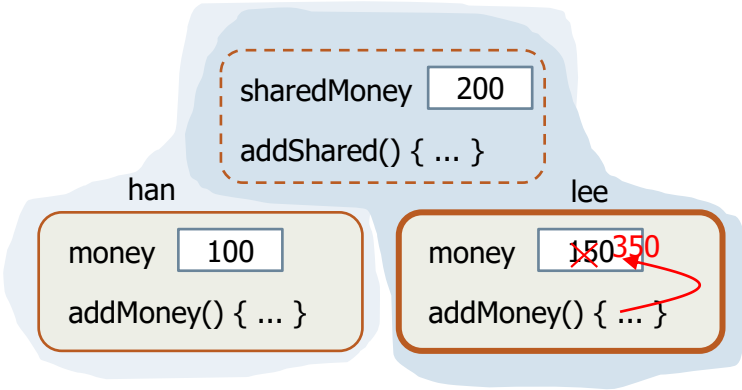
main()이 시작하기 직전



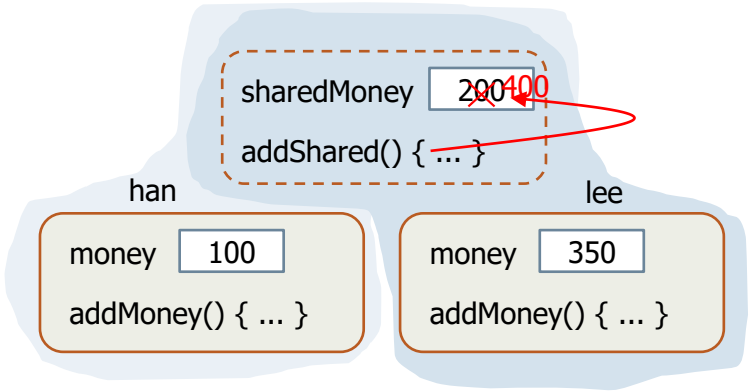
Person han;
han.money = 100;
han.sharedMoney = 200;



Person lee;
lee.money = 150;
lee.addMoney(200);



lee.addshared(200);



static 멤버 사용 : 클래스명과 범위 지정 연산자(::)로 접근

❖ 클래스 이름과 범위 지정 연산자(::)로 접근 가능

- static 멤버는 클래스마다 오직 한 개만 생성되기 때문

클래스명::static멤버

han.sharedMoney = 200;	<->	**Person::sharedMoney** = 200;
lee.addShared(200);	<->	**Person::addShared(200)**;

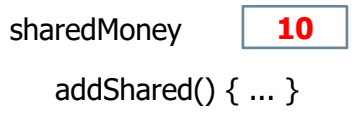
- non-static 멤버는 클래스 이름을 접근 불가

Person::money = 100; // 컴파일 오류. non-static 멤버는 클래스 명으로 접근불가
Person::addMoney(200); // 컴파일 오류. non-static 멤버는 클래스 명으로 접근불가

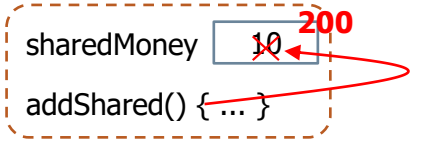
```
#include <iostream>
using namespace std;

class Person {
public:
    double money; // 개인 소유의 돈
    void addMoney(int money) {
        this->money += money;
    }

    static int sharedMoney; // 공금
    static void addShared(int n);
};

// static 변수 생성. 전역 공간에 생성
int Person::sharedMoney=10;  // 10으로 초기화
// static 함수 생성. 전역 공간에 생성, 일반적인 함수 초기화와 같음
void Person::addShared(int n){
        sharedMoney += n;
}

// main() 함수
int main() {
    Person::addShared(200);
    Person han;
    han.money = 100; // han의 개인 돈=100
    //han.sharedMoney = 200; // static 멤버 접근, 공금=200

    Person::sharedMoney = 300;   Person lee;
    lee.money = 150; // lee의 개인 돈=150
    lee.addMoney(200); // lee의 개인 돈=350
    lee.addShared(200); // static 멤버 접근, 공금=400

    cout << han.money << ' ' << lee.money << endl;
    cout << han.sharedMoney << ' ' << lee.sharedMoney << endl;
}
```

```
100 350
300 300
```

