

# 03

# 학습 목표

1. 실세계의 객체와 C++ 객체에 대해 이해한다.
2. C++ 클래스를 작성할 수 있다.
3. 객체를 생성하고 활용할 수 있다.
4. 생성자와 소멸자를 알고 작성할 수 있다.
5. private, protected, public 접근 지정자를 이해한다.
6. 인라인 함수의 목적을 이해하고 활용할 수 있다.
7. C++ 구조체를 작성하고, 클래스와의 차이점을 안다.
8. 헤더 파일과 cpp 파일을 분리하여 C++ 프로그램을 작성할 수 있다.

# 세상의 모든 것이 객체이다.

3

## □ 세상 모든 것이 객체



TV



의자



책



집



카메라



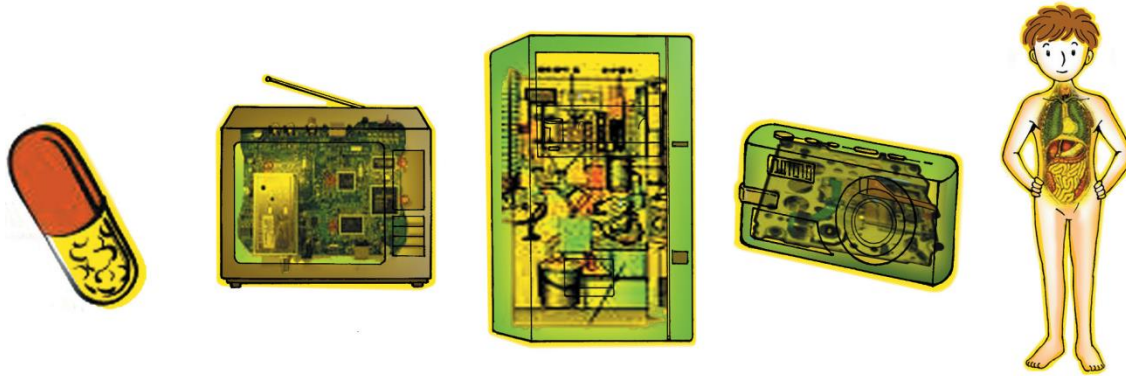
컴퓨터

# 객체는 캡슐화된다.

4

## □ 캡슐화(encapsulation)

- ▣ 객체의 본질적인 특성
- ▣ 객체를 캡슐로 싸서 그 내부를 보호하고 볼 수 없게 함
  - 캡슐에 든 약은 어떤 색인지 어떤 성분인지 보이지 않고, 외부로부터 안전
- ▣ 캡슐화 사례



## ▣ 캡슐화의 목적

- 객체 내 데이터에 대한 보안, 보호, 외부 접근 제한

# 토끼의 간과 객체의 캡슐화

5



쯔쯔, 용왕님이 C++를 알았으면 객체의 요소가 캡슐에 싸여 있는 것을 알았을 텐데... 어떻게 간이 몸 바깥에 있을 수 있어!





# 객체의 일부 요소는 공개된다.

6

## □ 객체의 일부분 공개

- ▣ 외부와의 인터페이스(정보 교환 및 통신)를 위해 객체의 일부분 공개
- ▣ TV 객체의 경우, On/Off 버튼, 밝기 조절, 채널 조절, 음량 조절 버튼 노출. 리모콘 객체와 통신하기 위함



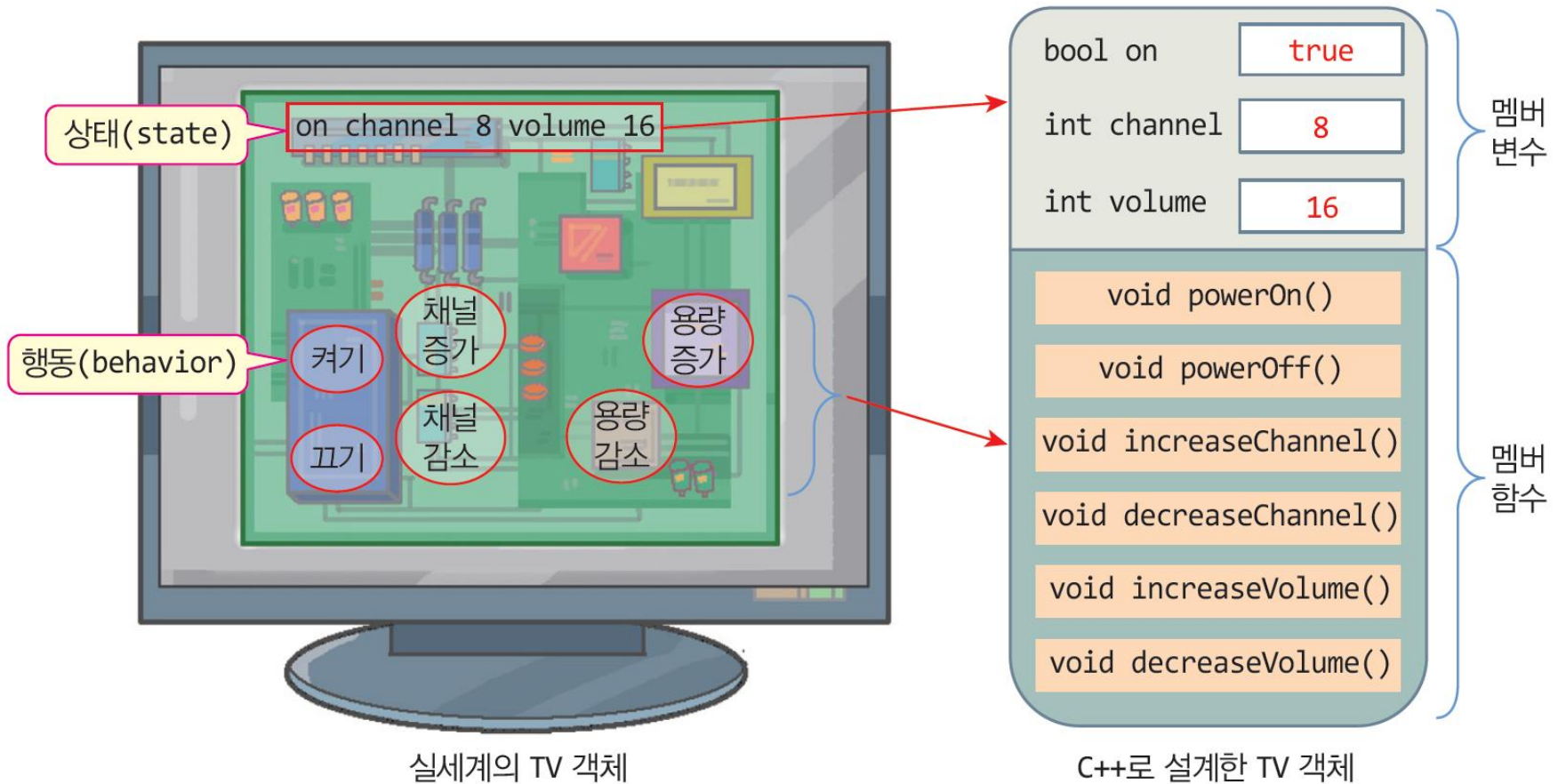
# C++ 객체는 멤버 함수와 멤버 변수로 구성된다.

