07. 여러가지 객체의 생성방법

7.1 객체배열과 객체포인터

7.2 동적메모리 할당 및 반환

7.3 객체 및 객체배열의 동적 생성 및 반환

7.4 멤버함수의 this 포인터

7.1 객체배열과 객체포인터

객체 배열 및 포인터의 생성 및 사용

- 객체 배열 선언 가능
  - 기본 타입 배열 선언과 형식 동일
  - 객체 배열 정의 시 따로 지정하지 않으면 항상 **디폴트 생성자** 로 초기화

```
Circle circleArray[3];  // 디폴트 생성자로 초기화
```

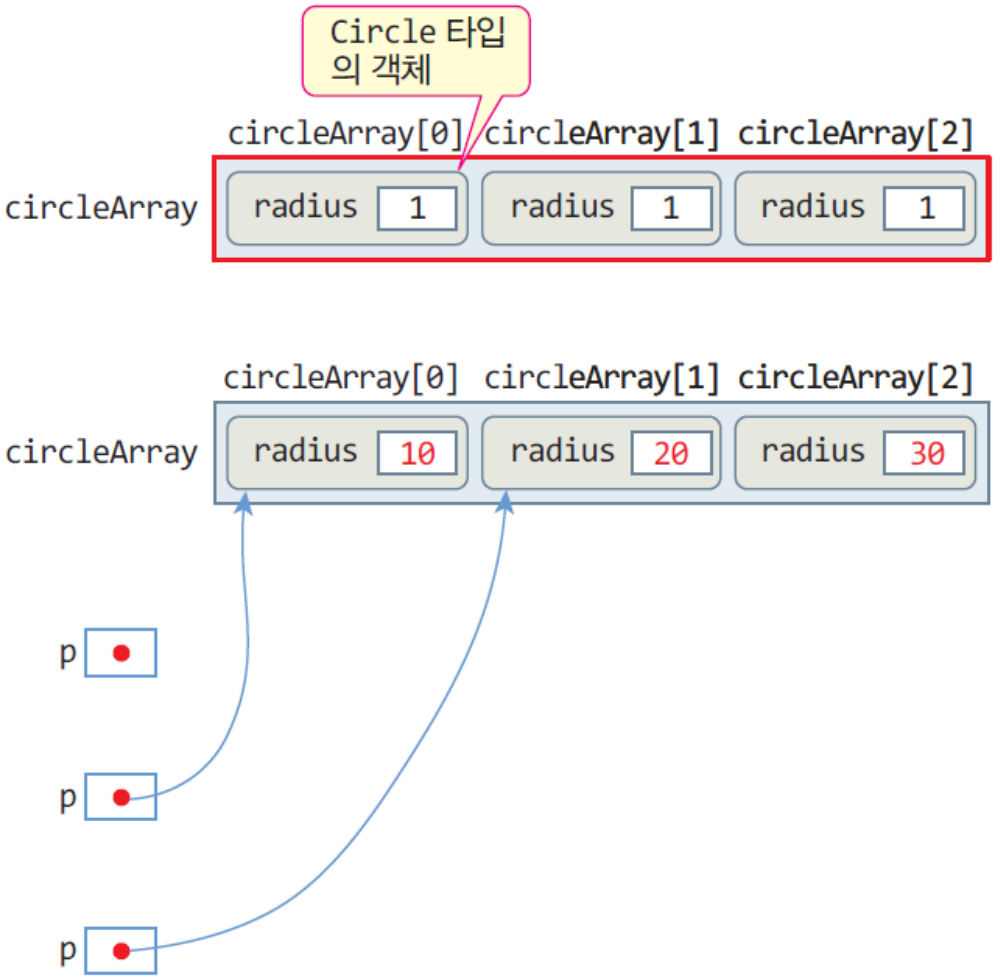
- 객체 배열의 초기화
  - 배열의 각 원소 객체당 생성자 지정하는 방법
    - { } 안에 생성자 나열

```
Point circleArray[3] = {Circle(10), Circle(20), Circle( )};
```

    - circleArray[0] 객체가 생성될 때, 생성자 Circle(10) 호출
    - circleArray[1] 객체가 생성될 때, 생성자 Circle(20) 호출
    - circleArray[2] 객체가 생성될 때, 생성자 Circle() 호출

객체 배열 및 포인터 생성과 활용

```
(1) Circle circleArray[3];

(2) circleArray[0].setRadius(10);
    circleArray[1].setRadius(20);
    circleArray[2].setRadius(30);

(3) Circle *p;

(4) p = circleArray;

(5) p++;
```



Circle 클래스의 배열 선언 및 활용

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

class Circle {
    int radius;
public:
    Circle() {   radius = 1; }
    Circle(int r)  { radius = r; }
    void setRadius(int r)  { radius = r; }
    double getArea();
};

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

```
int main() {
    Circle circleArray[3];                    // (1) Circle 객체 배열 생성

    // 배열의 각 원소 객체의 멤버 접근
    circleArray[0].setRadius(10);             // (2)
    circleArray[1].setRadius(20);
    circleArray[2].setRadius(30);

    for(int i=0; i<3; i++) // 배열의 각 원소 객체의 멤버 접근
        cout << "Circle " << i << "의 면적은 " << circleArray[i].getArea() <<
        endl;

    Circle *p;                                // (3)
    p = circleArray;                          // (4)
    for(int i=0; i<3; i++) { // 객체 포인터로 배열 접근
        cout << "Circle " << i << "의 면적은 " << p→getArea() << endl;
        p++;                                  // (5)
    }
}
```

```
Circle 0의 면적은 314
Circle 1의 면적은 1256
Circle 2의 면적은 2826
Circle 0의 면적은 314
Circle 1의 면적은 1256
Circle 2의 면적은 2826
```