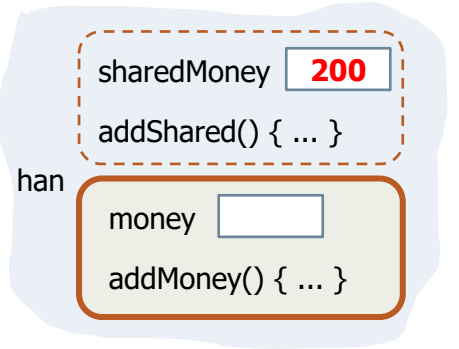
main()이 시작하기 직전



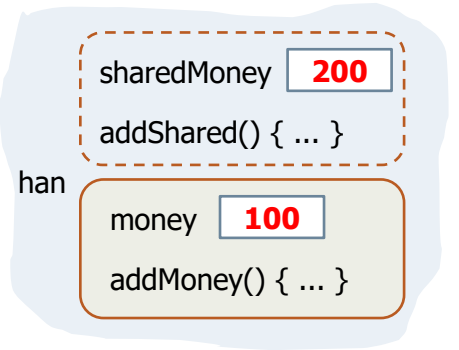
Person::addShared(200);



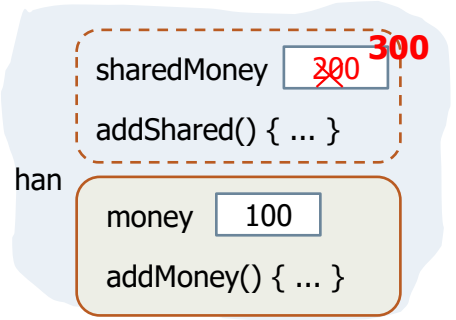
Person han;



han.money = 100;



Person::sharedMoney = 300;



static 활용

❖ static의 주요 활용

- 전역 변수나 전역 함수를 클래스에 캡슐화
  - 전역 변수나 전역 함수를 가능한 사용하지 않도록
  - 전역 변수나 전역 함수를 static으로 선언하여 클래스 멤버로 선언
- 객체 사이에 공유 변수를 만들고자 할 때
  - static 멤버를 선언하여 모든 객체들이 공유

❖ static 멤버 함수는 static 멤버만 접근 가능

- static 멤버 함수가 접근할 수 있는 것
  - static 멤버 함수
  - static 멤버 변수
  - 함수 내의 지역 변수
- static 멤버 함수는 non-static 멤버에 접근 불가
  - 객체가 생성되지 않은 시점에서 static 멤버 함수가 호출될 수 있기 때문
  - 아직 메모리에 올라가 있지도 않은 non-static 멤버들을 사용할 수 있기 때문에 이를 막기위한 조치