7

- 객체는 상태(state)와 행동(behavior)으로 구성
- TV 객체 사례
  - ▣ 상태
    - on/off 속성 - 현재 작동 중인지 표시
    - 채널(channel) - 현재 방송중인 채널
    - 음량(volume) - 현재 출력되는 소리 크기
  - ▣ 행동
    - 켜기(power on)
    - 끄기(power off)
    - 채널 증가(increase channel)
    - 채널 감소(decrease channel)
    - 음량 증가(increase volume)
    - 음량 줄이기(decrease volume)

# TV와 C++로 설계된 TV 객체

8





# C++ 클래스와 C++ 객체

9

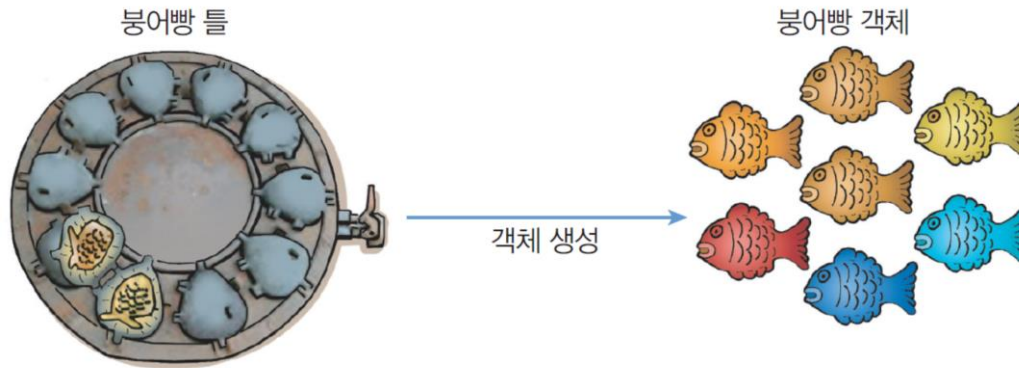
## □ 클래스

- ▣ 객체를 만들어내기 위해 정의된 설계도, 틀
- ▣ 클래스는 객체가 아님. 실체도 아님
- ▣ 멤버 변수와 멤버 함수 선언

## □ 객체

- ▣ 객체는 생성될 때 클래스의 모양을 그대로 가지고 탄생
- ▣ 멤버 변수와 멤버 함수로 구성
- ▣ 메모리에 생성, 실체(instance)라고도 부름
- ▣ 하나의 클래스 틀에서 찍어낸 여러 개의 객체 생성 가능
- ▣ 객체들은 상호 별도의 공간에 생성

# 클래스와 객체 관계

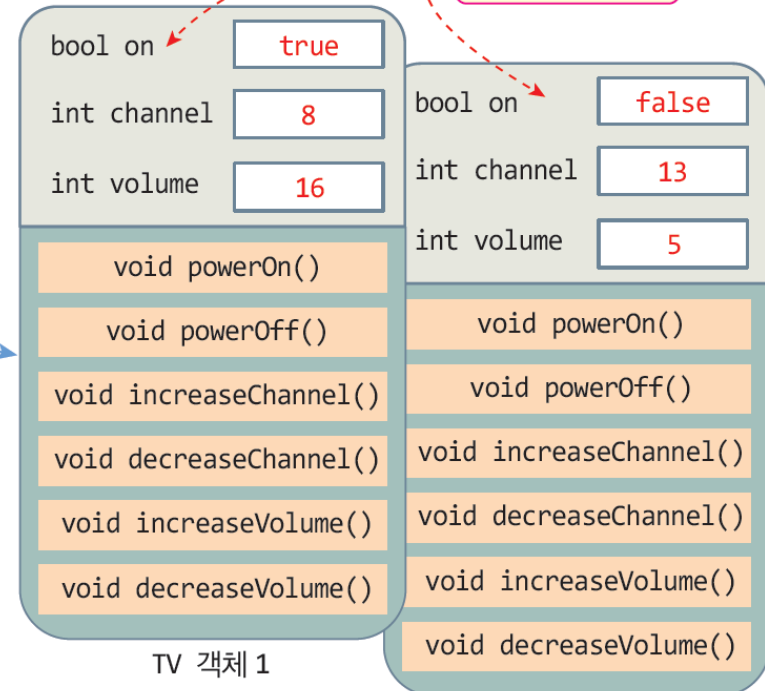


(a) 붕어빵 틀과 붕어빵 객체들

```
class TV {  
    bool on;  
    int channel;  
    int volume;  
public:  
    void powerOn() { ... }  
    void powerOff() { ... }  
    void increaseChannel() { ... }  
    void decreaseChannel() { ... }  
    void increaseVolume() { ... }  
    void decreaseVolume() { ... }  
};
```

TV 클래스

객체 생성



객체들은 서로  
별도의 영역에  
멤버들을 만든다.

TV 객체 2

TV 객체 1

(b) C++로 표현한 TV 클래스와 TV 객체들

# C++ 클래스 만들기

11

- 클래스 작성
  - ▣ 멤버 변수와 멤버 함수로 구성
  - ▣ 클래스 선언부와 클래스 구현부로 구성
- 클래스 선언부(class declaration)
  - ▣ class 키워드를 이용하여 클래스 선언
  - ▣ 멤버 변수와 멤버 함수 선언
    - 멤버 변수는 클래스 선언 내에서 초기화할 수 없음
    - 멤버 함수는 원형(prototype) 형태로 선언
  - ▣ 멤버에 대한 접근 권한 지정
    - private, public, protected 중의 하나
    - 디폴트는 private
    - public : 다른 모든 클래스나 객체에서 멤버의 접근이 가능함을 표시
- 클래스 구현부(class implementation)
  - ▣ 클래스에 정의된 모든 멤버 함수 구현

# 클래스 만들기 설명

12

클래스의 선언은  
class 키워드 이용

클래스  
이름

멤버에 대한 접근 지정자

세미콜론으로 끝남

```
class Circle {  
public:  
    int radius; // 멤버 변수  
    double getArea(); // 멤버 함수  
};
```

클래스  
선언부

함수의 리  
턴 타입

클래스  
이름

범위지정  
연산자

멤버 함수명과  
매개변수

```
double Circle :: getArea() {  
    return 3.14*radius*radius;  
}
```

클래스  
구현부

클래스 선언과 클래스 구현  
으로 분리하는 이유는 클래  
스를 다른 파일에서 활용하  
기 위함

# 예제 3-1 Circle 클래스의 객체 생성 및 활용

13

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

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

Circle 선언부

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

Circle 구현부

```
int main() {
```

객체 donut 생성

```
    Circle donut;
```

donut의 멤버  
변수 접근

```
    donut.radius = 1; // donut 객체의 반지름을 1로 설정
```

```
    double area = donut.getArea(); // donut 객체의 면적 알아내기
```

donut의 멤버  
함수 호출

```
    cout << "donut 면적은 " << area << endl;
```

```
    Circle pizza;
```

```
    pizza.radius = 30; // pizza 객체의 반지름을 30으로 설정
```

```
    area = pizza.getArea(); // pizza 객체의 면적 알아내기
```

```
    cout << "pizza 면적은 " << area << endl;
```

```
}
```

donut 면적은 3.14  
pizza 면적은 2826

# 객체 생성 및 활용 설명

14

## □ 객체 이름 및 객체 생성

```
Circle donut; // 이름이 donut 인 Circle 타입의 객체 생성
```

객체의 타입.  
클래스 이름

객체 이름

## □ 객체의 멤버 변수 접근

```
donut.radius = 1; // donut 객체의 radius 멤버 값을 1로 설정
```

객체 이름

멤버 변수

객체 이름과  
멤버 사이에  
· 연산자

## □ 객체의 멤버 함수 접근

```
double area = donut.getArea(); // donut 객체의 면적 알아내기
```

객체 이름

멤버 함수 호출

객체 이름과  
멤버 사이에  
· 연산자



# 객체 이름과 생성, 접근 과정

15

(1) Circle donut;

객체 이름

객체가 생성되면  
메모리가 할당된다.

int radius   
double getArea() {...}

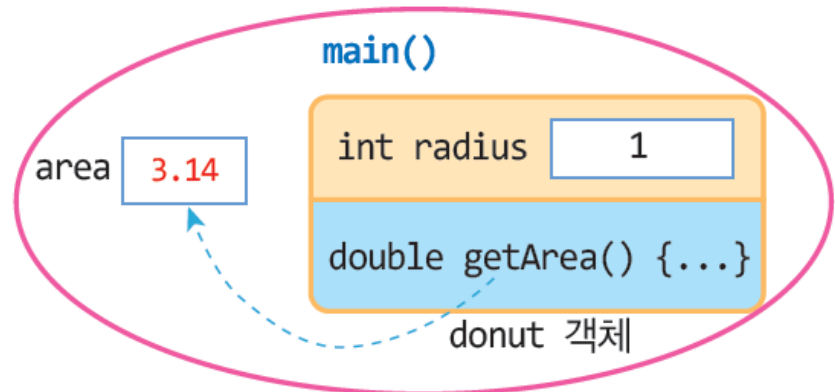
donut 객체

(2) donut.radius = 1;

int radius   
double getArea() {...}

donut 객체

(3) double area = donut.getArea();



