객체지향 프로그램

- **실세계에 존재하는 하나의 사물에 (자동차, 사람, 건물) 대한 소프트웨어 적인 표현이다.**

- **즉, 객체는 현실 세계에서 일어나는 것을 프로그램으로 처리하게 한다.**

- **객체(object) =>** 정보를 관리하기 위하여 사람들이 의미를 부여하고 분류하는 논리적인 단위.

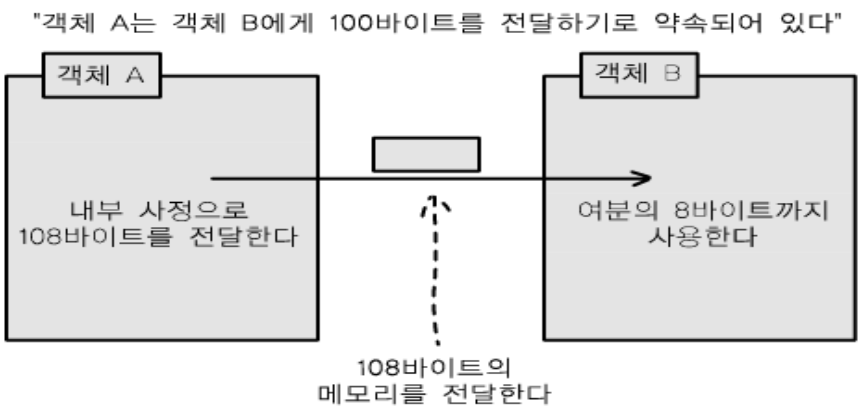
객체지향 프로그래밍

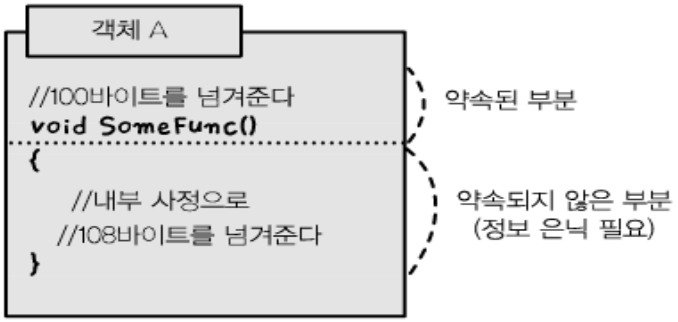
클래스(Classes)와 객체(Objects)

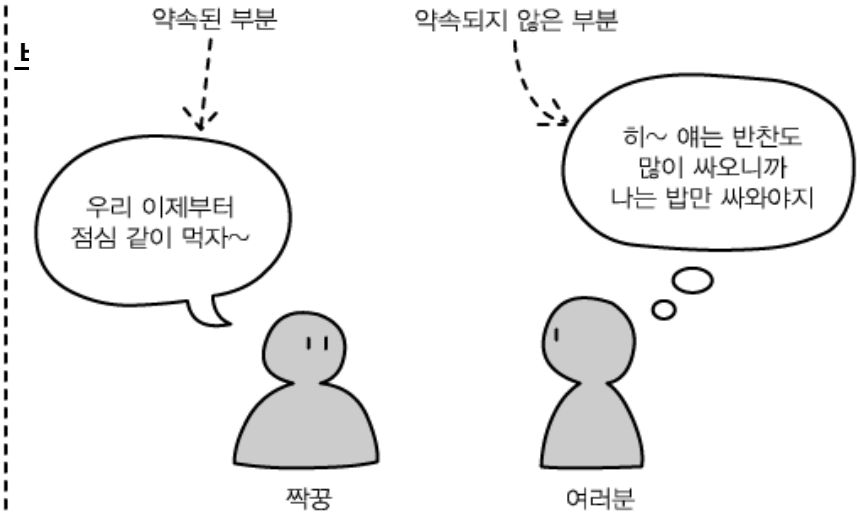
- 클래스와 객체는 '붕어빵 틀'과 '붕어빵'의 관계다. 혹은 '제품의 설계도'와 '제품'의 관계라고 말할 수도 있

정보은닉(Data Hiding)

- 정보은닉이란 객체간에 약속되지 않은 부분을 숨기는 것을 말한다.
  - 예) 약속되지 않은 여분 어긋나는 일이다.

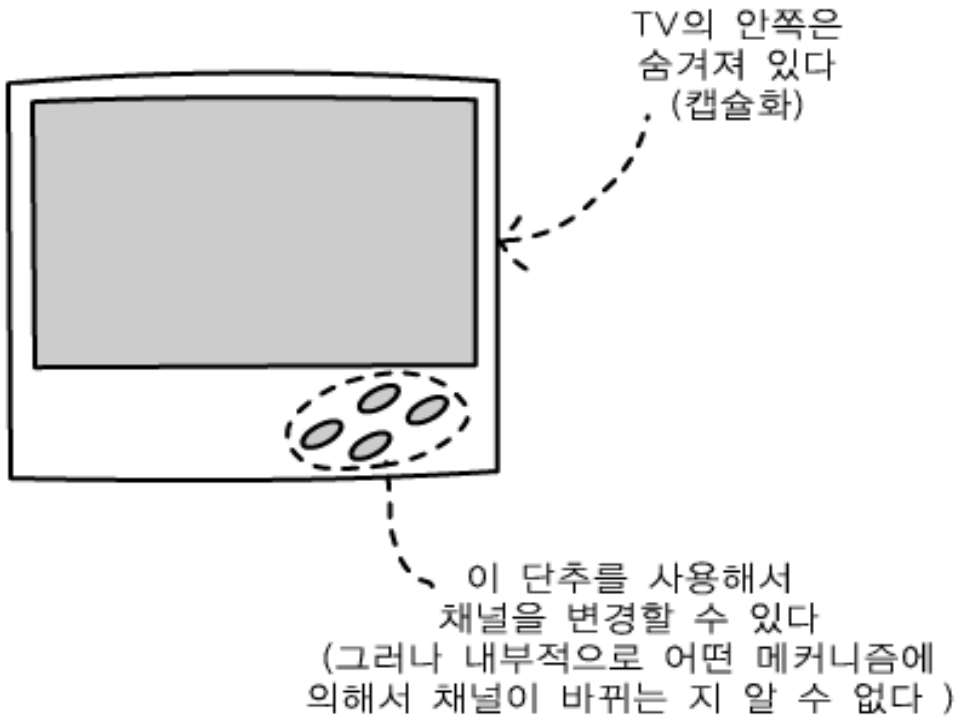






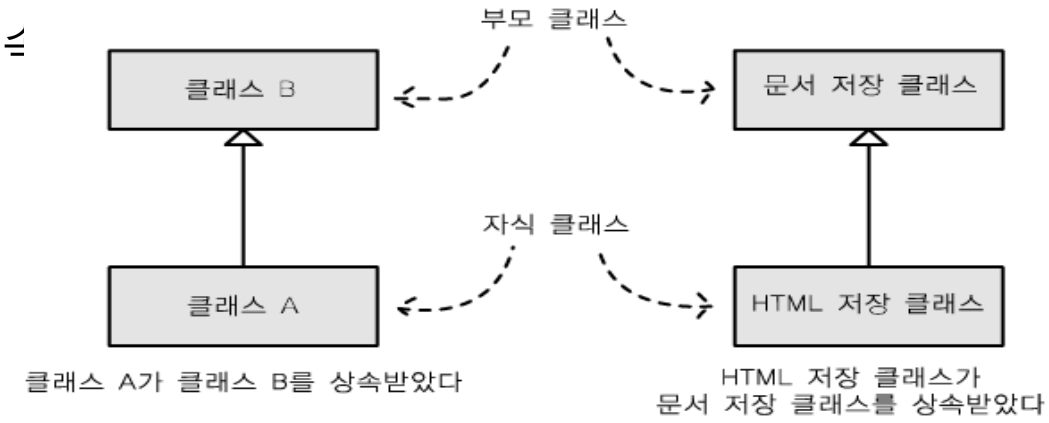
캡슐화(Encapsulation)

