# C++ Programming

7th Study: Object-Oriented Programming (3/8)


- Copy constructor
- Copy assignment operator
- Shallow copy / deep copy
- Rule of zero / rule of three

C++ Korea 옥찬호 (utilForever@gmail.com)

# Copy constructor

A constructor called whenever an object is initialized from another object of same type

# Copy constructor

- 다음과 같은 코드를 고려해 보면…
  - class Person { … }
    Person p1;
    Person p2(p1);
  - “p2라는 객체를 p1과 똑같이 만들어주세요!”
    → 이 때 호출되는 생성자가 바로 복사 생성자(Copy Constructor)!
  - 복사 생성자는 이전에 배웠던 생성자와 동일하되,
    매개변수로 객체가 들어온다는 차이가 있음
    - 예를 들어, Person(const Person& src) { … }
    - const와 &를 붙이는 이유? : 객체 복사로 인한 오버로드를 피하고, 안전성을 높임

# Copy constructor

- 하지만 만든 적도 없는데 코드는 잘 실행된다!?
  - 생성자와 마찬가지로, 프로그래머가 명시적으로 복사 생성자를 만들지 않으면 컴파일러가 자동으로 만들어 줌
- 복제 생성자를 구현하는 방법 : 멤버 변수들을 전부 복사!
  - 멤버 이니셜라이저를 이용하는 방법
    - Person(const Person& src) : height(src.height), weight(src.weight) { }
  - 생성자 본문을 이용하는 방법
    - Person(const Person& src) { height = src.height; weight = src.weight; }
- 대부분 복사 생성자를 직접 만들 필요는 없음 (하지만…)

# Copy constructor: Example

```
// 1
Person(const Person& src)
    : height(src.height), weight(src.weight) { }
// 2
Person(const Person& src)
{
    height = src.height;
    weight = src.height;
}


int main()
{
    Person p1(183.4, 78.5);
    Person p2(p1);
}
```

# Copy assignment operator

A non-static member function called whenever selected by overload resolution

# Copy assignment operator

- 다음과 같은 코드를 고려해 보면…
  - class Person { … }
    Person p1, p2;
    p1 = p2;
  - "p2라는 객체를 p1에 대입!" → 이 때 호출되는 함수가 바로 복사 대입 연산자(Copy Assignment Operator)!
  - 복사 대입 연산자는 = 연산자를 각 클래스에서 오버로딩해서 구현함
    - 예를 들어, Person& operator=(const Person& rhs);
  - 복사 대입 연산자도 프로그래머가 명시적으로 만들지 않으면 객체 간 대입이 가능하도록 컴파일러가 자동으로 만들어 줌

# Copy assignment operator

- Person& operator=(const Person& rhs);
  - 복사 생성자와 달리 복사 대입 연산자는 Person의 참조 객체를 리턴
  - 그 이유는 대입 연산이 중첩되어서 수행될 수 있기 때문
  - 예를 들어, p1 = p2 = p3; 라는 문장이 있다고 가정하면…
    - 먼저 p2의 대입 연산자가 p3을 우변 항목 인자로 호출
    - 그 다음, p1의 대입 연산자가 호출되는데 이 때 우변 항목 인자는 p2가 아님
    - p1의 대입 연산자는 p2의 대입 연산자가 p3를 인자로 해 실행된 리턴값을 우변 항목 인자로 취함
    - 만약 대입 연산이 실패해서 리턴값이 없으면 p1으로 전달할 인자가 없게 됨

# Copy assignment operator

- Person& operator=(const Person& rhs);
  - p1의 대입 연산자가 그냥 p2를 인자로 취해도 문제가 없지 않을까?
  - 문제는 = 기호가 멤버 함수 호출을 래핑(Wrapping)만 한다는 점
    - p1 = p2 = p3;를 다시 쓰면 p1.operator=(p2.operator=(p3));
    - p1.operator=이 실행되려면 p2.operator=가 반드시 리턴값을 줘야만 함
    - 리턴값이 있어야만 멤버 함수 operator=를 실행할 인자가 존재함
    - 맥락상 p2.operator=의 올바른 리턴값은 p2 그 자체가 되어야 하고,
      객체 리턴에 따른 임시 객체로의 복제 오버로드를 피하려면
      참조형으로 리턴되는 것이 바람직함