## 예제 3-2(실습) – Rectangle 클래스 만들기

16

다음 main() 함수가 잘 작동하도록 너비(width)와 높이(height)를 가지고 면적 계산 기능을 가진 Rectangle 클래스를 작성하고 전체 프로그램을 완성하라.

```
int main() {  
    Rectangle rect;  
    rect.width = 3;  
    rect.height = 5;  
    cout << "사각형의 면적은 " << rect.getArea() << endl;  
}
```

사각형의 면적은 15

## 예제 3-2(실습) 정답

17

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

class Rectangle { // Rectangle 클래스 선언부
public:
    int width;
    int height;
    int getArea(); // 면적을 계산하여 리턴하는 함수
};

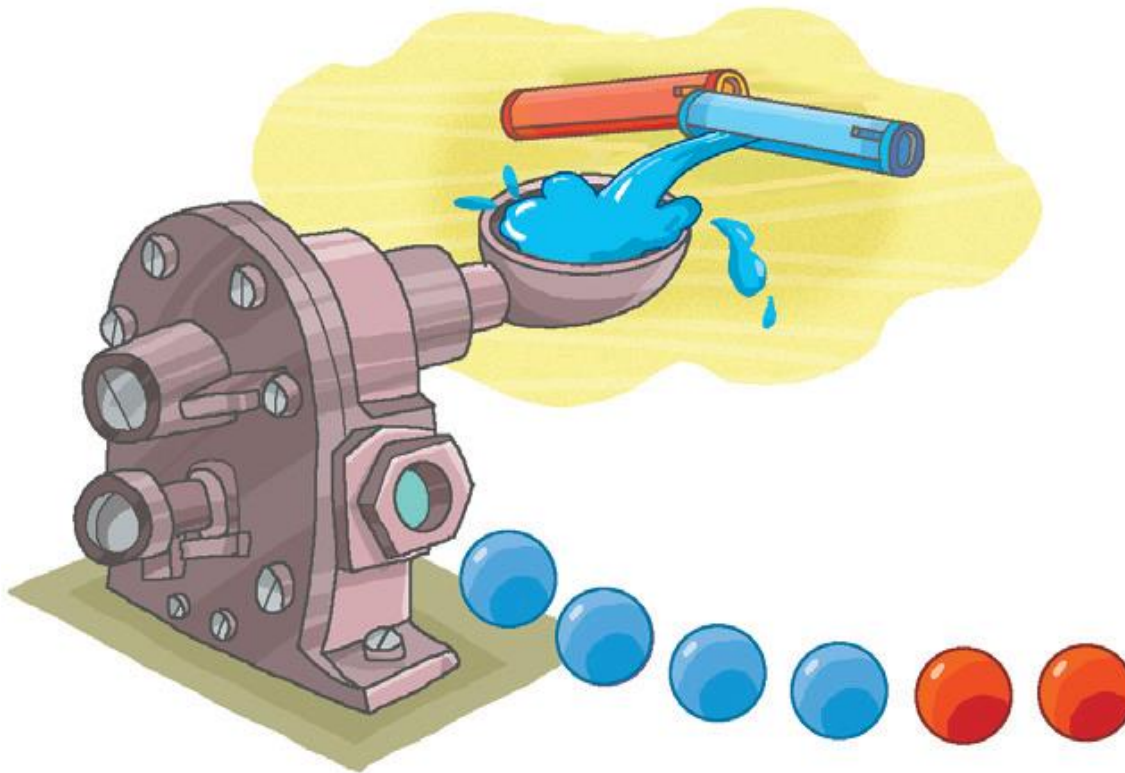
int Rectangle::getArea() { // Rectangle 클래스 구현부
    return width*height;
}

int main() {
    Rectangle rect;
    rect.width = 3;
    rect.height = 5;
    cout << "사각형의 면적은 " << rect.getArea() << endl;
}
```

사각형의 면적은 15

# 탁구공 생산 장치와 생성자

18



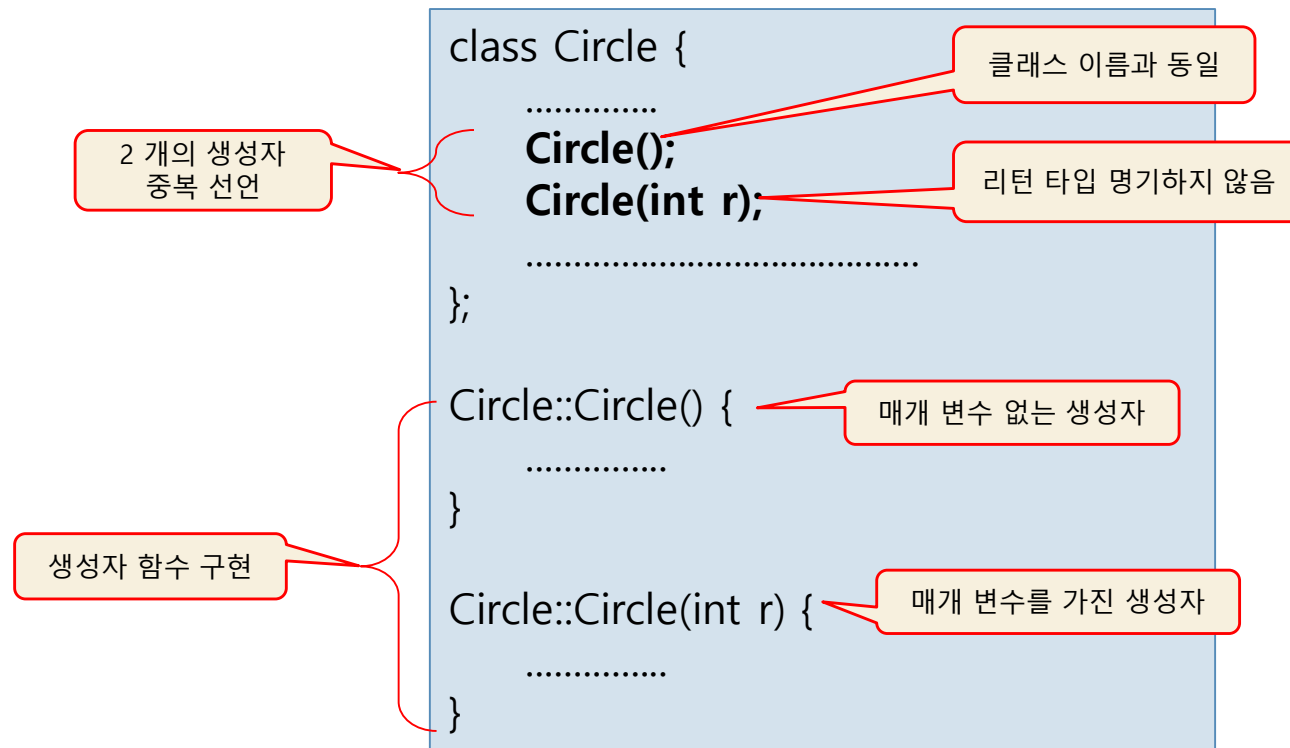
똑 같은 탁구공이  
생산되지만 페인트  
색으로 초기화된다.

# 생성자

19

## □ 생성자(constructor)

- 객체가 **생성**되는 시점에서 **자동**으로 호출되는 **멤버 함수**
- 클래스 이름과 동일한 멤버 함수



# 생성자 함수의 특징

20

- ▣ 생성자의 목적
  - 객체가 생성될 때 객체가 필요한 초기화를 위해
    - 멤버 변수 값 초기화, 메모리 할당, 파일 열기, 네트워크 연결 등
- ▣ 생성자 이름
  - 반드시 클래스 이름과 동일
- ▣ 생성자는 리턴 타입을 선언하지 않는다.
  - 리턴 타입 없음. void 타입도 안됨
- ▣ 객체 생성 시 오직 한 번만 호출
  - 자동으로 호출됨. 임의로 호출할 수 없음. 각 객체마다 생성자 실행
- ▣ 생성자는 중복 가능
  - 생성자는 한 클래스 내에 여러 개 가능
  - 중복된 생성자 중 하나만 실행
- ▣ 생성자가 선언되어 있지 않으면 기본 생성자 자동으로 생성
  - 기본 생성자 – 매개 변수 없는 생성자
  - 컴파일러에 의해 자동 생성



# 예제 3-3 2 개의 생성자를 가진 Circle 클래스

21

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