- 캡슐화란 약속되지 않은 부분은 감싸서 숨겨버리는 것을 말한다. 캡슐화를 통해서 정보은닉을 달성할 수 있다.

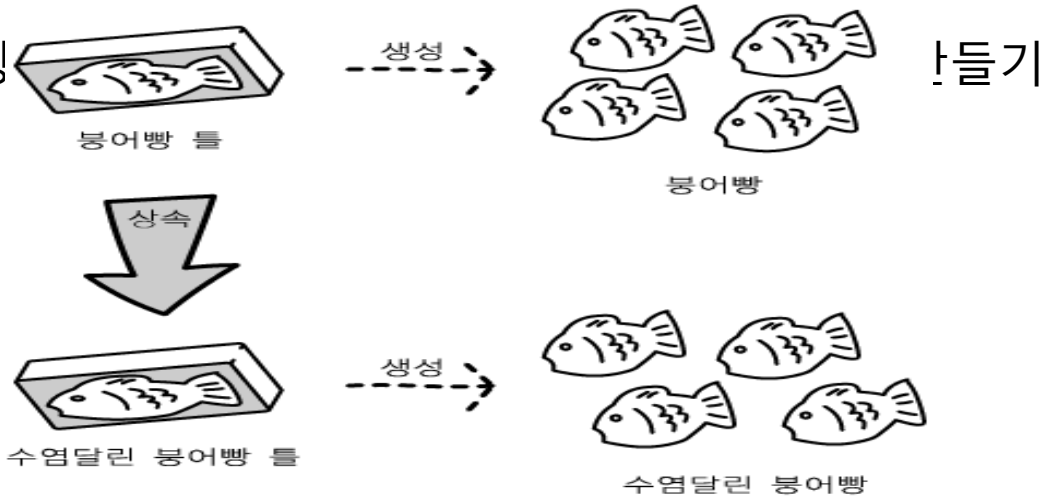


상속(Inheritance)

- 상속이란 기존 클래스를 토대로 새로운 클래스를 만드는 방법을 말한다.
  - 예) 상속

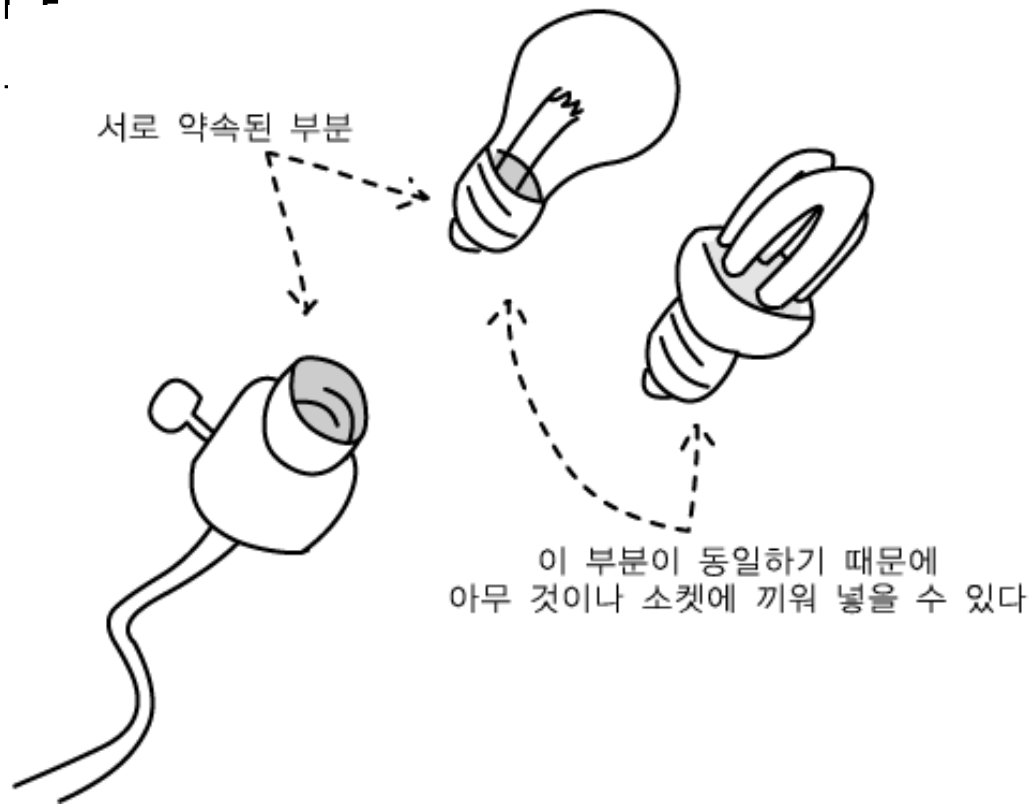


  - 예) 붕어빵 틀 만들기



다형성(Polymorphism)

- 다형성이란 서로 다른 객체를 동일한 방식으로 명령을 내릴 수 있는 성질을 말한다. 이 때 서로 다른 객체들은 같은 명령을 받지만 제각기 다른 방식으로 명령을 수행할 수 있다.
  - 예) 백열 ... 할 수 있다.



구조체

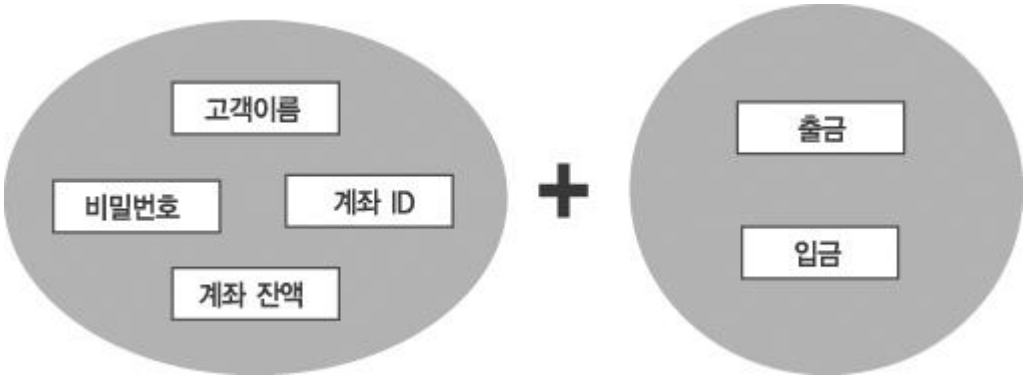
- **구조체의 유용성**
  - 관련 있는 데이터를 하나의 자료형으로 묶을 수 있다.
  - 함께 움직이는 데이터들을 묶어주는 효과!