객체 배열 생성시 디폴트 생성자 호출

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

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

int main() {
   Circle circleArray[3];
}
```

컴파일러가 자동으로 디폴트 생성자 **Circle() { }** 삽입. 컴파일 오류가 발생하지 않음

디폴트 생성자 Circle() 호출

(a) 생성자가 선언되어 있지 않은 Circle 클래스

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

class Circle {
   int radius;
public:
   Circle(int r)  { radius = r; }
   double getArea()  {
      return 3.14*radius*radius;
   }
};

int main() {
   Circle waffle(15);
   Circle circleArray[3];
}
```

Circle(int r) 호출

디폴트 생성자 Circle() 호출. 디폴트 생성자가 없으므로 컴파일 오류

error.cpp(15): error C2512: 'Circle' : 사용할 수 있는 적절한 디폴트 생성자가 없습니다

(b) 디폴트 생성자가 없으므로 컴파일 오류

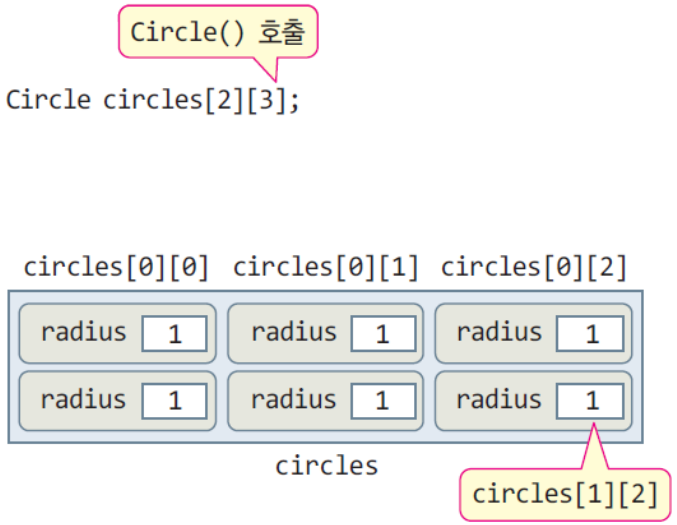
객체 배열의 인자있는 생성자로 초기화

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

class Circle {
    int radius;
public:
    Circle() {radius = 1; }
    Circle(int r)  { radius = r; }
    void setRadius(int r)  { radius = r; }
    double getArea();
};

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

int main() {
    Circle circleArray[3] = { Circle(10), Circle(20), Circle() }; // Circle 배열 초기화

    for(int i=0; i<3; i++)
        cout << "Circle " << i << "의 면적은 " << circleArray[i].getArea() << endl;
}
```

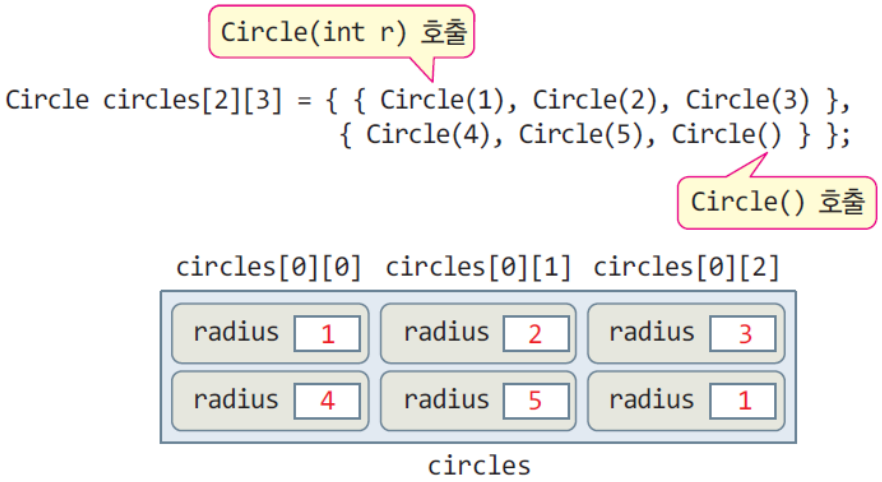
circleArray[0] 객체가 생성될 때, 생성자 Circle(10),
circleArray[1] 객체가 생성될 때, 생성자 Circle(20),
circleArray[2] 객체가 생성될 때, 기본 생성자
Circle()이 호출된다.

```
Circle 0의 면적은 314
Circle 1의 면적은 1256
Circle 2의 면적은 3.14
```

2차원 객체 배열



(a) 2차원 배열 선언 시



(b) 2차원 배열 선언과 초기화

```
circles[0][0].setRadius(1);
circles[0][1].setRadius(2);
circles[0][2].setRadius(3);
circles[1][0].setRadius(4);
circles[1][1].setRadius(5);
circles[1][2].setRadius(6);
```

2차원 배열을 초기화하는 다른 방식

Circle 클래스의 2차원 배열 선언 및 활용

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

class Circle {
    int radius;
public:
    Circle() {   radius = 1; }
    Circle(int r)  { radius = r; }
    void setRadius(int r)  { radius = r; }
    double getArea();
};

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

```
int main() {
    Circle circles[2][3];

    circles[0][0].setRadius(1);
    circles[0][1].setRadius(2);
    circles[0][2].setRadius(3);
    circles[1][0].setRadius(4);
    circles[1][1].setRadius(5);
    circles[1][2].setRadius(6);

    for(int i=0; i<2; i++)    // 배열의 각 원소 객체의 멤버 접근
        for(int j=0; j<3; j++) {
            cout << "Circle [" << i << "," << j  << "]의 면적은 ";
            cout << circles[i][j].getArea() <<endl;
        }
}
```

```
Circle circles[2][3] =
    { { Circle(1), Circle(2), Circle(3) },
      { Circle(4), Circle(5), Circle() } };
```