```
class Circle {
public:
    int radius;
    Circle(); // 매개 변수 없는 생성자
    Circle(int r); // 매개 변수 있는 생성자
    double getArea();
};
```

```
Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

```
Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

Circle(); 자동 호출

Circle(30); 자동 호출

```
int main() {
```

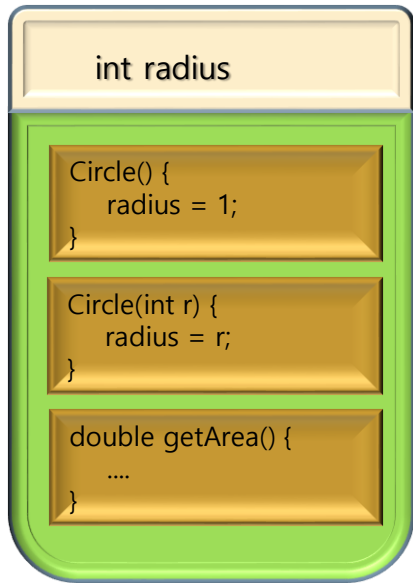
```
    Circle donut; // 매개 변수 없는 생성자 호출
    double area = donut.getArea();
    cout << "donut 면적은 " << area << endl;
```

```
    Circle pizza(30); // 매개 변수 있는 생성자 호출
    area = pizza.getArea();
    cout << "pizza 면적은 " << area << endl;
}
```

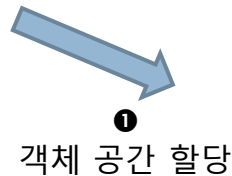
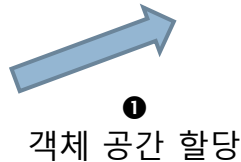
반지름 1 원 생성  
donut 면적은 3.14  
반지름 30 원 생성  
pizza 면적은 2826

# 객체 생성 및 생성자 실행 과정

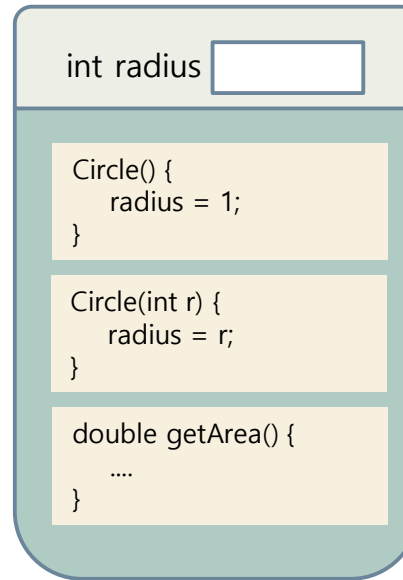
Circle donut;



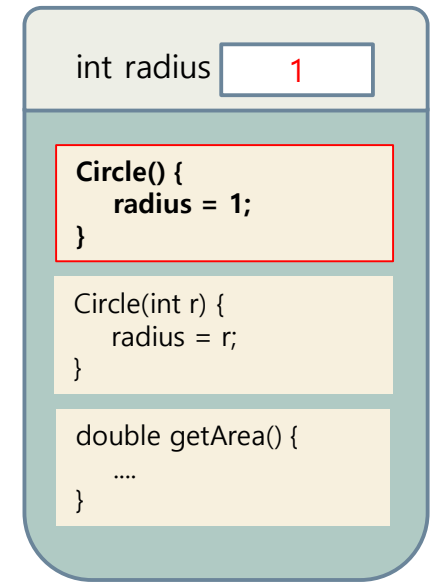
Circle 클래스



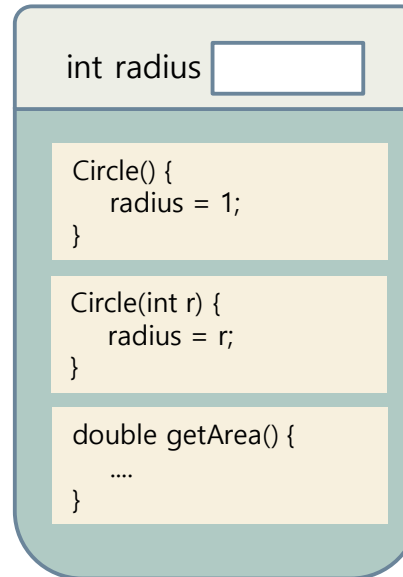
Circle pizza(30);



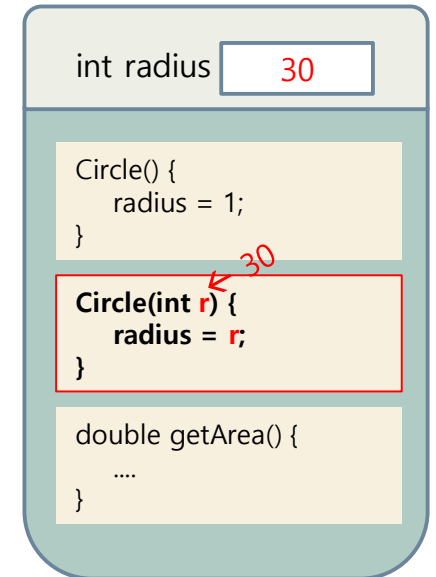
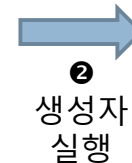
donut 객체



donut 객체



pizza 객체



pizza 객체

# 생성자가 다른 생성자 호출(위임 생성자)

23

- 여러 생성자에 중복 작성된 코드의 간소화
  - 타겟 생성자와 이를 호출하는 위임 생성자로 나누어 작성
    - 타겟 생성자 : 객체 초기화를 전담하는 생성자
    - 위임 생성자 : 타겟 생성자를 호출하는 생성자, 객체 초기화를 타겟 생성자에 위임

```
Circle::Circle() {  
    radius = 1;  
    cout << "반지름 " << radius << " 원 생성" << endl;  
}  
  
Circle::Circle(int r) {  
    radius = r;  
    cout << "반지름 " << radius << " 원 생성" << endl;  
}
```

여러 생성자에 코드 중복

위임 생성자

```
Circle::Circle() : Circle(1) { } // Circle(int r)의 생성자 호출
```

타겟 생성자

```
Circle::Circle(int r) {  
    radius = r;  
    cout << "반지름 " << radius << " 원 생성" << endl;  
}
```

호출

r에 1 전달

간소화된 코드

# 예제 3-4 생성자에서 다른 생성자 호출 연습(위임 생성자 만들기)

24

예제 3-3을 수정하여 객체 초기화를 전담하는 타겟 생성자와 타겟 생성자에 개체 초기화를 위임하는 위임 생성자로 재작성하라.

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

```
class Circle {
public:
    int radius;
    Circle(); // 위임 생성자
    Circle(int r); // 타겟 생성자
    double getArea();
};
```

```
Circle::Circle() : Circle(1) {} // 위임 생성자
```

호출. r에 1 전달

```
Circle::Circle(int r) { // 타겟 생성자
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

호출

```
int main() {
```

```
    Circle donut; // 매개 변수 없는 생성자 호출
    double area = donut.getArea();
    cout << "donut 면적은 " << area << endl;
```

```
    Circle pizza(30); // 매개 변수 있는 생성자 호출
    area = pizza.getArea();
    cout << "pizza 면적은 " << area << endl;
```

```
}
```

반지름 1 원 생성  
donut 면적은 3.14  
반지름 30 원 생성  
pizza 면적은 2826

# 다양한 생성자의 멤버 변수 초기화 방법

25

```
class Point {  
    int x, y;  
public:  
    Point();  
    Point(int a, int b);  
};
```

(1) 생성자 코드에서  
    멤버 변수 초기화

```
Point::Point() { x = 0; y = 0; }  
Point::Point(int a, int b) { x = a; y = b; }
```

(2) 생성자 서두에  
    초깃값으로 초기화

```
Point::Point() : x(0), y(0) { // 멤버 변수 x, y를 0으로 초기화  
}  
Point::Point(int a, int b) // 멤버 변수 x=a로, y=b로 초기화  
    : x(a), y(b) { // 콜론(:) 이하 부분을 밑줄에 써도 됨  
}
```