- **C 언어의 구조체에 대한 불만**
  - 기본 자료형으로 인식해 주지 않는다.
  - C++은 사용자 정의 자료형도 기본형과 동일시하게 처리하라.

```
struct Person{
        int age;
        char name[10];
 };
 int main()
 {
      int a=10;
      Person p;
// struct Person p;
      return 0;
 }
```

구조체

- 함수를 넣으면 좋은 구조체
  - 프로그램=데이터+데이터 조작 루틴(함수)
  - 잘 구성된 프로그램은 데이터와 더불어 함수들도 부류를 형성
  - C++에서는 구조체는 클래스라는 넓은 개념의 의미로 존재한다.



구조체와 클래스

- 구조체가 아니라 클래스(Class)
  - 클래스 = 멤버 변수 + 멤버 함수
  - 변수가 아니라 객체(Object: 완전한 대상체)

- 사물의 관찰 이후의 데이터 추상화
  - 현실 세계의 사물을 데이터적인 측면과 기능적인 측면을 통해서 정의하는 것

클래스

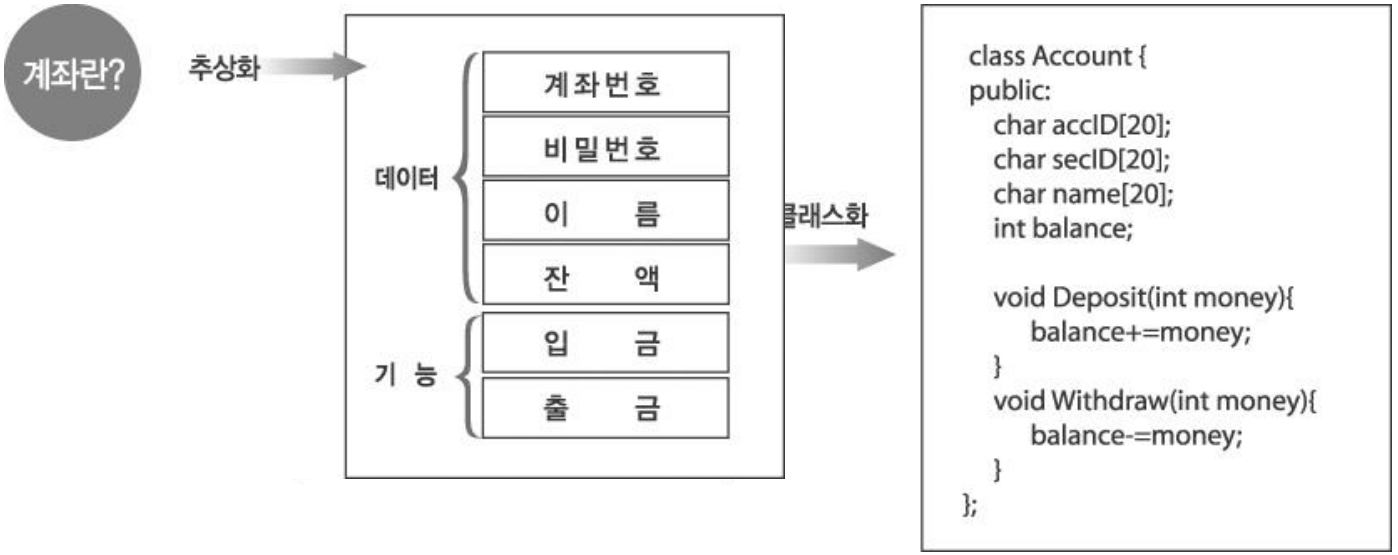
클래스 = 특성(Attribute) + 방법(Method)
멤버변수 + 멤버함수

클래스 : 추상화된 데이터(데이터, 함수)를 자료형으로 정의한 것을 말한다.
사용자 정의 자료형을 정의하는 것을 "클래스화 한다"라고 표현한다.
즉 새로운 자료형을 하나 정의 하는 것이다.

오브젝트(개체) : 클래스로 정의한 자료형으로 프로그램에서 사용할
변수(오브젝트)를 생성하는 것을 말한다.
클래스로 기반으로 오브젝트를 생성하는 것을
"인스턴스화 한다" 라고 표현한다.