```
Circle [0,0]의 면적은 3.14
Circle [0,1]의 면적은 12.56
Circle [0,2]의 면적은 28.26
Circle [1,0]의 면적은 50.24
Circle [1,1]의 면적은 78.5
Circle [1,2]의 면적은 113.04
```

객체 포인터

- 객체에 대한 포인터
  - C 언어의 포인터와 동일
  - 객체의 주소 값을 가지는 변수

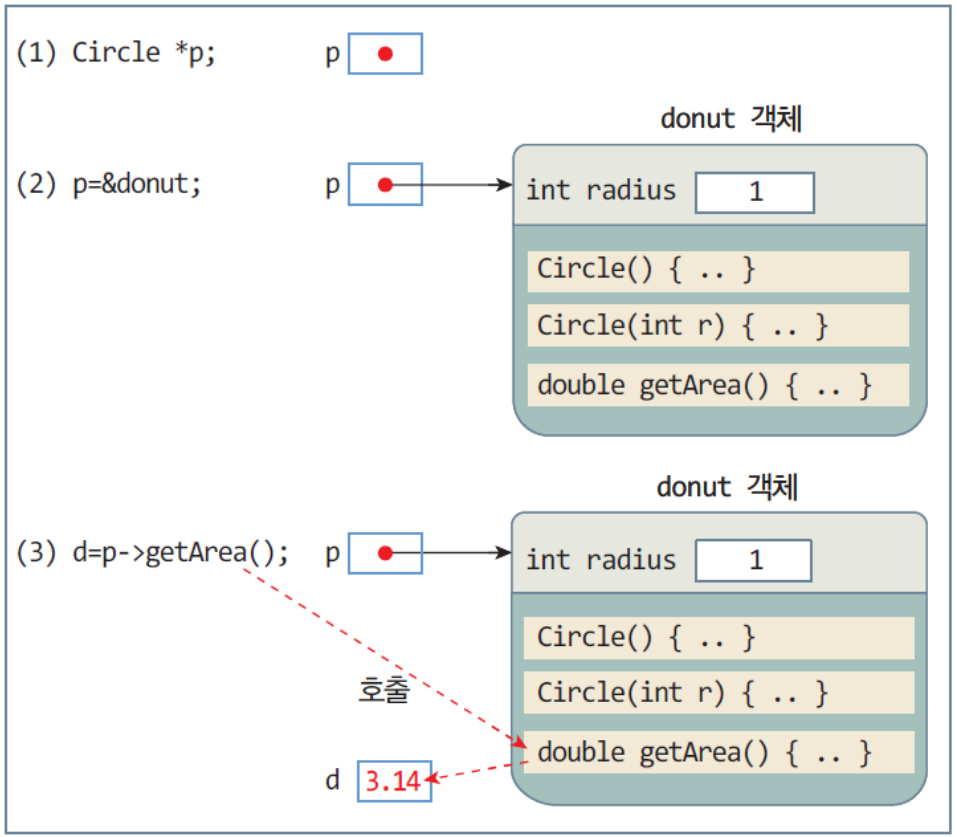
- 포인터로 멤버를 접근할 때
  - 객체포인터→멤버

객체에 대한 포인터 선언

포인터에 객체 주소 저장

멤버 함수 호출

```
Circle donut;
double d = donut.getArea();

Circle *p; // (1)
p = &donut; // (2)
d = p->getArea(); // (3)
```



객체 포인터 선언 및 사용

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

class Circle {
   int radius;
public:
   Circle() {radius = 1; }
   Circle(int r)  { radius = r; }
   double getArea();
};

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

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

   // 객체 이름으로 멤버 접근
   cout << donut.getArea() << endl;

   // 객체 포인터로 멤버 접근
   Circle *p;
   p = &donut;
   cout << p→getArea() << endl; // donut의 getArea() 호출
   cout << (*p).getArea() <<endl; // donut의 getArea() 호출

   p = &pizza;
   cout << p→getArea() << endl; // pizza의 getArea() 호출
   cout << (*p).getArea() << endl; // pizza의 getArea() 호출
}
```

```
3.14
3.14
3.14
2826
2826
```

7.2 동적메모리 할당 및 반환

동적 메모리의 필요성

- 동적 메모리
  - **프로그램 실행 중에 메모리의 할당과 해제가 결정되는 메모리**

- 동적 메모리를 사용하면
  - 메모리 낭비 해결
    - 실행 중에 꼭 필요한 만큼 메모리를 할당 받아서 사용
    - 미리 정해진 크기가 아니라 원하는 크기만큼 할당 받는 것도 가능
  - 메모리의 할당과 해제 시점을 전적으로 프로그래머가 제어할 수 있음

정적 할당과 동적 할당

- 정적 할당
  - 변수 선언을 통해 필요한 메모리 할당
    - 많은 양의 메모리는 배열 선언을 통해 할당

- 동적 할당
  - 필요한 양이 예측되지 않는 경우. 프로그램 작성시 할당 받을 수 없음
  - 실행 중에 운영체제로부터 할당 받음
    - 힙(heap)으로부터 할당
      - 힙은 운영체제가 소유하고 관리하는 메모리. 모든 프로세스가 공유할 수 있는 메모리

정적 메모리 vs. 동적 메모리

특 징	정적 메모리	동적 메모리
메모리 할당	컴파일 시간에 이루어짐	실행시간에 이루어짐 C : malloc() C++ : new 연산자
메모리 해제	자동으로 해제	명시적으로 해제해야 함 C : free() C++ : delete 연산자 사용
사용범위	지역변수는 선언된 블록 내 전역변수는 프로그램 전체	명시적으로 해제해야 함 delete 연산자 사용 프로그래머가 원하는 동안만큼
메모리 관리 책임	컴파일러 책임	프로그래머의 책임