(3) 클래스 선언부  
    에서 직접 초기화

```
class Point {  
    int x=0, y=0; // 클래스 선언부에서 x, y를 0으로 직접 초기화  
public:  
    ...  
};
```

# 예제 3-5 멤버변수의 초기화와 위임 생성자 활용

26

다음 Point 클래스의 멤버 x, y를  
생성자 서두에 초기값으로 초기화하고  
위임 생성자를 이용하여 재작성하라.

```
class Point {  
    int x, y;  
public:  
    Point();  
    Point(int a, int b);  
};  
Point::Point() { x = 0; y = 0; }  
Point::Point(int a, int b) { x = a; y = b; }
```

```
#include <iostream>  
using namespace std;  
  
class Point {  
    int x, y;  
public:  
    Point();  
    Point(int a, int b);  
    void show() { cout << "(" << x << ", " << y << ")" << endl; }  
};  
  
Point::Point() : Point(0, 0) { } // 위임 생성자  
Point::Point(int a, int b) // 타겟 생성자  
    : x(a), y(b) { }  
  
int main() {  
    Point origin;  
    Point target(10, 20);  
    origin.show();  
    target.show();  
}
```

(0, 0)  
(10, 20)



# 기본 생성자

27

## 1. 생성자는 꼭 있어야 하는가?

- ▣ 예. C++ 컴파일러는 객체가 생성될 때, 생성자 반드시 호출

## 2. 개발자가 클래스에 생성자를 작성해 놓지 않으면?

- ▣ 컴파일러에 의해 기본 생성자가 자동으로 생성

### □ 기본 생성자란?

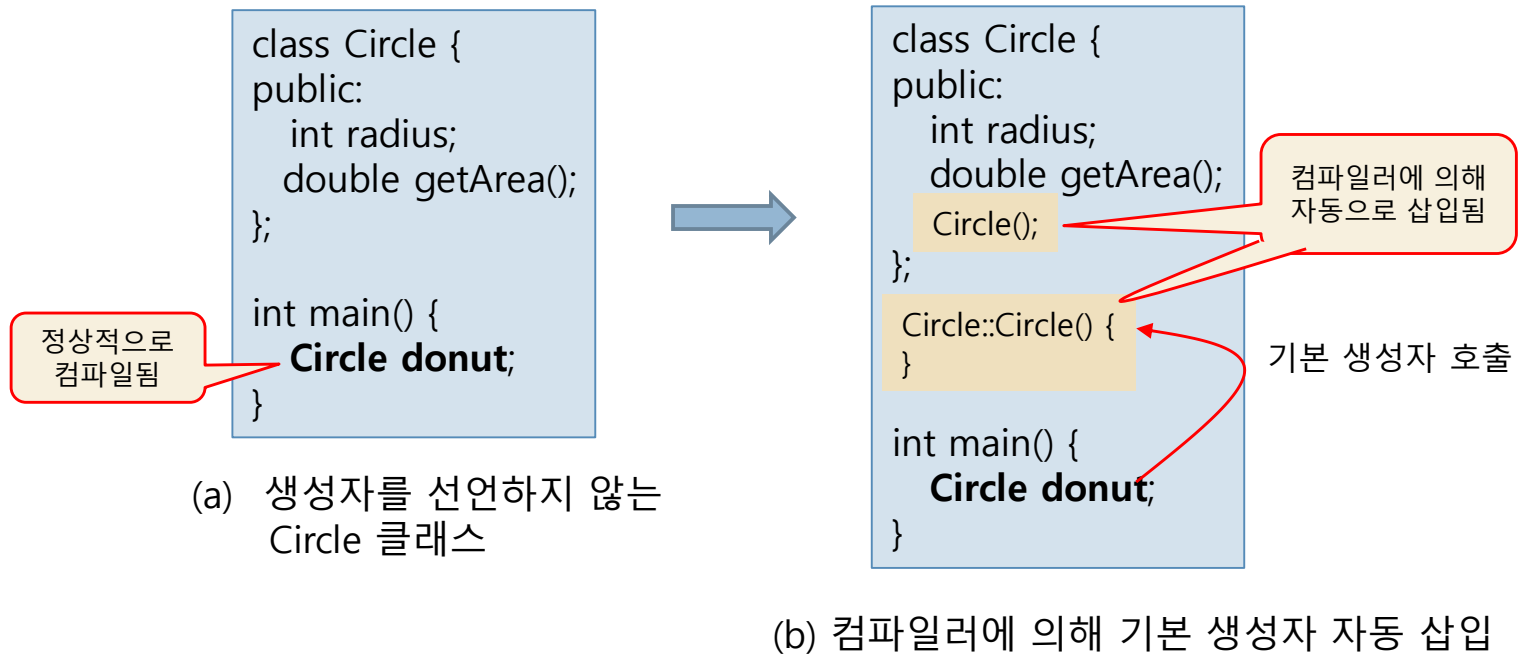
- ▣ 클래스에 생성자가 하나도 선언되어 있지 않은 경우, 컴파일러가 대신 삽입해주는 생성자
- ▣ 매개 변수 없는 생성자
- ▣ 디폴트 생성자라고도 부름

```
class Circle {  
    ....  
    Circle(); // 기본 생성자  
};
```

# 기본 생성자가 자동으로 생성되는 경우

28

- 생성자가 하나도 작성되어 있지 않은 클래스의 경우
  - ▣ 컴파일러가 기본 생성자 자동 생성



# 기본 생성자가 자동으로 생성되지 않는 경우

29

- 생성자가 하나라도 선언된 클래스의 경우
  - ▣ 컴파일러는 기본 생성자를 자동 생성하지 않음

```
class Circle {  
public:  
    int radius;  
    double getArea();  
    Circle(int r);  
};  
  
Circle::Circle(int r) {  
    radius = r;  
}  
  
int main() {  
    Circle pizza(30);  
    Circle donut;  
}
```

Circle 클래스에 생성자가 선언되어 있기 때문에, 컴파일러는 기본 생성자를 자동 생성하지 않음

호출

컴파일 오류.  
기본 생성자 없음

# 예제 3-6(실습) – Rectangle 클래스 만들기

30

다음 main() 함수가 잘 작동하도록 Rectangle 클래스를 작성하고 프로그램을 완성하라. Rectangle 클래스는 width와 height의 두 멤버 변수와 3 개의 생성자, 그리고 isSquare() 함수를 가진다.

```
int main() {  
    Rectangle rect1;  
    Rectangle rect2(3, 5);  
    Rectangle rect3(3);  
  
    if(rect1.isSquare()) cout << "rect1은 정사각형이다." << endl;  
    if(rect2.isSquare()) cout << "rect2는 정사각형이다." << endl;  
    if(rect3.isSquare()) cout << "rect3는 정사각형이다." << endl;  
}
```

rect1은 정사각형이다.  
rect3는 정사각형이다.

## 예제 3-6 정답

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

class Rectangle {
public:
    int width, height;

    Rectangle();
    Rectangle(int w, int h);
    Rectangle(int length);
    bool isSquare();
};

Rectangle::Rectangle() {
    width = height = 1;
}

Rectangle::Rectangle(int w, int h) {
    width = w; height = h;
}

Rectangle::Rectangle(int length) {
    width = height = length;
}

// 정사각형이면 true를 리턴하는 멤버 함수
bool Rectangle::isSquare() {
    if(width == height) return true;
    else return false;
}
```