클래스와 객체

현실적인사물 -> 데이터적인 측면과 기능적인 측면을 통해서 정의



C++클래스와 C++객체

- 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

C++ 클래스 만들기

- 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

C++ 클래스 만들기

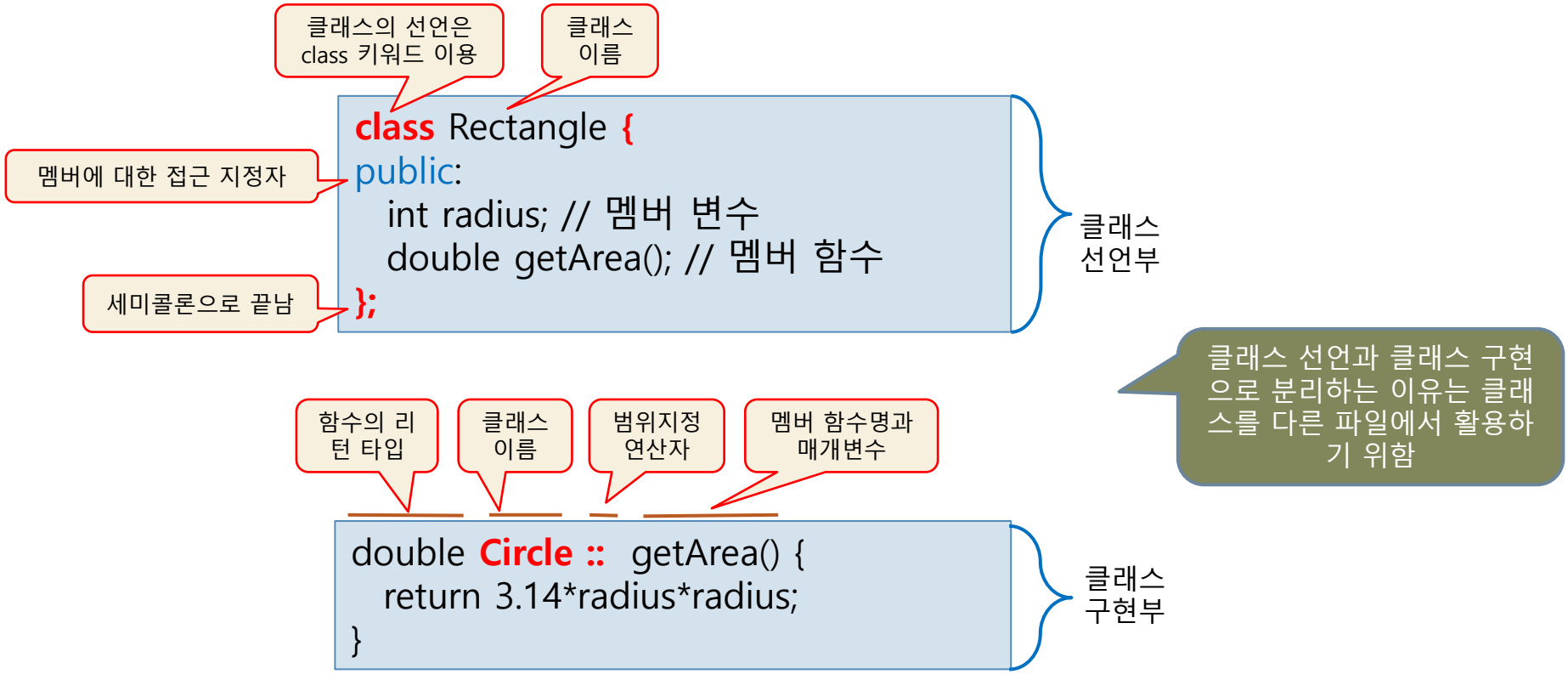
- **멤버함수의 외부 정의**

- 클래스 안에 멤버와 함수를 모두 기술하다보니 클래스의 구조가 한눈에 파악되지 않는다. 또한 복잡한 클래스인 경우 클래스의 부피가 너무 커지게 될 것이다.

- 따라서 C++에서는 멤버 함수를 클래스 밖에서 정의하는 방법을 제공하고 있다.

- 즉 클래스 선언과 정의를 분리한다.

클래스 만들기 설명


클래스의 선언은 class 키워드 이용
클래스 이름
멤버에 대한 접근 지정자
세미콜론으로 끝남
class Rectangle {
public:
int radius; // 멤버 변수
double getArea(); // 멤버 함수
};
클래스 선언부
클래스 선언과 클래스 구현으로 분리하는 이유는 클래스를 다른 파일에서 활용하기 위함
함수의 리턴 타입
클래스 이름
범위지정 연산자
멤버 함수명과 매개변수
double Circle :: getArea() {
return 3.14*radius*radius;
}
클래스 구현부

Circle 클래스의 객체 생성 및 활용


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

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

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

int main() {
    Circle donut;
    donut.radius = 1; // donut 객체의 반지름을 1로 설정
    double area = donut.getArea(); // donut 객체의 면적 알아내기
    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;
}
```

Circle 선언부

Circle 구현부

객체 donut 생성

donut의 멤버 변수 접근

donut의 멤버 함수 호출

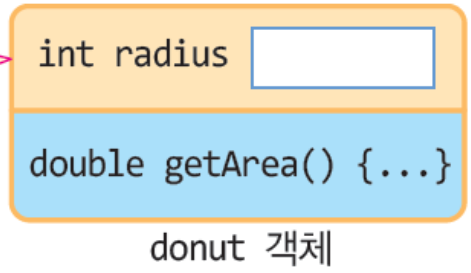
```
donut 면적은 3.14
pizza 면적은 2826
```

객체 이름과 생성, 접근 과정

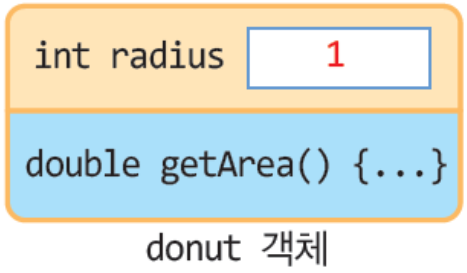

```
(1) Circle donut;
```

객체 이름

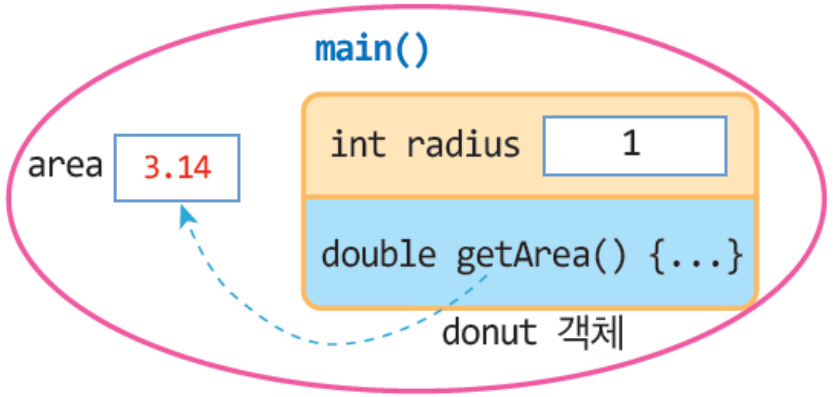
객체가 생성되면
메모리가 할당된다.



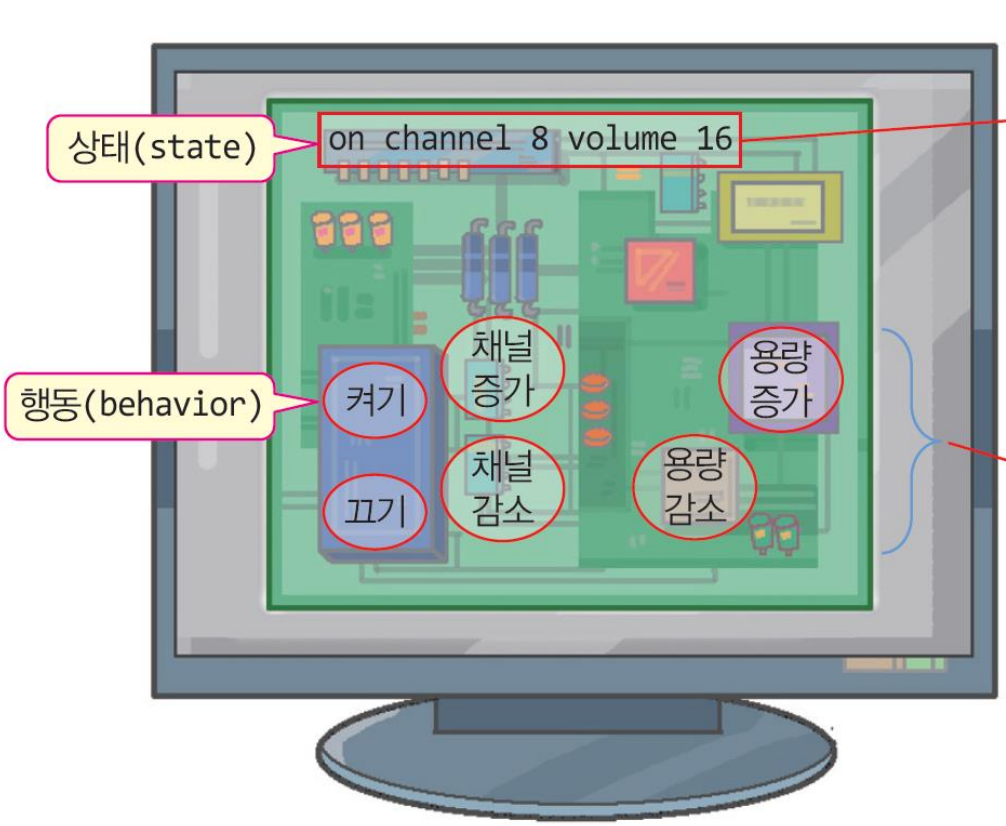