new와 delete 연산자

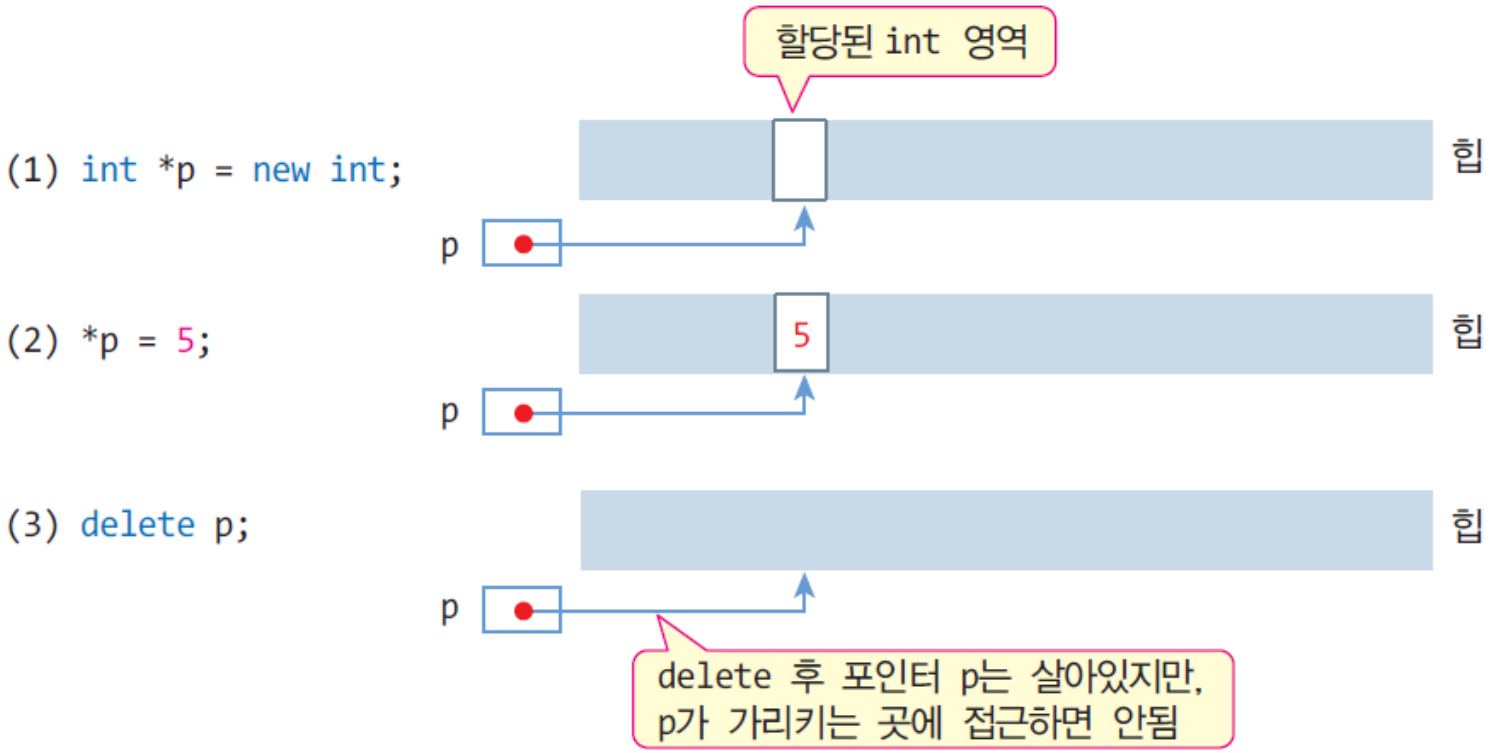
- C++의 기본 연산자
- new/delete 연산자의 사용 형식

```
데이터타입 *포인터변수 = new 데이터타입 ;
delete 포인터변수;
```

- new/delete의 사용

```
int *pInt = new int; // int 타입의 메모리 동적 할당
char *pChar = new char; // char 타입의 메모리 동적 할당
Circle *pCircle = new Circle(); // Circle 클래스 타입의 메모리 동적 할당

delete pInt; // 할당 받은 정수 공간 반환
delete pChar; // 할당 받은 문자 공간 반환
delete pCircle; // 할당 받은 객체 공간 반환
```

기본 타입의 메모리 동적 할당 및 반환



- delete 주의!)
  - 동적으로 할당된 메모리 주소를 저장하는 포인터 변수가 없어지는 것이 아님.
  - 따라서 delete 연산자로 동적 메모리를 해제한 다음에는 동적 메모리의 주소를 저장하는 포인터 변수를 널 포인터로 지정하는 것이 안전함.

정수형 공간의 동적 할당 및 반환 예

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

int main() {
   int *p;

   p = new int;
   if(!p) {
      cout << "메모리를 할당할 수 없습니다.";
      return 0;
   }

   *p = 5; // 할당 받은 정수 공간에 5 삽입
   int n = *p;
   cout << "*p = " << *p << '□n';
   cout << "n = " << n << '□n';

   delete p;
}
```

int 타입 1개 할당

p 가 NULL이면, 메모리 할당 실패

할당 받은 메모리 반환

```
*p = 5
n = 5
```

delete 사용 시 주의 사항

- 적절치 못한 포인터로 delete하면 실행 시간 오류 발생
  - 동적으로 할당 받지 않는 메모리 반환 – 오류