```
int main() {
    Rectangle rect1;
    Rectangle rect2(3, 5);
    Rectangle rect3(3);

    if(rect1.isSquare()) cout << "rect1은 정사각형이다." << endl ;
    if(rect2.isSquare()) cout << "rect2는 정사각형이다." << endl;
    if(rect3.isSquare()) cout << "rect3는 정사각형이다." << endl;
}
```

3 개의 생성자가 필요함

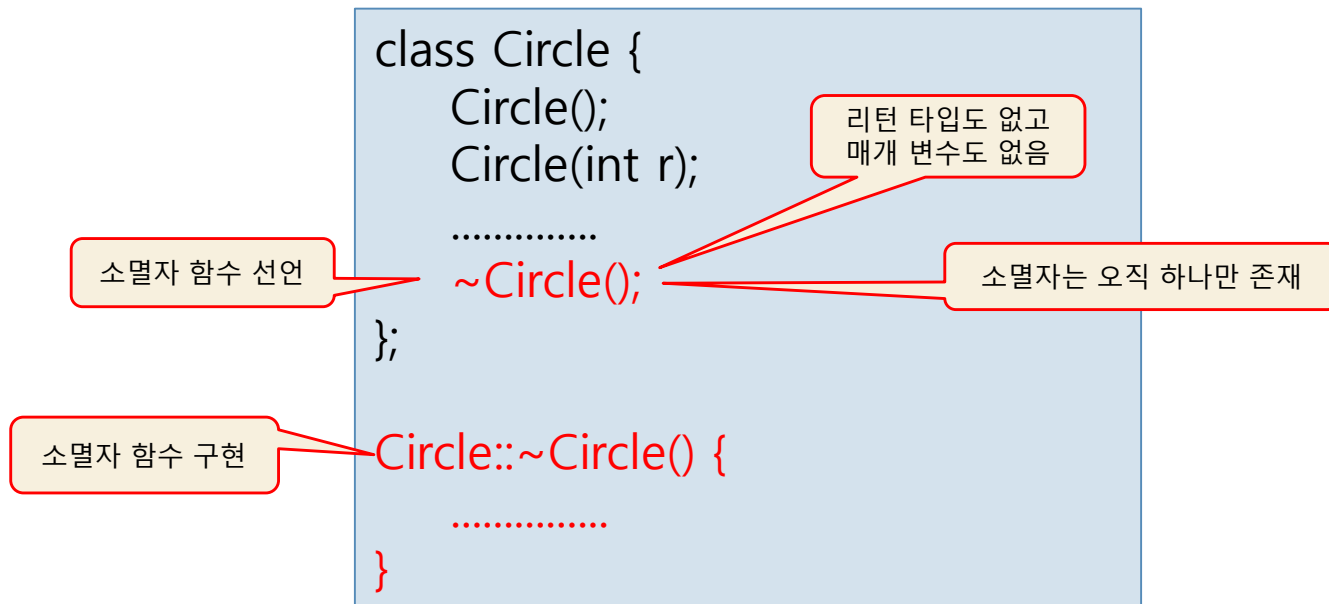
rect1은 정사각형이다.  
rect3는 정사각형이다.

# 소멸자

32

## □ 소멸자

- ▣ 객체가 **소멸**되는 시점에서 **자동**으로 호출되는 **함수**
  - 오직 한번만 자동 호출, 임의로 호출할 수 없음
  - 객체 메모리 소멸 직전 호출됨





# 소멸자 특징

33

## □ 소멸자의 목적

- 객체가 사라질 때 마무리 작업을 위함
- 실행 도중 동적으로 할당 받은 메모리 해제, 파일 저장 및 닫기, 네트워크 닫기 등

## □ 소멸자 함수의 이름은 클래스 이름 앞에 ~를 붙인다.

- 예) Circle::~~Circle() { ... }

## □ 소멸자는 리턴 타입이 없고, 어떤 값도 리턴하면 안됨

- 리턴 타입 선언 불가

## □ 중복 불가능

- 소멸자는 한 클래스 내에 오직 한 개만 작성 가능
- 소멸자는 매개 변수 없는 함수

## □ 소멸자가 선언되어 있지 않으면 기본 소멸자가 자동 생성

- 컴파일러에 의해 기본 소멸자 코드 생성
- 컴파일러가 생성한 기본 소멸자 : 아무 것도 하지 않고 단순 리턴

# 예제 3-7 Circle 클래스에 소멸자 작성 및 실행

34

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

class Circle {
public:
    int radius;

    Circle();
    Circle(int r);
    ~Circle(); // 소멸자
    double getArea();
};

Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::~~Circle() {
    cout << "반지름 " << radius << " 원 소멸" << endl;
}
```

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

```
int main() {
    Circle donut;
    Circle pizza(30);

    return 0;
}
```

main() 함수가 종료하면 main() 함수의 스택에 생성된 pizza, donut 객체가 소멸된다.

반지름 1 원 생성  
반지름 30 원 생성  
반지름 30 원 소멸  
반지름 1 원 소멸

객체는 생성의 반대 순으로 소멸된다.

# 생성자/소멸자 실행 순서

35

- 객체가 선언된 위치에 따른 분류
  - ▣ 지역 객체
    - 함수 내에 선언된 객체로서, 함수가 종료하면 소멸된다.
  - ▣ 전역 객체
    - 함수의 바깥에 선언된 객체로서, 프로그램이 종료할 때 소멸된다.
- 객체 생성 순서
  - ▣ 전역 객체는 프로그램에 선언된 순서로 생성
  - ▣ 지역 객체는 함수가 호출되는 순간에 순서대로 생성
- 객체 소멸 순서
  - ▣ 함수가 종료하면, 지역 객체가 생성된 순서의 역순으로 소멸
  - ▣ 프로그램이 종료하면, 전역 객체가 생성된 순서의 역순으로 소멸
- new를 이용하여 동적으로 생성된 객체의 경우
  - ▣ new를 실행하는 순간 객체 생성
  - ▣ delete 연산자를 실행할 때 객체 소멸

# 예제 3-8 지역 객체와 전역 객체의 생성 및 소멸 순서

36

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

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

Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::~~Circle() {
    cout << "반지름 " << radius << " 원 소멸" << endl;
}

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

다음 프로그램의 실행 결과는 무엇인가?

```
Circle globalDonut(1000);
Circle globalPizza(2000);
```

전역 객체 생성

```
void f() {
    Circle fDonut(100);
    Circle fPizza(200);
}
```

지역 객체 생성

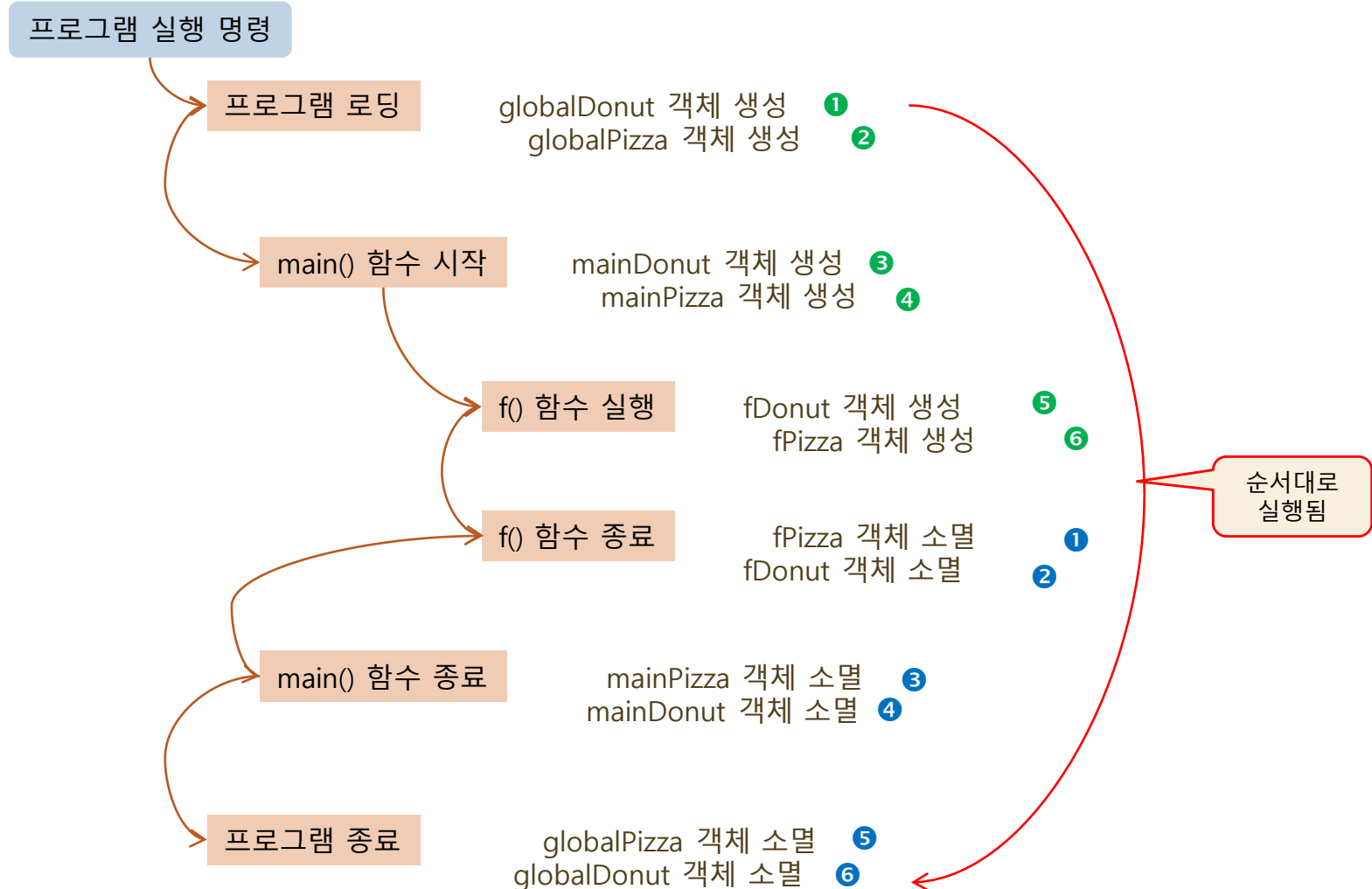
```
int main() {
    Circle mainDonut;
    Circle mainPizza(30);
    f();
}
```