# Copy assignment operator

- 복사 대입 연산자의 구현 방법
  - 복사 생성자의 구현 방법과 비슷하지만 몇 가지 중요한 차이점이 존재
    - 복사 생성자는 객체 초기화 시점에만 호출되기 때문에 대상 객체들의 멤버들이 아직 유효하지 않음
    - 복사 대입 연산자는 이미 생성된 객체를 대상으로 하기 때문에 멤버들의 메모리 할당 완료 여부에 신경쓰지 않고도 값을 덮어 쓸 수 있음
  - C++에서는 객체가 자기 스스로를 대입하는 것이 문법적으로 가능
    - 예를 들어, 다음 코드는 컴파일도 되고 실행도 잘됨
      Person p(183.4, 78.5); p = p;
    - 복사 대입 연산자를 구현할 때 이 경우를 배제해서는 안 됨!
      (자기 스스로를 대입하는 경우 문제가 발생할 수도 있음)

# Copy assignment operator

- 복사 대입 연산자의 구현 방법
  - 복사 대입 연산자가 실행되면 가장 먼저 인자가 자기 자신인지 검사하고, 그렇다면 복제 작업을 하지 않고 그대로 리턴하게 만듬
  - if (this == &rhs) { return *this };
    - this : 객체의 포인터 (주소값을 가지고 있음), &rhs : 우변 객체의 주소값
    - 따라서 this와 &rhs를 비교해서 그 값이 같다면 (즉, 같은 주소 값을 가진 객체라면), 두 객체가 같은 대상으로 가리키고 있으므로 자기 자신을 대입하고 있다는 것!
    - 복사 대입 연산자는 연산 결과로 자기 자신을 리턴해야 함 → return *this;
    - this 포인터는 연산자가 실행될 객체를 가리키기 때문에 *this는 객체 자신이 됨
  - 자기 자신이 아니라면, 복사 생성자와 같이 멤버에 대한 대입을 수행

# Copy assignment operator: Example

```
Person& operator=(const Person& rhs)
{
    if (this == &rhs)
        return *this;

    height = rhs.height;
    weight = rhs.weight;

    return *this;
}


int main()
{
    Person p1(183.4, 78.5), p2(175.6, 68.3);
    p1 = p2;
}
```

# Shallow copy / deep copy

Duplicate as little as possible / duplicate everything

# Shallow copy / deep copy

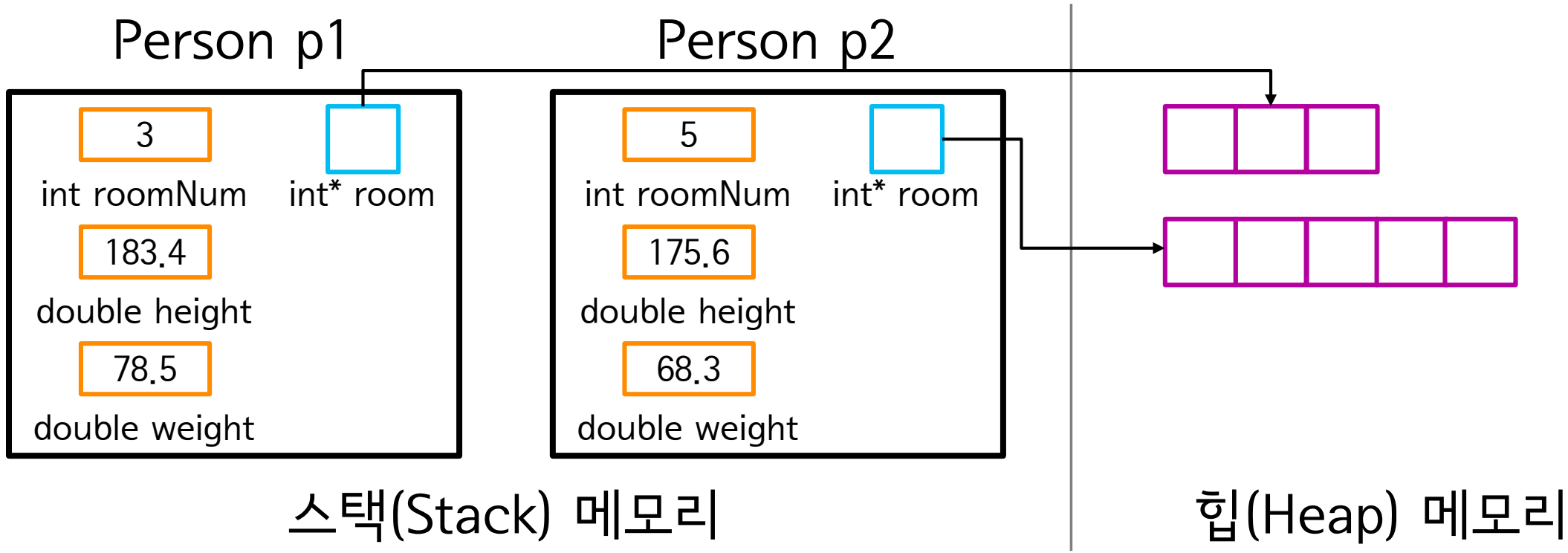
- 앞서 배운 복사 생성자, 복사 할당 연산자의 경우 대부분 직접 만들 필요가 없음 (컴파일러가 자동으로…)
- 컴파일러가 자동으로 생성한 멤버 함수들은 멤버 변수들에 대해 재귀적으로 복사 생성자 또는 복사 대입 연산자를 호출
- 단 int, double, 포인터와 같이 기본 데이터 타입에 대해서는 복사 생성자나 복사 대입 연산자 대신에 얕은 복사가 일어남
  - 얕은 복사(Shallow Copy)란 포인터가 가리키는 데이터는 빼놓고 주소 값만 복사하는 방식을 말함