```
int n;
int *p = &n;
delete p; // 실행 시간 오류
// 포인터 p가 가리키는 메모리는 동적으로 할당 받은 것이 아님
```

  - 동일한 메모리 두 번 반환 – 오류

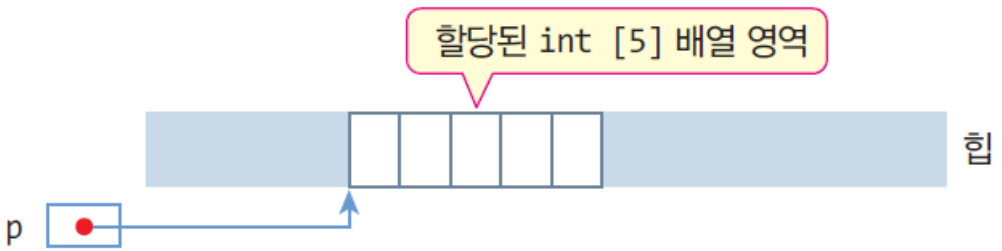
```
int *p = new int;
delete p; // 정상적인 메모리 반환
delete p; // 실행 시간 오류. 이미 반환한 메모리를 중복 반환할 수 없음
```

배열의 동적 할당 및 반환

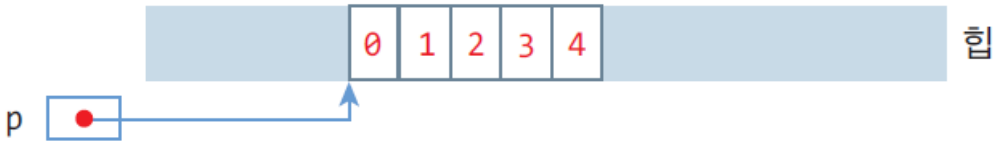
- new/delete 연산자의 사용 형식

데이터타입 *포인터변수 = **new** 데이터타입 **[배열의 크기]**; // 동적 배열 할당
delete [] 포인터변수; // 배열 반환

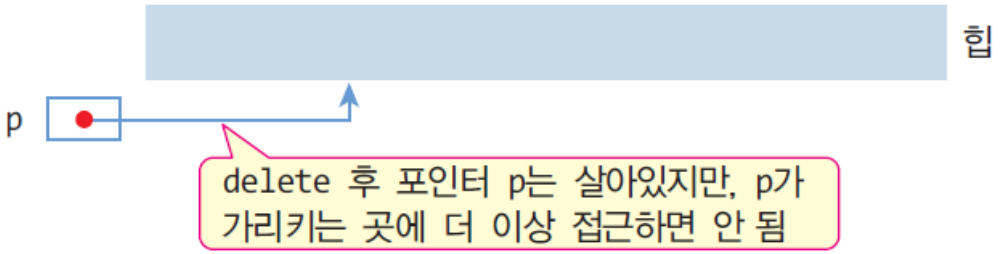
```
(1) int *p = new int [5];
```




```
(2) for(int i=0; i<5; i++)
        p[i] = i;
```




```
(3) delete [] p;
```



정수형 배열의 동적 할당 및 반환

- 사용자로부터 입력할 정수의 개수를 입력 받아 배열을 동적 할당 받고, 하나씩 정수를 입력 받은 후 합을 출력하는 프로그램

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

int main() {
    cout << "입력할 정수의 개수는?";
    int n;
    cin >> n; // 정수의 개수 입력
    if(n <= 0) return 0;
    int *p = new int[n]; // n 개의 정수 배열 동적 할당
    if(!p) {
        cout << "메모리를 할당할 수 없습니다.";
        return 0;
    }

    for(int i=0; i<n; i++) {
        cout << i+1 << "번째 정수: "; // 프롬프트 출력
        cin >> p[i]; // 키보드로부터 정수 입력
    }

    int sum = 0;
    for(int i=0; i<n; i++)
        sum += p[i];
    cout << "평균 = " << sum/n << endl;

    delete [] p; // 배열 메모리 반환
}
```

```
입력할 정수의 개수는?4
1번째 정수: 4
2번째 정수: 20
3번째 정수: -5
4번째 정수: 9
평균 = 7
```

동적 메모리 초기화 및 delete시 유의사항

동적 할당 메모리 초기화

- 동적 할당 시 초기화

```
데이터타입 *포인터변수 = new 데이터타입(초기값);
```

```
int *pInt = new int(20); // 20으로 초기화된 int 타입 할당
char *pChar = new char('a'); // 'a'로 초기화된 char 타입 할당
```

- 배열은 동적 할당 시 초기화 불가능

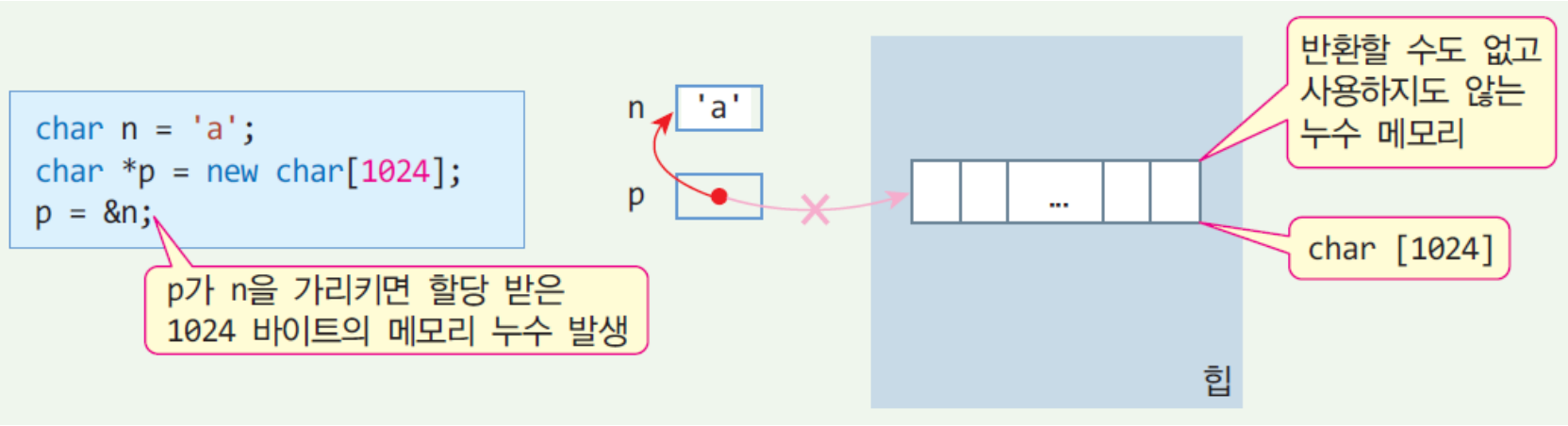
```
int *pArray = new int [10](20); // 구문 오류. 컴파일 오류 발생
int *pArray = new int(20)[10]; // 구문 오류. 컴파일 오류 발생
```