```
(2) donut.radius = 1;
```



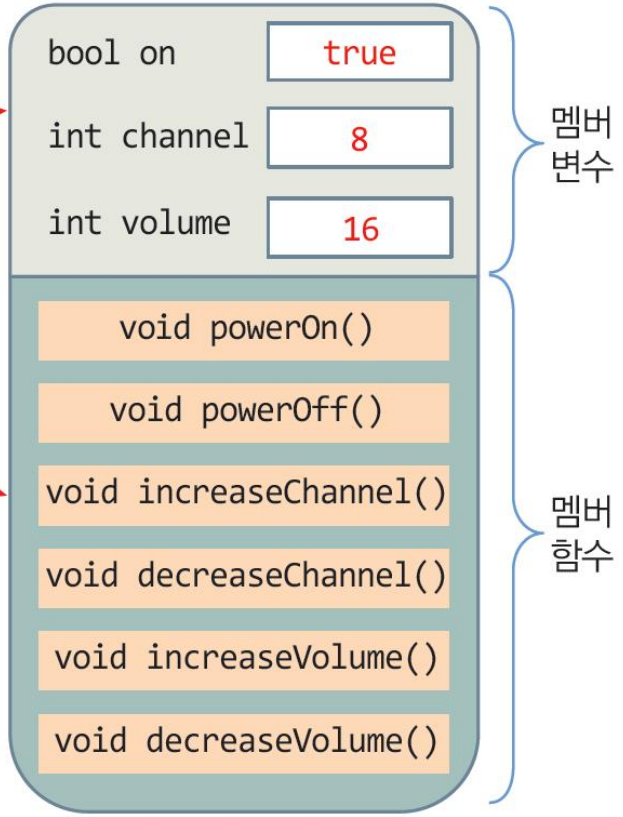

```
(3) double area = donut.getArea();
```



TV와 C++로 설계된 TV 객체



실세계의 TV 객체



C++로 설계한 TV 객체

클래스 멤버로의 접근

내부접근 : 클래스 내에 존재하는 멤버에 의한 접근이다.
외부접근 : 클래스 외에 존재하는 모든 접근이다. (메서드)

public : public 멤버가 선언되면 클래스 외부 접근도 허용한다.
즉 어디서든 접근을 허용하겠다는 의미이다.
private : private 멤버가 선언되면 클래스 내부 접근만 허용한다.
외부 접근을 시도하면 오류이다.

클래스 멤버의 접근 제어

```
class Counter {
public:
    int val;
    void Increment(void)
    {
        val++;                  //내부 접근
    }
};
int main(void)
{
   Counter cnt;
   cnt.val=0;                   //외부 접근
   cnt.Increment();             //외부 접근
   cout<<cnt.val<<endl;         //외부 접근
   return 0;
}
```

클래스 완성

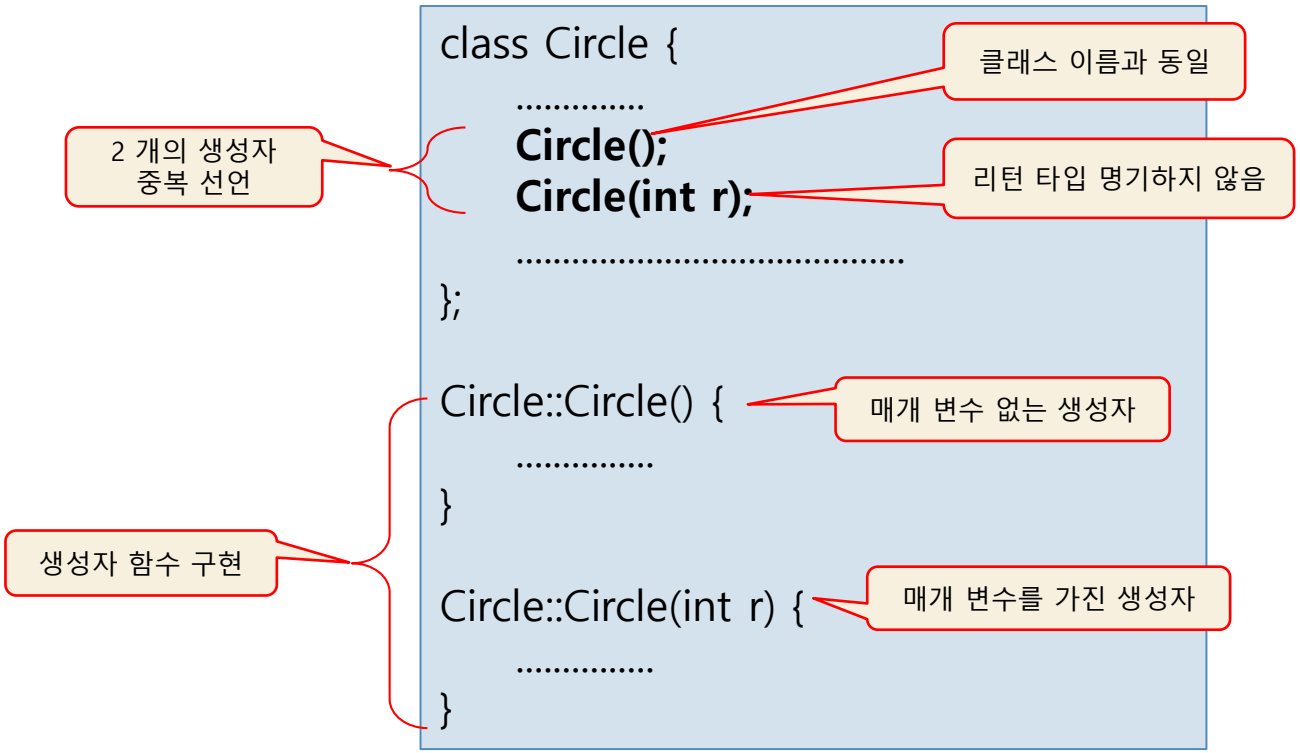
클래스는 두 가지 조건을 충족시켜야 좋은 클래스라 할 수 있다.

1. **정보 은닉(Information Hiding)** : 객체의 외부에서 객체 내에 존재하는 멤버 변수에 직접 접근하는 것을 허용하지 않는 것을 정보 은닉이라 한다. 데이터를 검사.

2. **캡슐화(Encapsulation)** : 관련이 있는 데이터와 함수를 하나의 단위로 묶는 것을 캡슐화라 한다. 즉 관련된 함수를 또 다른 클래스가 아닌 같은 클래스로 묶어서 처리해야 한다.

생성자

- 생성자(constructor)
  - 객체가 생성되는 시점에서 자동으로 호출되는 멤버 함수
  - 클래스 이름과 동일한 멤버 함수



생성자 함수의 특징

- 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

2 개의 생성자를 가진 Circle 클래스