지역 객체 생성

반지름 1000 원 생성  
반지름 2000 원 생성  
반지름 1 원 생성  
반지름 30 원 생성  
반지름 100 원 생성  
반지름 200 원 생성  
반지름 200 원 소멸  
반지름 100 원 소멸  
반지름 30 원 소멸  
반지름 1 원 소멸  
반지름 2000 원 소멸  
반지름 1000 원 소멸

# 예제 3-8의 지역 객체와 전역 객체의 생성과 소멸 과정

37



# 접근 지정자

38

- 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
- 멤버에 대한 3 가지 접근 지정자
  - ▣ private
    - 동일한 클래스의 멤버 함수에만 제한함
  - ▣ public
    - 모든 다른 클래스에 허용
  - ▣ protected
    - 클래스 자신과 상속받은 자식 클래스에만 허용

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

# 중복 접근 지정과 디폴트 접근 지정

39

## 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

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

40

```
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;  
}
```

(a) 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  
}
```

(b) 멤버 변수를 private으로 선언한 바람직한 사례



```

#include <iostream>
using namespace std;

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

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)
}

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

```

예제 3-9 다음 소스의 컴파일 오류가 발생하는 곳은 어디인가?

```

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

```

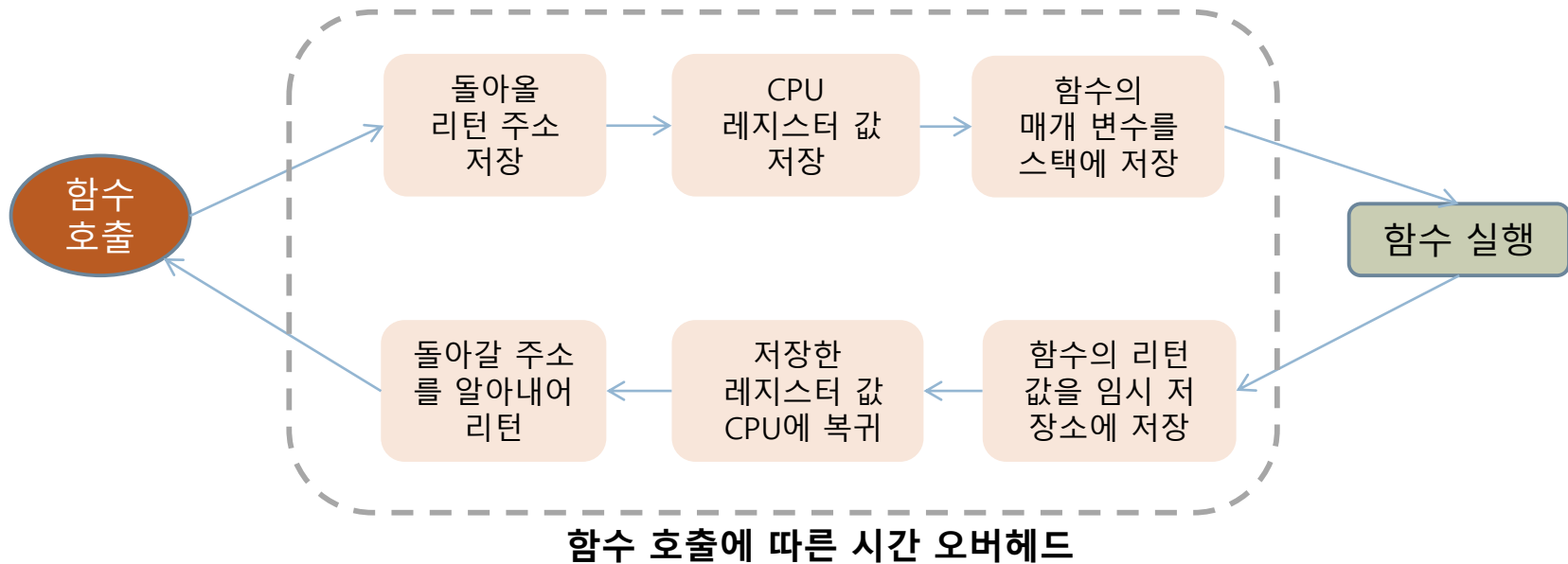
정답

- (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 있다. 생성자를 private으로 선언하는 경우는 한 클래스에서 오직 하나의 객체만 생성할 수 있도록 하기 위한 것으로 부록 D의 singleton 패턴을 참조하라.

# 함수 호출에 따른 시간 오버헤드

42



작은 크기의 함수를 호출하면, 함수 실행 시간에 비해, 호출을 위해 소요되는 추가적인 시간 오버헤드가 상대적으로 크다.

# 함수 호출에 따른 오버헤드가 심각한 사례

43

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

```
int odd(int x) {
    return (x%2);
}
```

10000번의 함수 호출.  
호출에 따른 엄청난 오  
버헤드 시간이 소모됨.

```
int main() {
    int sum = 0;

    // 1에서 10000까지의 홀수의 합 계산
    for(int i=1; i<=10000; i++) {
        if(odd(i))
            sum += i;
    }
    cout << sum;
}
```

odd() 함수의 코드  
x%2를 계산하는 시간보  
다 odd() 함수 호출에  
따른 오버헤드가 더 크  
며, 루프를 돌게 되면 오  
버헤드는 가중됩니다.



25000000

# 인라인 함수

44

- 인라인 함수
  - ▣ inline 키워드로 선언된 함수
- 인라인 함수에 대한 처리
  - ▣ 인라인 함수를 호출하는 곳에 인라인 함수 코드를 확장 삽입
    - 매크로와 유사
    - 코드 확장 후 인라인 함수는 사라짐
  - ▣ 인라인 함수 호출
    - 함수 호출에 따른 오버헤드 존재하지 않음
    - 프로그램의 실행 속도 개선
  - ▣ 컴파일러에 의해 이루어짐
- 인라인 함수의 목적
  - ▣ C++ 프로그램의 실행 속도 향상
    - 자주 호출되는 짧은 코드의 함수 호출에 대한 시간 소모를 줄임
    - C++에는 짧은 코드의 멤버 함수가 많기 때문

# 인라인 함수 사례

45

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

```
inline int odd(int x) {
    return (x%2);
}
```

```
int main() {
    int sum = 0;

    for(int i=1; i<=10000; i++) {
        if(odd(i))
            sum += i;
    }
    cout << sum;
}
```

컴파일러에 의해  
inline 함수의 코드  
확장 삽입

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