# Shallow copy / deep copy

- 하지만, 얕은 복사는 객체가 동적으로 할당받은 메모리를 가지고 있을 경우 문제가 됨
  - 예를 들어, Person 클래스에서 사람이 가지고 있는 방의 개수(int roomNum;)를 받아 각 방의 면적을 나타내는 배열(int* room;)을 동적으로 생성한다고 가정
    - Person(int _roomNum, double _height, double _weight)
      : roomNum(_roomNum), height(_height), weight(_weight)
      {
      room = new int[roomNum];
      }
    - ~Person() { delete[] room; room = nullptr; }

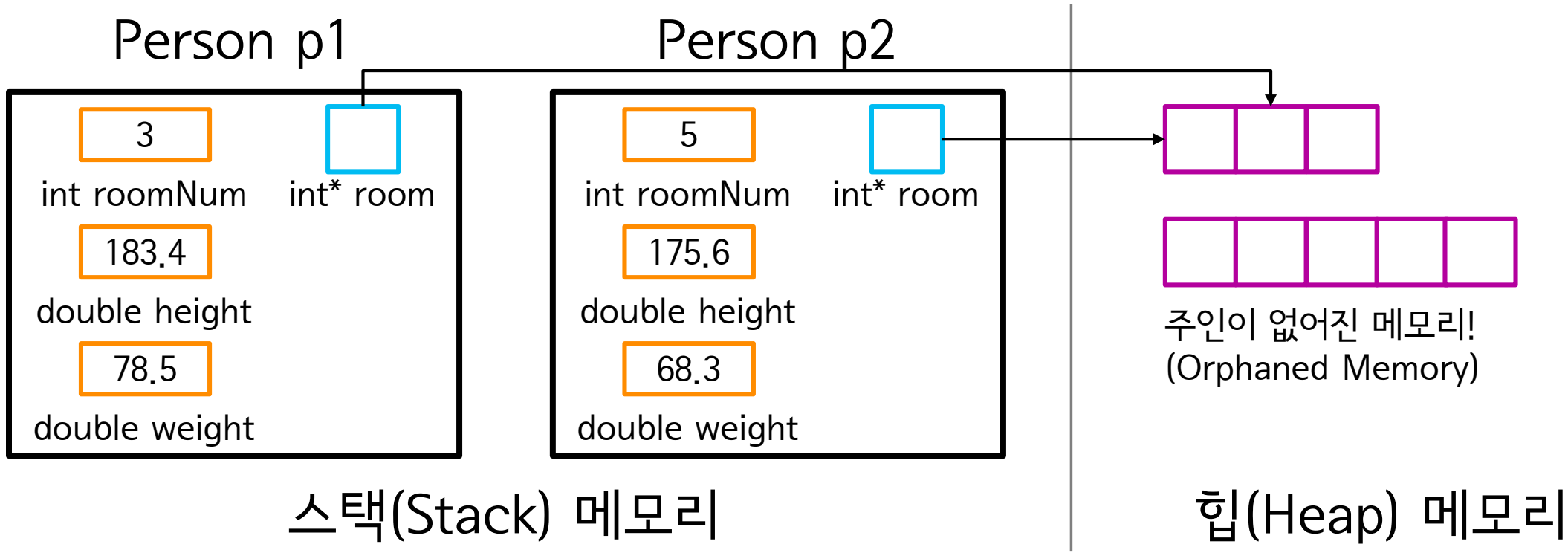
# Shallow copy / deep copy

- 이후 main 함수에서 다음 코드를 실행했다고 생각하면…
  - Person p1(3, 183.4, 78.5), p2(5, 175.6, 68.3);