```
#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

30

Circle(30); 자동 호출

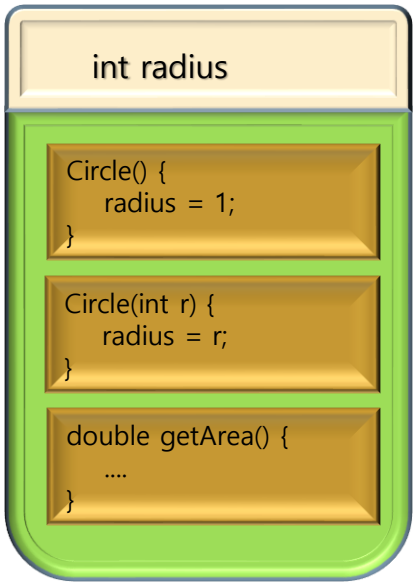
```
int main() {
    Circle donut; // 매개 변수 없는 생성자 호출
    double area = donut.getArea();
    cout << "donut 면적은 " << area << endl;

    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
```

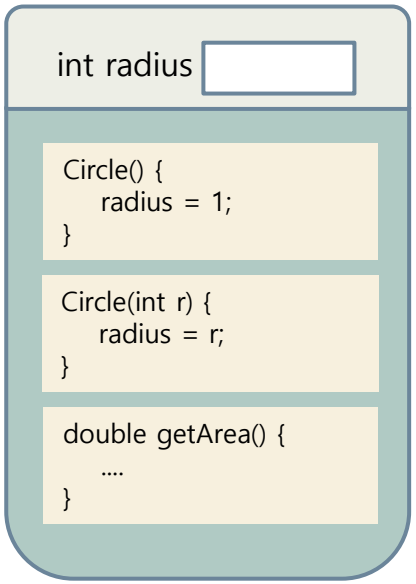
객체 생성 및 생성자 실행 과정

Circle donut;



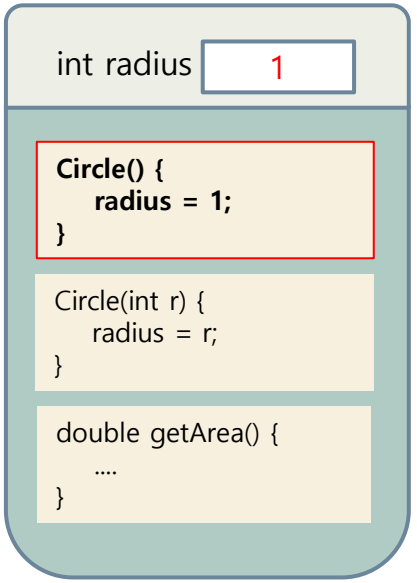
Circle 클래스

❶ 객체 공간 할당



donut 객체

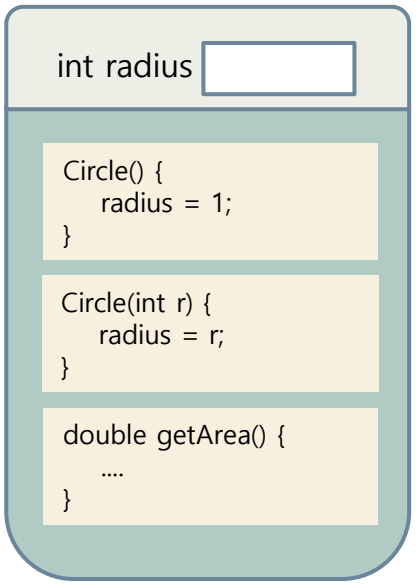
❷ 생성자 실행



donut 객체

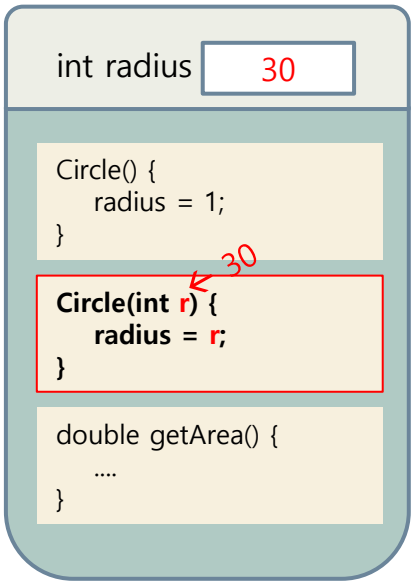
Circle pizza(30);

❶ 객체 공간 할당



pizza 객체

❷ 생성자 실행



pizza 객체

생성자가 다른 생성자 호출(위임 생성자)

- 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

타겟 생성자

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

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

호출

r에 1 전달

간소화된 코드

다양한 생성자의 멤버 변수 초기화 방법


```
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 콜론 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 선언부에서 직접 초기화, C++11에 추가된 문법

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

기본(Default) 생성자

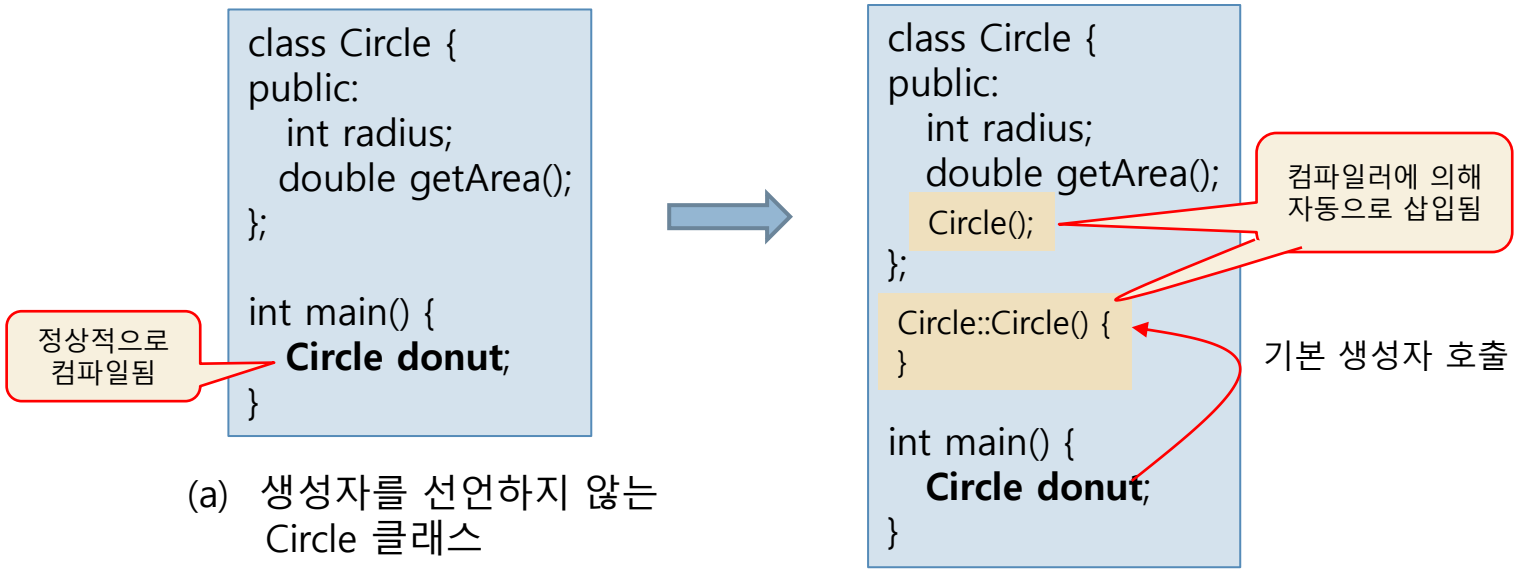
1. *생성자는 꼭 있어야 하는가?*
   - 예. C++ 컴파일러는 객체가 생성될 때, 생성자 반드시 호출
2. *개발자가 클래스에 생성자를 작성해 놓지 않으면?*
   - 컴파일러에 의해 기본 생성자가 자동으로 생성

- 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

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



(a) 생성자를 선언하지 않는 Circle 클래스

(b) 컴파일러에 의해 기본 생성자 자동 삽입

기본 생성자가 자동으로 생성되지 않는 경우

- 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;
}
```