```
int main() {
    int sum = 0;

    for(int i=1; i<=10000; i++) {
        if(i%2)
            sum += i;
    }
    cout << sum;
}
```

컴파일러는 inline 처리 후,  
확장된 C++ 소스 파일을  
컴파일 한다.

## 인라인 제약 사항

- inline은 컴파일러에게 주는 요구 메시지
- 컴파일러가 판단하여 inline 요구를 수용할 지 결정
- recursion, 긴 함수, static 변수, 반복문, switch 문, goto 문 등을 가진 함수는 수용하지 않음

# 인라인 함수 장단점 및 자동 인라인

46

## □ 장점

- ▣ 프로그램의 실행 시간이 빨라진다.

## □ 단점

- ▣ 인라인 함수 코드의 삽입으로 컴파일된 전체 코드 크기 증가
  - 통계적으로 최대 30% 증가
  - 짧은 코드의 함수를 인라인으로 선언하는 것이 좋음

# 자동 인라인 함수

47

- ▣ 자동 인라인 함수 : 클래스 선언부에 구현된 멤버 함수
  - inline으로 선언할 필요 없음
  - 컴파일러에 의해 자동으로 인라인 처리
  - 생성자를 포함, 모든 함수가 자동 인라인 함수 가능

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

inline  
멤버 함수

```
inline Circle::Circle() {  
    radius = 1;  
}
```

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

inline  
멤버 함수

```
inline double Circle::getArea() {  
    return 3.14*radius*radius;  
}
```

=

```
class Circle {  
private:  
    int radius;  
public:  
    Circle() { // 자동 인라인 함수  
        radius = 1;  
    }  
  
    Circle(int r);  
    double getArea() { // 자동 인라인 함수  
        return 3.14*radius*radius;  
    }  
};  
  
Circle::Circle(int r) {  
    radius = r;  
}
```

(a) 멤버함수를 inline으로 선언하는 경우

(b) 자동 인라인 함수로 처리되는 경우

# C++ 구조체

48

## □ 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

## □ C++에서 구조체를 수용한 이유?

- ▣ C 언어와의 호환성 때문
  - C의 구조체 100% 호환 수용
  - C 소스를 그대로 가져다 쓰기 위해

## □ 구조체 객체 생성

- ▣ struct 키워드 생략

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

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



# 구조체와 클래스의 디폴트 접근 지정 비교

49

구조체에서  
디폴트 접근 지정은  
**public**

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

동일

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

클래스에서  
디폴트 접근 지정은  
**private**

# 예제 3-10 Circle 클래스를 C++ 구조체를 이용하여 재작성

50

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

// C++ 구조체 선언
struct StructCircle {
private:
    int radius;
public:
    StructCircle(int r) { radius = r; } // 구조체의 생성자
    double getArea();
};

double StructCircle::getArea() {
    return 3.14*radius*radius;
}

int main() {
    StructCircle waffle(3);
    cout << "면적은 " << waffle.getArea();
}
```

면적은 28.26

# 바람직한 C++ 프로그램 작성법

51

- 클래스를 헤더 파일과 cpp 파일로 분리하여 작성
  - ▣ 클래스마다 분리 저장
  - ▣ 클래스 선언 부
    - 헤더 파일(.h)에 저장
  - ▣ 클래스 구현 부
    - cpp 파일에 저장
    - 클래스가 선언된 헤더 파일 include
  - ▣ main() 등 전역 함수나 변수는 다른 cpp 파일에 분산 저장
    - 필요하면 클래스가 선언된 헤더 파일 include
- 목적
  - ▣ 클래스 재사용

## 예제 3-3의 소스를 헤더 파일과 cpp 파일로 분리하여 작성한 사례

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

Circle.h

```
#include <iostream>  
using namespace std;  
  
#include "Circle.h"  
  
Circle::Circle() {  
    radius = 1;  
    cout << "반지름 " << radius;  
    cout << " 원 생성" << endl;  
}  
  
Circle::Circle(int r) {  
    radius = r;  
    cout << "반지름 " << radius;  
    cout << " 원 생성" << endl;  
}  
  
double Circle::getArea() {  
    return 3.14*radius*radius;  
}
```

반지름 1 원 생성  
donut 면적은 3.14  
반지름 30 원 생성  
pizza 면적은 2826

```
#include <iostream>  
using namespace std;  
  
#include "Circle.h"  
  
int main() {  
    Circle donut;  
    double area = donut.getArea();  
    cout << "donut 면적은 ";  
    cout << area << endl;  
  
    Circle pizza(30);  
    area = pizza.getArea();  
    cout << "pizza 면적은 ";  
    cout << area << endl;  
}
```

main.cpp

컴파일

Circle.cpp

컴파일

Circle.obj

main.obj

링킹

main.exe

# 헤더 파일의 중복 include 문제

53

- 헤더 파일을 중복 include 할 때 생기는 문제

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

#include "Circle.h"
#include "Circle.h" // 컴파일 오류 발생
#include "Circle.h"

int main() {
    .....
}
```

circle.h(4): error C2011: 'Circle' : 'class' 형식 재정의

# 헤더 파일의 중복 include 문제를 조건 컴파일로 해결

54

조건 컴파일 문.  
Circle.h를 여러 번  
include해도 문제  
없게 하기 위함

조건 컴파일 문의 상수(CIRCLE\_H)는  
다른 조건 컴파일 상수와 충돌을 피하기 위해  
클래스의 이름으로 하는 것이 좋음.

```
#ifndef CIRCLE_H
#define CIRCLE_H

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

#endif
```

Circle.h

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

```
#include "Circle.h"
#include "Circle.h"
#include "Circle.h"
```

컴파일 오류 없음

```
int main() {
    .....
}
```

main.cpp

## include 뵚

```
#ifndef CIRCLE_H
#define CIRCLE_H

// Circle 클래스 선언
class Circle {
private:
    int radius;
public:
    Circle();
    Circle(int r);
    double getArea();
};

#endif
```

circle.h

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

#include "circle.h"

// Circle 클래스 구현. 모든 멤버 함수를 작성한다.
Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성\n";
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성\n";
}

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

circle.cpp

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

#include "circle.h"

int main() {
    Circle donut;
    double area = donut.getArea();
    cout << " donut의 면적은 " << area << "\n";

    Circle pizza(30);
    area = pizza.getArea();
    cout << "pizza의 면적은 " << area << "\n";
}
```

main.cpp

컴파일

circle.obj

main.obj

링킹

main.exe

## 예제 3-11 헤더 파일과 cpp 파일로 분리하기

56

아래의 소스를 헤더 파일과 cpp 파일로 분리하여 재작성하라.

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

class Adder { // 덧셈 모듈 클래스
    int op1, op2;
public:
    Adder(int a, int b);
    int process();
};

Adder::Adder(int a, int b) {
    op1 = a; op2 = b;
}

int Adder::process() {
    return op1 + op2;
}
```

```
class Calculator { // 계산기 클래스
public:
    void run();
};

void Calculator::run() {
    cout << "두 개의 수를 입력하세요>>";
    int a, b;
    cin >> a >> b; // 정수 두 개 입력
    Adder adder(a, b); // 덧셈기 생성
    cout << adder.process(); // 덧셈 계산
}

int main() {
    Calculator calc; // calc 객체 생성
    calc.run(); // 계산기 시작
}
```

두 개의 수를 입력하세요>>5 -20  
-15



# 예제 3-11 정답

Adder.h

```
#ifndef ADDER_H
#define ADDER_H

class Adder { // 덧셈 모듈 클래스
    int op1, op2;
public:
    Adder(int a, int b);
    int process();
};

#endif
```

Calculator.h

```
#ifndef CALCULATOR_H
#define CALCULATOR_H

class Calculator { // 계산기 클래스
public:
    void run();
};

#endif
```

Adder.cpp

```
#include "Adder.h"

Adder::Adder(int a, int b) {
    op1 = a; op2 = b;
}

int Adder::process() {
    return op1 + op2;
}
```

Calculator.cpp

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

#include "Calculator.h"
#include "Adder.h"

void Calculator::run() {
    cout << "두 개의 수를 입력하세요>>";
    int a, b;
    cin >> a >> b; // 정수 두 개 입력
    Adder adder(a, b); // 덧셈기 생성
    cout << adder.process(); // 덧셈 계산
}
```

main.cpp

```
#include "Calculator.h"

int main() {
    Calculator calc; // calc 객체 생성
    calc.run(); // 계산기 시작
}
```

두 개의 수를 입력하세요>>5 -20  
-15