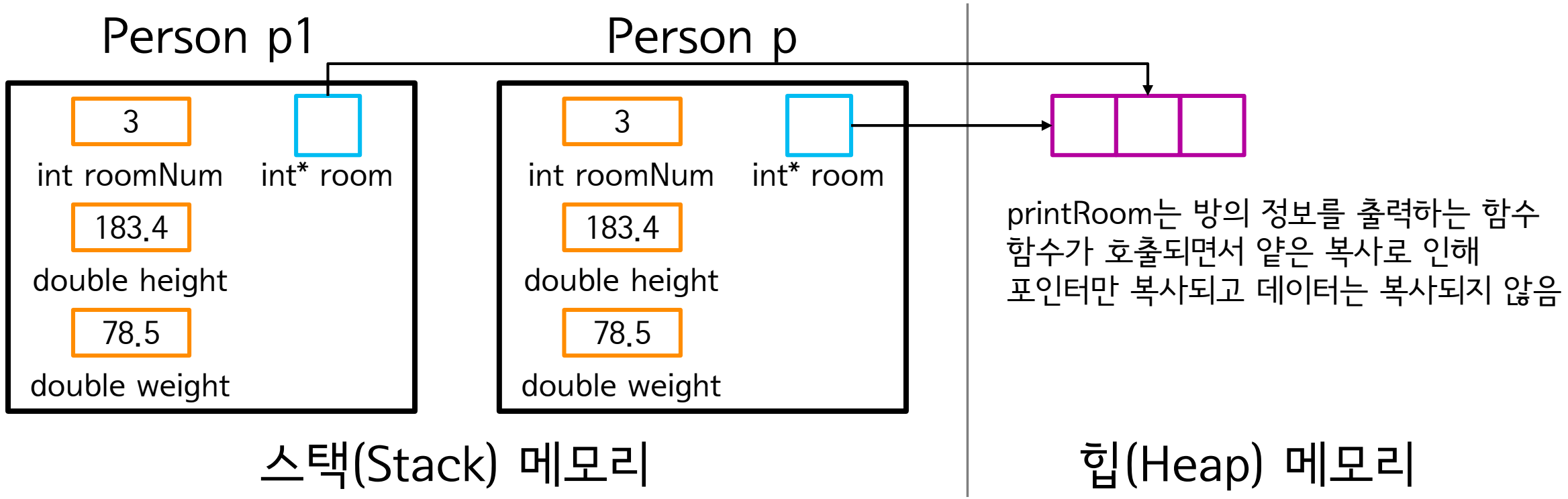
# Shallow copy / deep copy

- 이후 main 함수에서 다음 코드를 실행했다고 생각하면…
  - p1 = p2;