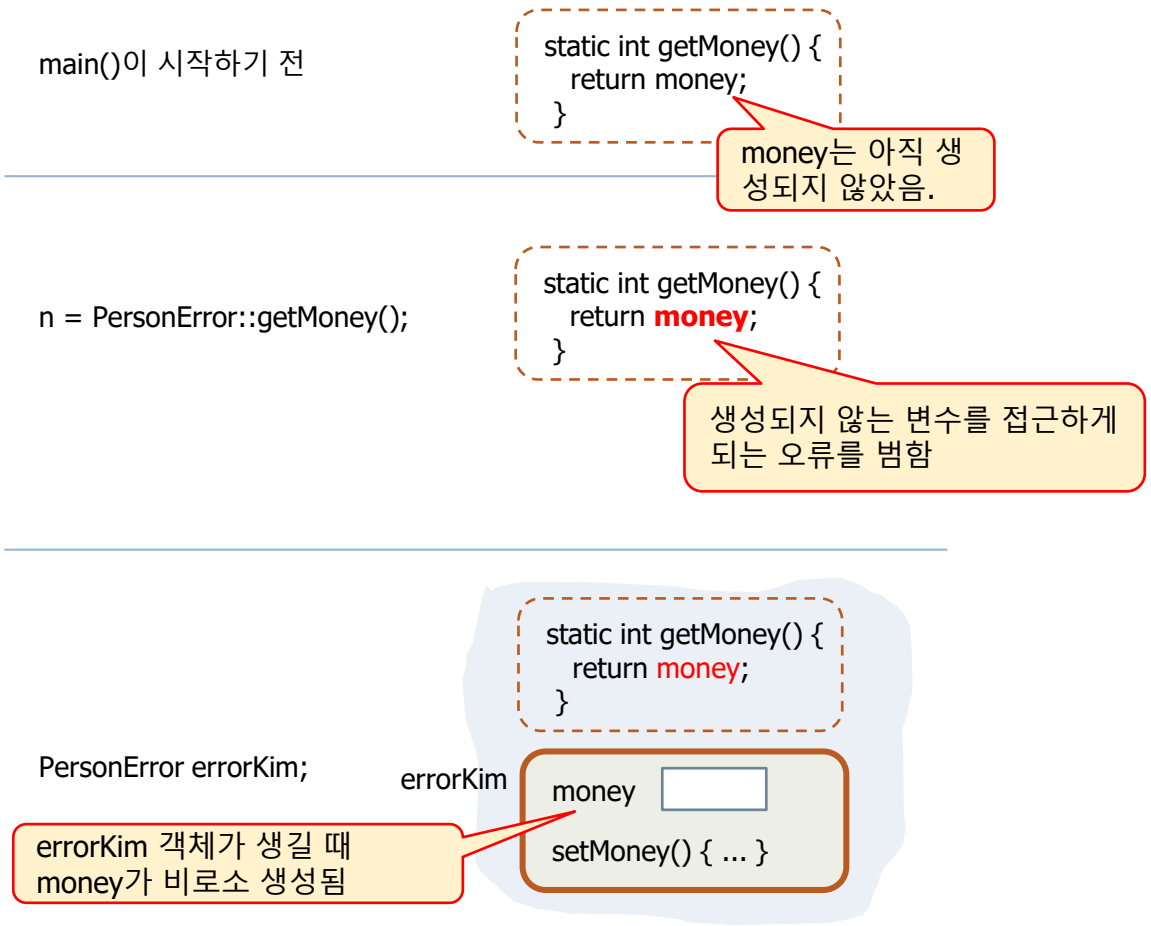
han.sharedMoney = 200;	<->	**Person::sharedMoney** = 200;
lee.addShared(200);	<->	**Person::addShared(200)**;

static 멤버 함수는 static 멤버만 접근 가능

static 멤버 함수 getMoney()가 non-static 멤버 변수 money에 접근하는 오류

```
class PersonError {
   int money;
public:
   static int getMoney() { return money; }

   void setMoney(int money) { // 정상 코드
      this->money = money;
   }
};

int main(){
   int n = PersonError::getMoney();

   PersonError errorKim;
   errorKim.setMoney(100);
}
```

컴파일 오류.
static 멤버 함수는 non-static 멤버에 접근할 수 없음.



non-static 멤버 함수는 static에 접근 가능

static 멤버 함수 getMoney()가 non-static 멤버 변수 money에 접근하는 오류

```
class Person {
   public: double money; // 개인 소유의 돈
   static int sharedMoney; // 공금
    ....
   int total() { // non-static 함수는 non-static이나 static 멤버에 모두 접근 가능
      return money + sharedMoney;
   }
};
```

non-static

static

클래스에서 static 멤버와 non-static 멤버 비교

항목	non-static 멤버	static 멤버

선언 사례	class Sample {
	  int n;
	  void f();
	};
		class Sample {
		  static int n;
		  static void f();
		};

공간 특성	멤버는 객체마다 별도 생성 • 인스턴스 멤버라고 부름	멤버는 클래스 당 하나 생성 • 멤버는 객체 내부가 아닌 별도의 공간에 생성 • 클래스 멤버라고 부름
시간적 특성	객체와 생명을 같이 함 • 객체 생성 시에 멤버 생성 • 객체 소멸 시 함께 소멸 • 객체 생성 후 객체 사용 가능	프로그램과 생명을 같이 함 • 프로그램 시작 시 멤버 생성 • 객체가 생기기 전에 이미 존재 • 객체가 사라져도 여전히 존재 • 프로그램이 종료될 때 함께 소멸
공유의 특성	공유되지 않음 • 멤버는 객체 별로 따로 공간 유지	동일한 클래스의 모든 객체들에 의해 공유됨

다음 수업

❖ 여러가지 객체의 생성 방법
- 1_ 객체배열과 객체포인터
- 2_ 동적메모리 할당 및 반환
- 3_ 객체 및 객체배열의 동적 생성 및 반환
- 4_ 멤버함수의 this 포인터