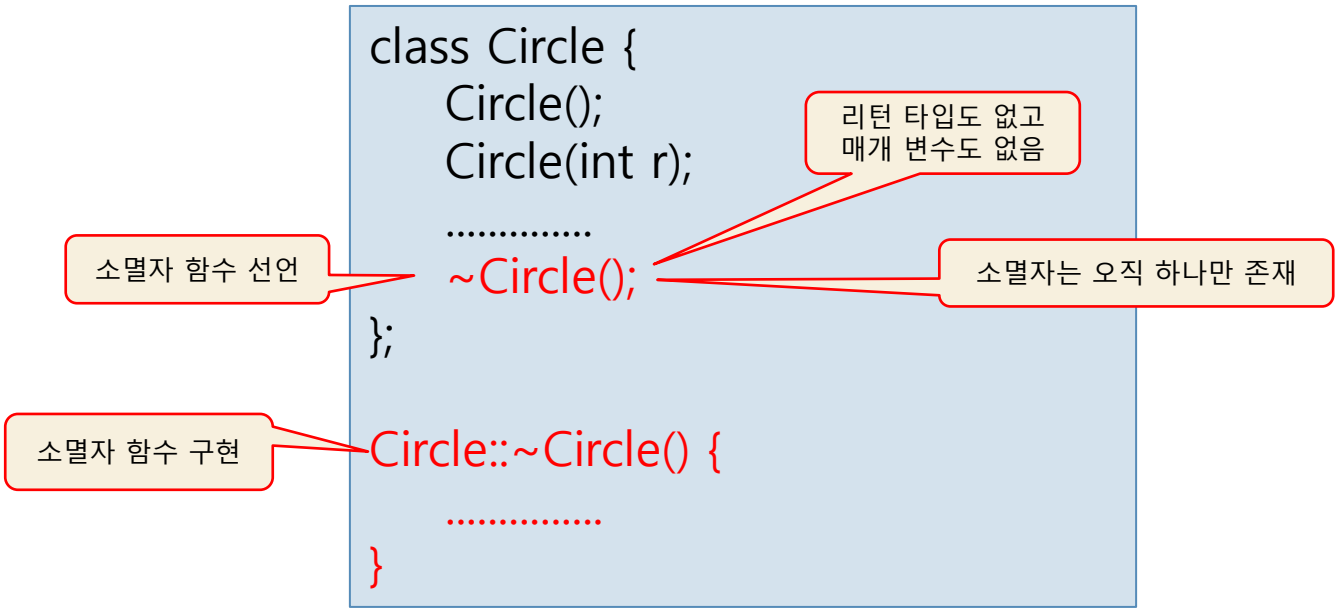
Circle 클래스에 생성자가 선언되어 있기 때문에, 컴파일러는 기본 생성자를 자동 생성하지 않음

호출

컴파일 오류.
기본 생성자 없음

소멸자

- 소멸자
  - 객체가 소멸되는 시점에서 자동으로 호출되는 함수
    - 오직 한번만 자동 호출, 임의로 호출할 수 없음
    - 객체 메모리 소멸 직전 호출됨