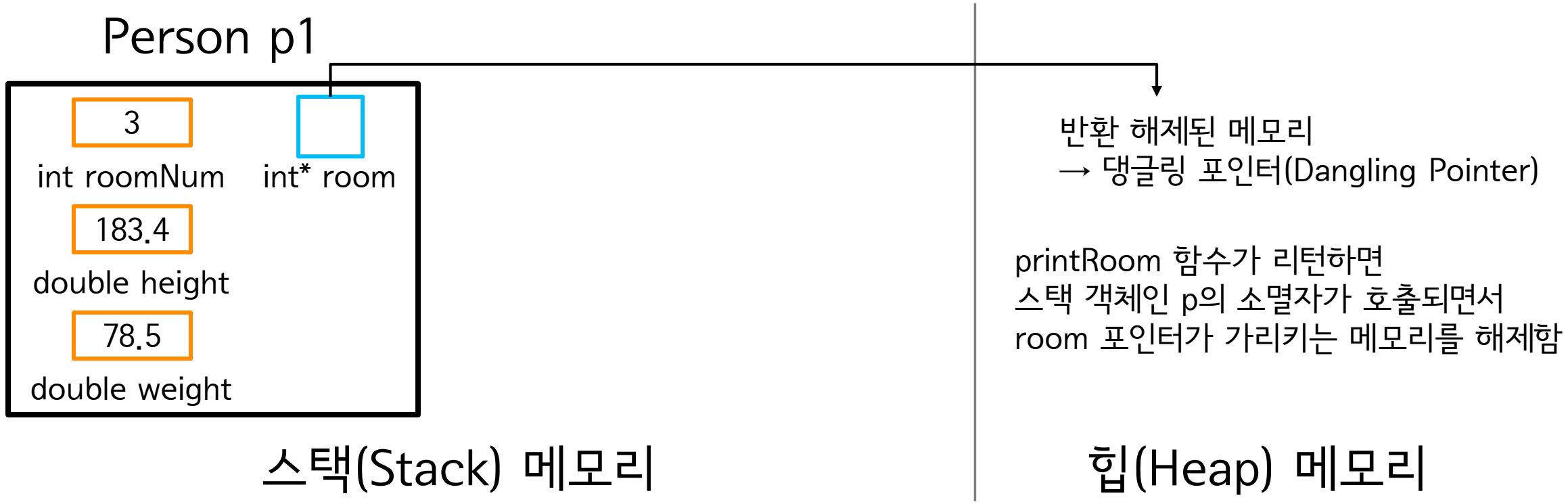
# Shallow copy / deep copy

- 또 다른 경우 : void printRoom(Person p);가 호출될 때
  - Person p1(3, 183.4, 78.5);
    printRoom(p1);



# Shallow copy / deep copy

- 또 다른 경우 : void printRoom(Person p); 호출이 끝날 때
  - Person p1(3, 183.4, 78.5);
    printRoom(p1);



# Shallow copy / deep copy

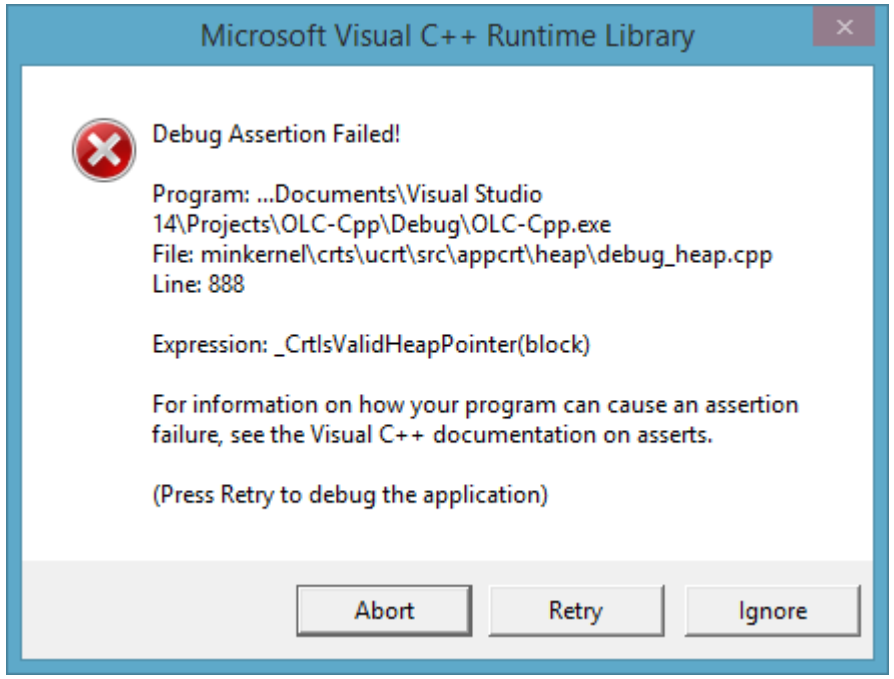
- 이러한 문제를 피하기 위해서는 대입 대상인 좌변항의 객체가 참조하고 있는 메모리를 반환한 후에 새로 메모리를 준비해 깊은 복사를 해야 함
  - 깊은 복사(Deep Copy)란 포인터만 복사하지 않고 변수의 맥락에 맞게 연관된 데이터까지 재귀적으로 복사하는 방식을 말함
    - 복사 생성자의 경우
      - room = new int[roomNum];
      - for (int i = 0; i < roomNum; ++i) { room[i] = src.room[i]; }
    - 복사 할당 연산자의 경우
      - delete[] room; room = nullptr;
      - room = new int[roomNum]; for (int i = 0; i < roomNum; ++i) { room[i] = src.room[i]; }