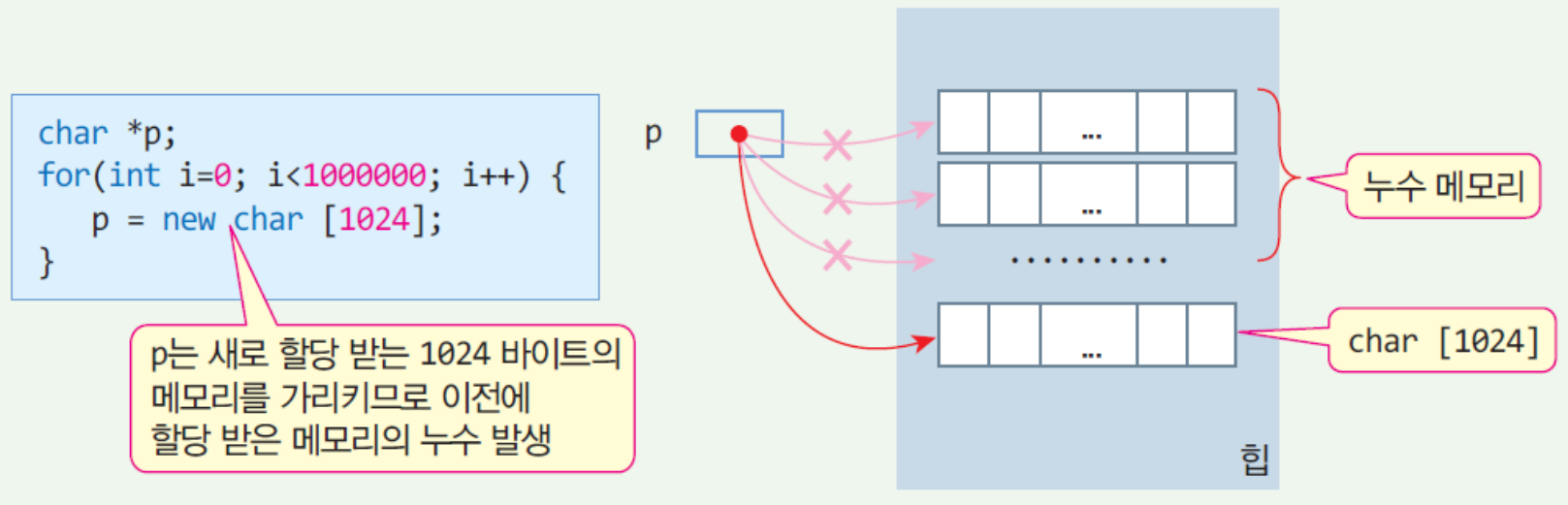
delete시 [] 생략

- 컴파일 오류는 아니지만 비정상적인 반환

```
int *p = new int [10];
delete p; // 비정상 반환. delete [] p;로 하여야 함.

int *q = new int;
delete [] q; // 비정상 반환. delete q;로 하여야 함.
```

동적 메모리 할당과 메모리 누수





* 프로그램이 종료되면, 운영체제는 누수 메모리를 모두 힙에 반환

기본적인 동적 메모리 할당과 해제

- 동적 메모리 할당을 사용하는 방법의 정리
  - 메모리를 할당할 때는 타입과 크기를 지정한다. 그러면 컴퓨터가 메모리를 할당한 후에 그 메모리의 주소를 보관한다. 우리는 이 주소를 보관해두어야 한다.

```
int num;
int *data = new int;
```

  - 보관해둔 주소를 통해서 메모리 공간을 사용할 수 있다. 이 때는 배열의 원소를 가리키는 포인터처럼 사용할 수 있다.

```
*data = 6;
```

  - 사용이 끝난 후에는 반드시 보관해 둔 주소를 알려주면서 메모리를 해제한다.

```
delete data;
```

동적 메모리 할당의 규칙(1)

① new, delete와 new[], delete[] 쌍을 맞춰서 사용하자.

② NULL 포인터를 해제하는 것은 안전하다.

- delete, delete[] 연산자는 메모리가 주소 값으로 NULL이 넘겨져 온 경우에는 아무일도 하지 않음
- delete, delete[] 연산자가 알아서 NULL인 경우를 처리함

```
char* p = NULL;
delete p;   // OK
```

동적 메모리 할당의 규칙(2)

③ 적절치 못한 포인터로 delete하면 실행시간 오류 발생

- 동적으로 할당 받지 않는 메모리 반환 – 오류

```
int n;
int *p = &n;
delete p; // 실행 시간 오류
// 포인터 p가 가리키는 메모리는 동적으로 할당 받은 것이 아님
```