소멸자 특징

- 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

Circle 클래스에 소멸자 작성 및 실행


```
#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.

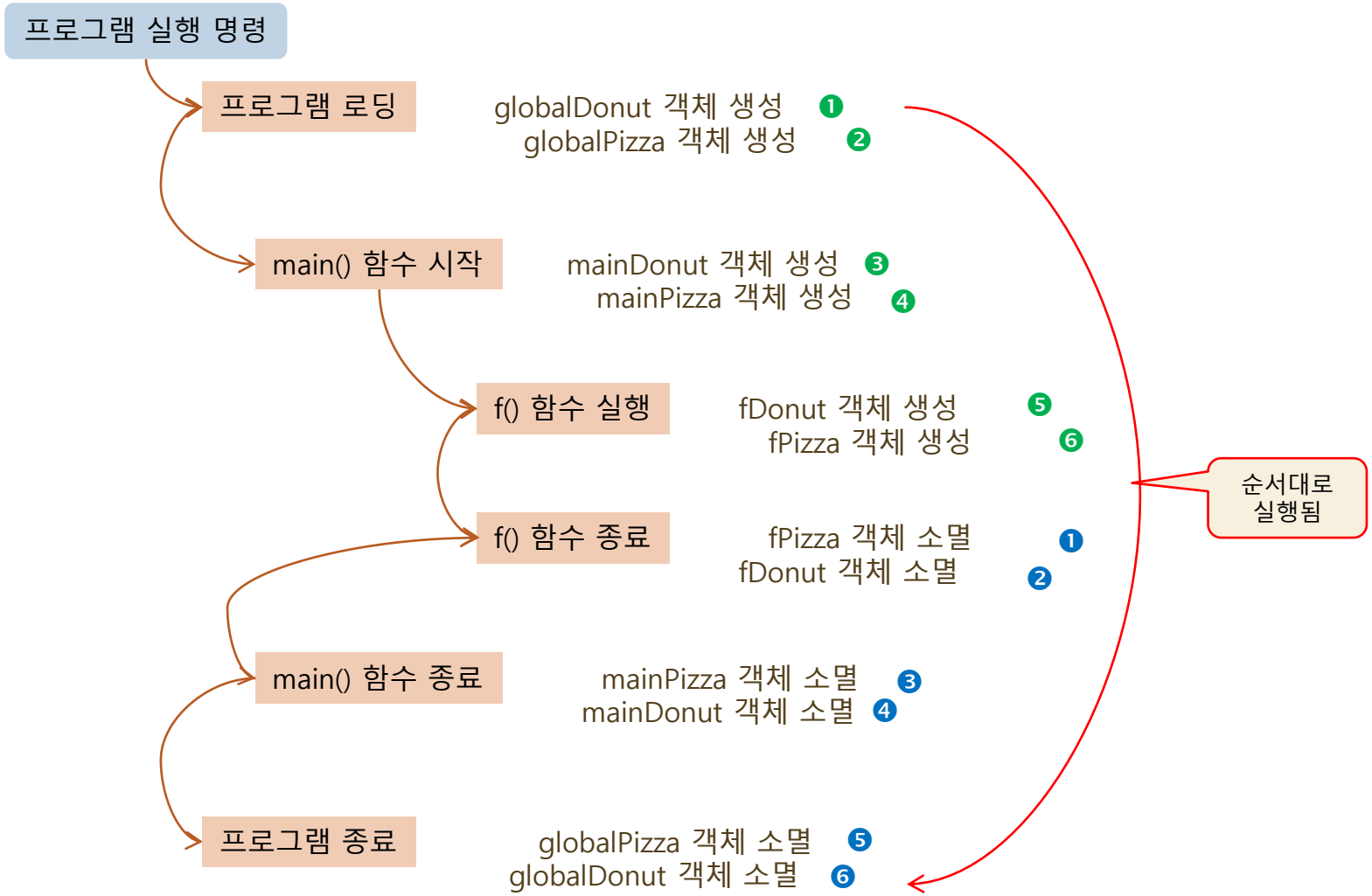
```
반지름 1 원 생성
반지름 30 원 생성
반지름 30 원 소멸
반지름 1 원 소멸
```

객체는 생성의 반대 순으로 소멸된다.

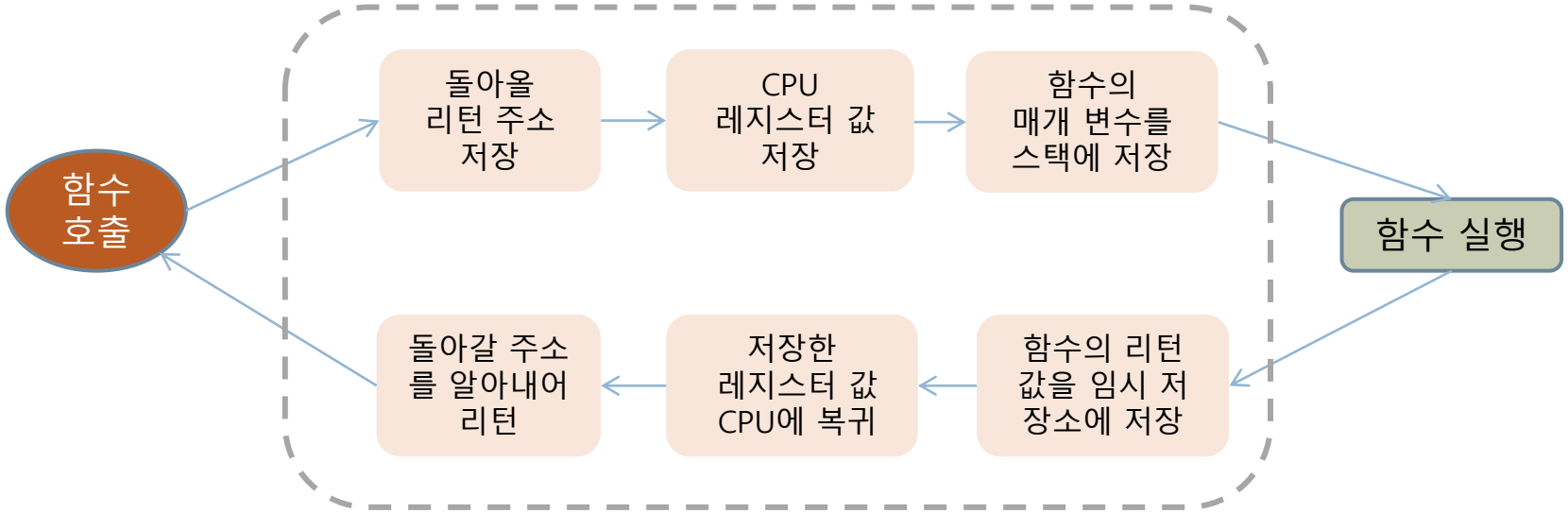
생성자/소멸자 실행 순서

- 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

지역 객체와 전역 객체의 생성과 소멸 과정



함수 호출에 따른 시간 오버헤드



함수 호출에 따른 시간 오버헤드

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

인라인 함수

- 인라인 함수

  - inline 키워드로 선언된 함수
  - 클래스 선언부에 바로 구현된 멤버 함수들에 대해서는 inline선언이 없어도 인라인 함수로 자동으로 처리한다.

  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 장단점 및 자동 인라인

- 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

분할 컴파일

- **헤더파일과 구현파일을 나누어 분할 컴파일** 한다.

- 헤더파일을 이용한 파일의 분할
  (프로그램을 관리 및 확장하기 좋아진다)

- Door.h : 클래스 선언은 헤더파일에 구현한다.
- Door.cpp : 클래스 멤버함수 정의는
  .cpp 파일에 구 현한다.
- Door_Main.cpp : 이 클래스를 필요로 하는 모든 파일에
  헤더파일을 삽입한다.

헤더 파일을 이용한 파일의 분할

Door.h

```
#include <iostream>
using std::cout;
using std::endl;

const int OPEN=1;
const int CLOSE=2;

class Door{
private:
   int state;

public:
   void Open();
   void Close();
   void ShowState();
};
```

Door.cpp

```
#include "Door.h"

void Door::Open(){
   state=OPEN;
}
void Door::Close(){
   state=CLOSE;
}
void Door::ShowState(){
   cout<<"현재 문의 상태 :";
   … 생략…
}
```

Main.cpp

```
#include "Door.h"

int main()
{
   Door d;

   d.Open();
   d.ShowState();

   return 0;
}
```

인라인 함수

- 함수를 인라인으로 만들면 실제로 함수가 호출되는 대신에, 함수를 호출한 위치에 함수의 내용이 그대로 옮겨진다.
- C 언어의 매크로함수와 유사하지만 함수의 특성을 그대로 사용하는 장점이 있다.

- 크기가 작은 함수만 인라인으로 만드는 것이 좋다.
  - 크기가 큰 함수를 인라인으로 만들면 실행 파일의 크기가 커지기 때문에 실행 성능이 낮아질 수 있다.

인라인 함수를 만드는 법

- 멤버 함수를 인라인으로 만드는 방법에는 두 가지가 있다.
  - 클래스의 내부에 정의한 멤버 함수들은 자동으로 인라인 함수가 된다.
- 클래스의 외부에 정의한 멤버 함수는 함수의 정의 앞에 inline 키워드를 추가한다.

- 클래스의 헤더와 구현부분이 다를 경우 인라인 함수는 헤더 파일에 위치해야 한다.

- 이유는 main() 에서 Open 메서드 호출 시 함수부분으로 실행코드가 대치가 일어나는데 이때 Door.cpp를 참조할 수 없다. 이유는 컴파일을 소스마다 각각 일어나기 때문이다.

  따라서 인라인 함수는 헤더파일에 주어 참조하게 한다.

헤더 파일의 중복 include 문제

- 헤더 파일을 중복 include 할 때 생기는 문제

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

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

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

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

헤더 파일의 중복 include 문제를 조건 컴파일로 해결

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

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

```
#ifndef Door _H
#define Door _H

void Door::Close()
{
state=CLOSE;
}

void Door::ShowState()
{
    cout<<"현재 문의 상태 :
";
cout<<((state==OPEN)?
"OPEN" : "CLOSE")<<endl;
}
#endif
```

Door.h

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

#include "Door.h"
#include "Door h"
#include "Door.h"

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

컴파일 오류 없음

main.cpp

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