# Shallow copy / deep copy: Example

```
class Person {
private:
    int* room;
    int roomNum;
    double height;
    double weight;
public:
    Person() { }
    Person(int _roomNum, double _height, double _weight)
        : roomNum(_roomNum), height(_height), weight(_weight) {
        room = new int[roomNum];
    }
    ~Person() {
        delete[] room;
    }
};
```

# Shallow copy / deep copy: Example

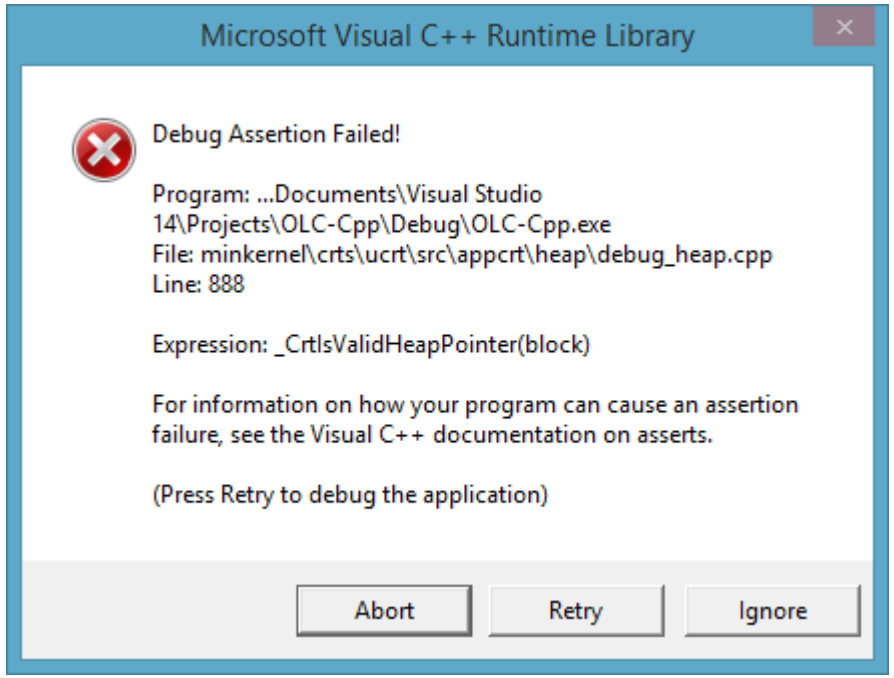
```
int main()
{
    Person p1(3, 183.4, 78.5);
    Person p2(5, 175.6, 68.3);
    p1 = p2;
}
```



# Shallow copy / deep copy: Example

```
void printRoom(Person p) { }

int main()
{
    Person p1(3, 183.4, 78.5);
    printRoom(p1);
}
```



# Shallow copy / deep copy: Example

```
Person(const Person& src)
    : roomNum(src.roomNum), height(src.height), weight(src.weight)
{
    room = new int[roomNum];
    for (int i = 0; i < roomNum; ++i)
        room[i] = src.room[i];
}
```

# Shallow copy / deep copy: Example

```
Person& operator=(const Person& rhs) {
    if (this == &rhs)
        return *this;
    delete[] room;
    room = nullptr;

    height = rhs.height;
    weight = rhs.weight;
    roomNum = rhs.roomNum;

    room = new int[roomNum];
    for (int i = 0; i < roomNum; ++i)
        room[i] = rhs.room[i];

    return *this;
}
```

# Rule of zero / rule of three

Rules of thumb in C++ for building of exception-safe code and for formalizing rules on resource management

# Rule of zero / rule of three

- 0의 법칙(Rule of Zero)
  - 소멸자, 복사 생성자, 복사 할당 연산자 모두를 명시적으로 만들지 않으면 컴파일러가 모두 자동으로 만들어준다는 법칙
- 3의 법칙(Rule of Three)
  - 소멸자, 복사 생성자, 복사 할당 연산자 중 하나를 명시적으로 만들었다면, 나머지 모두 명시적으로 만들어야한다는 법칙
- 자세한 내용은 다음 링크를 참조
  - https://github.com/utilForever/ModernCpp/blob/master/ModernCpp/Classes/ruleOfZero.cpp
  - https://github.com/utilForever/ModernCpp/blob/master/ModernCpp/Classes/ruleOfFive.cpp