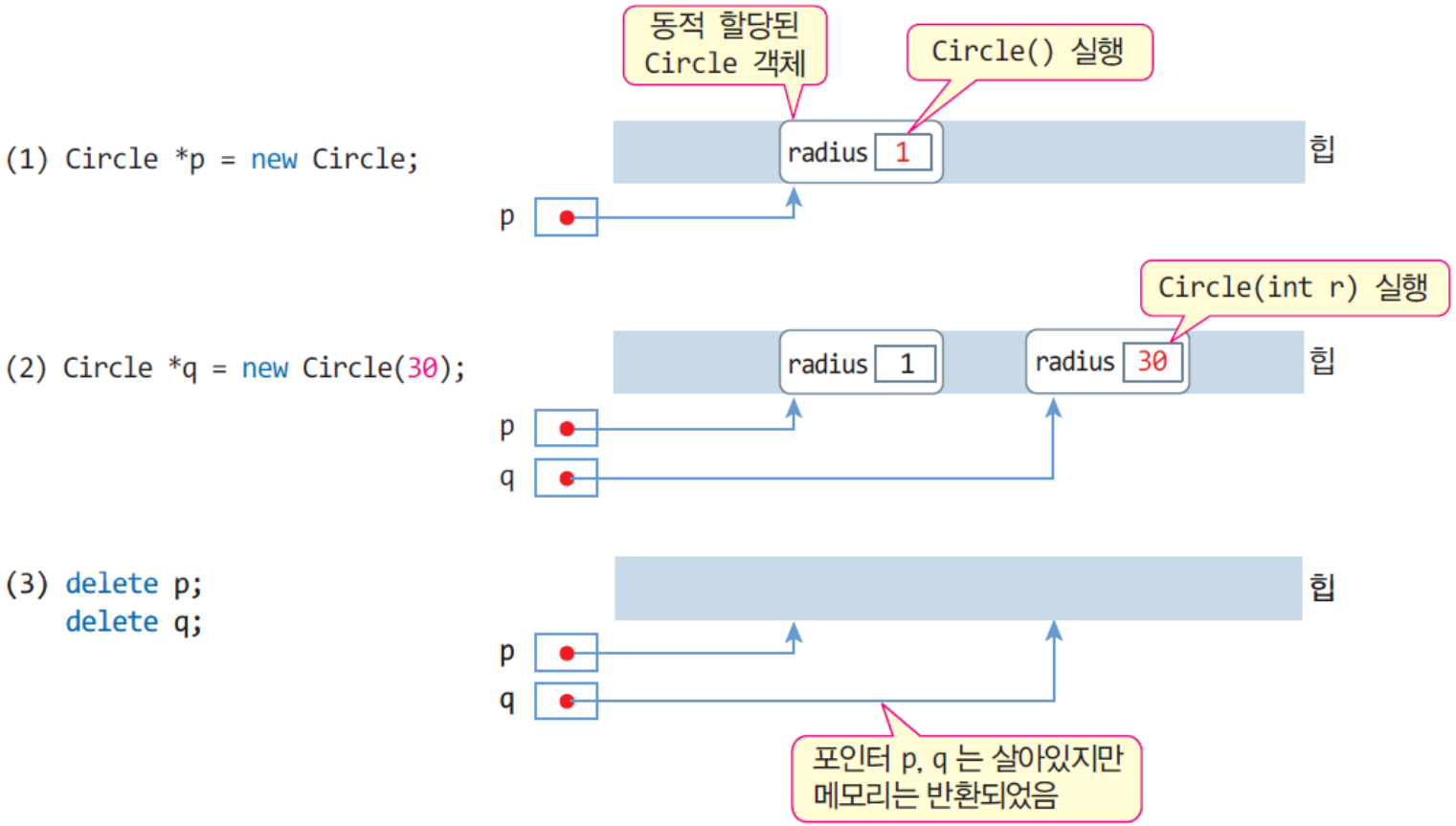
- 동일한 메모리 두 번 반환 - 오류

```
int *p = new int;
delete p; // 정상적인 메모리 반환
delete p; // 실행 시간 오류. 이미 반환한 메모리를 중복 반환할 수 없음
```

객체(배열)의 동적 생성 및 반환

객체의 동적 생성 및 반환

클래스이름 *포인터변수 = **new** 클래스이름;
클래스이름 *포인터변수 = **new** 클래스이름(생성자매개변수리스트);
delete 포인터변수;



Circle 객체의 동적 생성 및 반환

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

class Circle {
    int radius;
public:
    Circle();
    Circle(int r);
    ~Circle();
    void setRadius(int r) { radius = r; }
    double getArea() { return 3.14*radius*radius; }
};

Circle::Circle() {
    radius = 1;
    cout << "생성자 실행 radius = " << radius << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout << "생성자 실행 radius = " << radius << endl;
}

Circle::~Circle() {
    cout << "소멸자 실행 radius = " << radius << endl;
}
```

```
int main() {
    Circle *p, *q;
    p = new Circle;
    q = new Circle(30);
    cout << p→getArea() << endl << q→getArea() << endl;
    delete p;
    delete q;
}
```

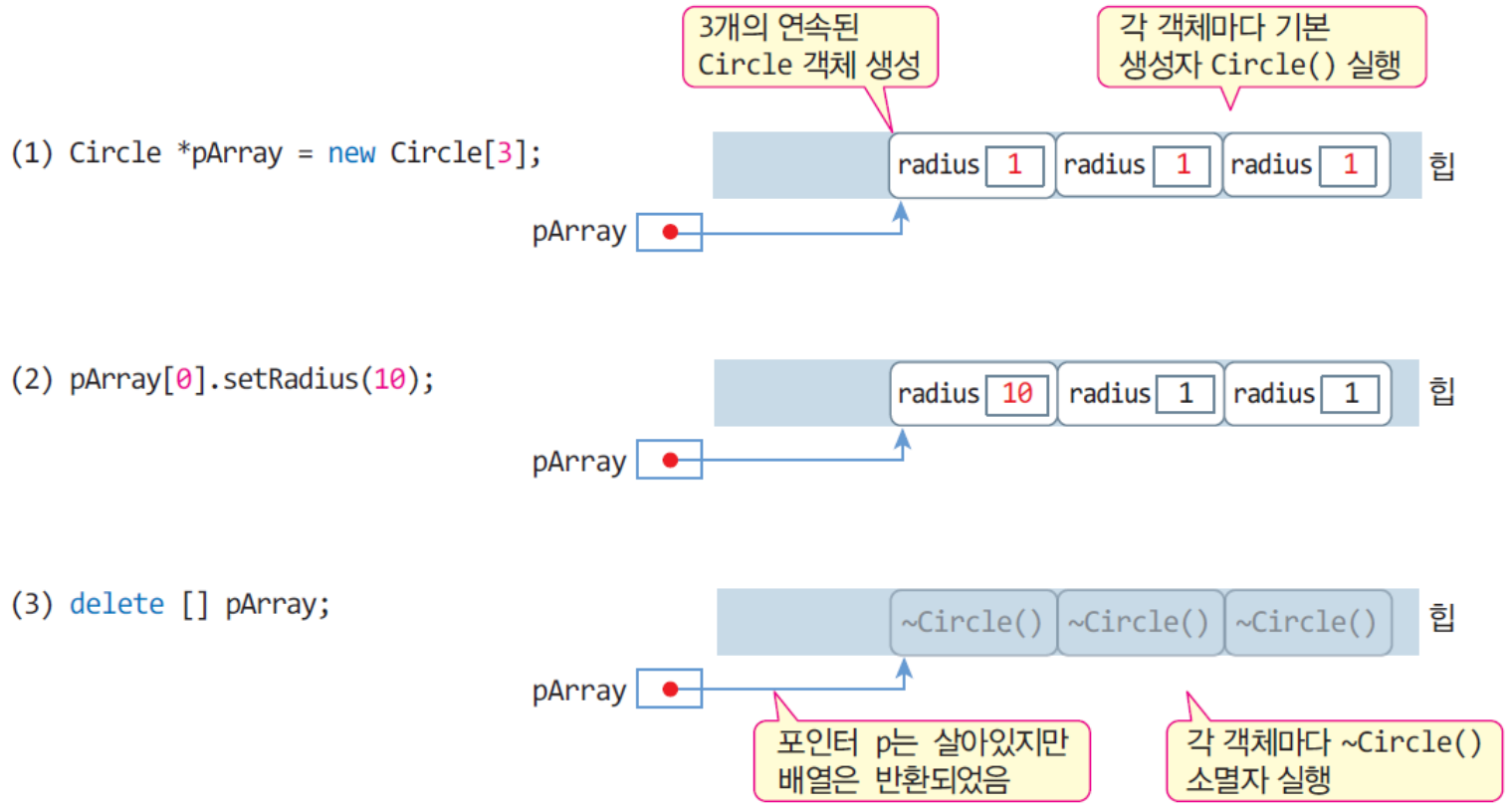
생성한 순서에 관계 없이 원하는 순서대로 delete 할 수 있음

```
생성자 실행 radius = 1
생성자 실행 radius = 30
3.14
2826
소멸자 실행 radius = 1
소멸자 실행 radius = 30
```

객체 배열의 동적 생성 및 반환

클래스이름 *포인터변수 = **new** 클래스이름**[배열크기];**
delete[] 포인터변수; // 포인터변수가 가리키는 객체 배열을 반환

동적 객체 배열은 항상 디폴트 생성자로 초기화
만일 클래스가 디폴트 생성자를 제공하지 않으면 동적 객체 배열을 생성하는 것은 불가능



객체 배열의 사용, 배열의 반환과 소멸자

- 동적으로 생성된 배열도 보통 배열처럼 사용

```
Circle *pArray = new Circle[3]; // 3개의 Circle 객체 배열의 동적 생성

pArray[0].setRadius(10); // 배열의 첫 번째 객체의 setRadius() 멤버 함수 호출
pArray[1].setRadius(20); // 배열의 두 번째 객체의 setRadius() 멤버 함수 호출
pArray[2].setRadius(30); // 배열의 세 번째 객체의 setRadius() 멤버 함수 호출

for(int i=0; i<3; i++) {
   cout << pArray[i].getArea(); // 배열의 i 번째 객체의 getArea() 멤버 함수 호출
}
```

- 포인터로 배열 접근

```
pArray->setRadius(10);
(pArray+1)->setRadius(20);
(pArray+2)->setRadius(30);

for(int i=0; i<3; i++) {
   (pArray+i)->getArea();
}
```

- 배열 소멸

```
delete [] pArray;
```

pArray[2] 객체의 소멸자 실행(1)
pArray[1] 객체의 소멸자 실행(2)
pArray[0] 객체의 소멸자 실행(3)

각 원소 객체의 소멸자 별도 실행. 생성의 반대순

Circle 배열의 동적 생성 및 반환

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

class Circle {
    int radius;
public:
    Circle();
    Circle(int r);
    ~Circle();
    void setRadius(int r) { radius = r; }
    double getArea() { return 3.14*radius*radius; }
};

Circle::Circle() {
    radius = 1;
    cout <<  " 기본생성자  radius = " << radius << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout <<  " 인자생성자 radius = " << radius << endl;
}

Circle::~Circle() {
    cout << "소멸자 radius = " << radius << endl;
}
```

```
int main() {
    Circle *pArray = new Circle [3]; // 객체 배열 생성

    pArray[0].setRadius(10);
    pArray[1].setRadius(20);
    pArray[2].setRadius(30);

    for(int i=0; i<3; i++) {
        cout << pArray[i].getArea() << '□n';
    }
    Circle *p = pArray; // 포인터 p에 배열의 주소값으로 설정
    for(int i=0; i<3; i++) {
        cout << p→getArea() << '□n';
        p++; // 다음 원소의 주소로 증가
    }

    delete [] pArray; // 객체 배열 소멸
}
```

각 원소 객체의
디폴트 생성자 Circle() 실행

각 배열 원소 객체의
소멸자 ~Circle() 실행

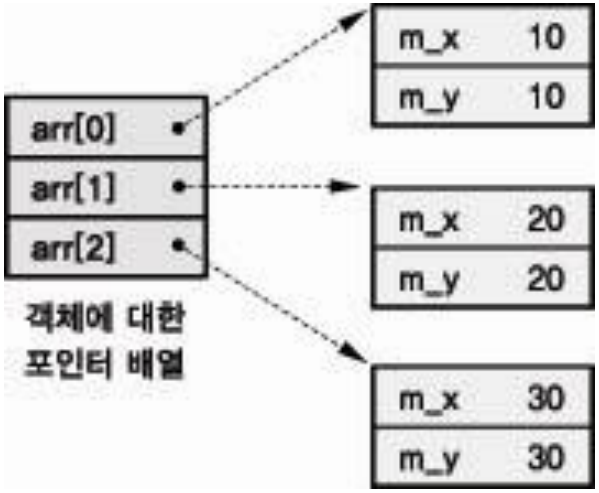
```
기본생성자 radius = 1
기본생성자 radius = 1
기본생성자 radius = 1
314
1256
2826
314
1256
2826
소멸자 radius = 30
소멸자 radius = 20
소멸자 radius = 10
```

소멸자는 생성의
반대 순으로 실행

객체에 대한 포인터 배열의 생성 및 사용

- 객체의 주소를 저장하는 포인터 배열

```
Point *arr[3] = { new Point(10, 10), new Point(20, 20), new Point(30, 30) };
for(int i = 0 ; i < 3 ; i++ )
    arr[i]→Print();  // arr[i]는 객체에 대한 포인터이므로 → 사용

for(int i = 0 ; i < 3 ; i++ )
    delete arr[i];
```



7.4 멤버함수의 this 포인터

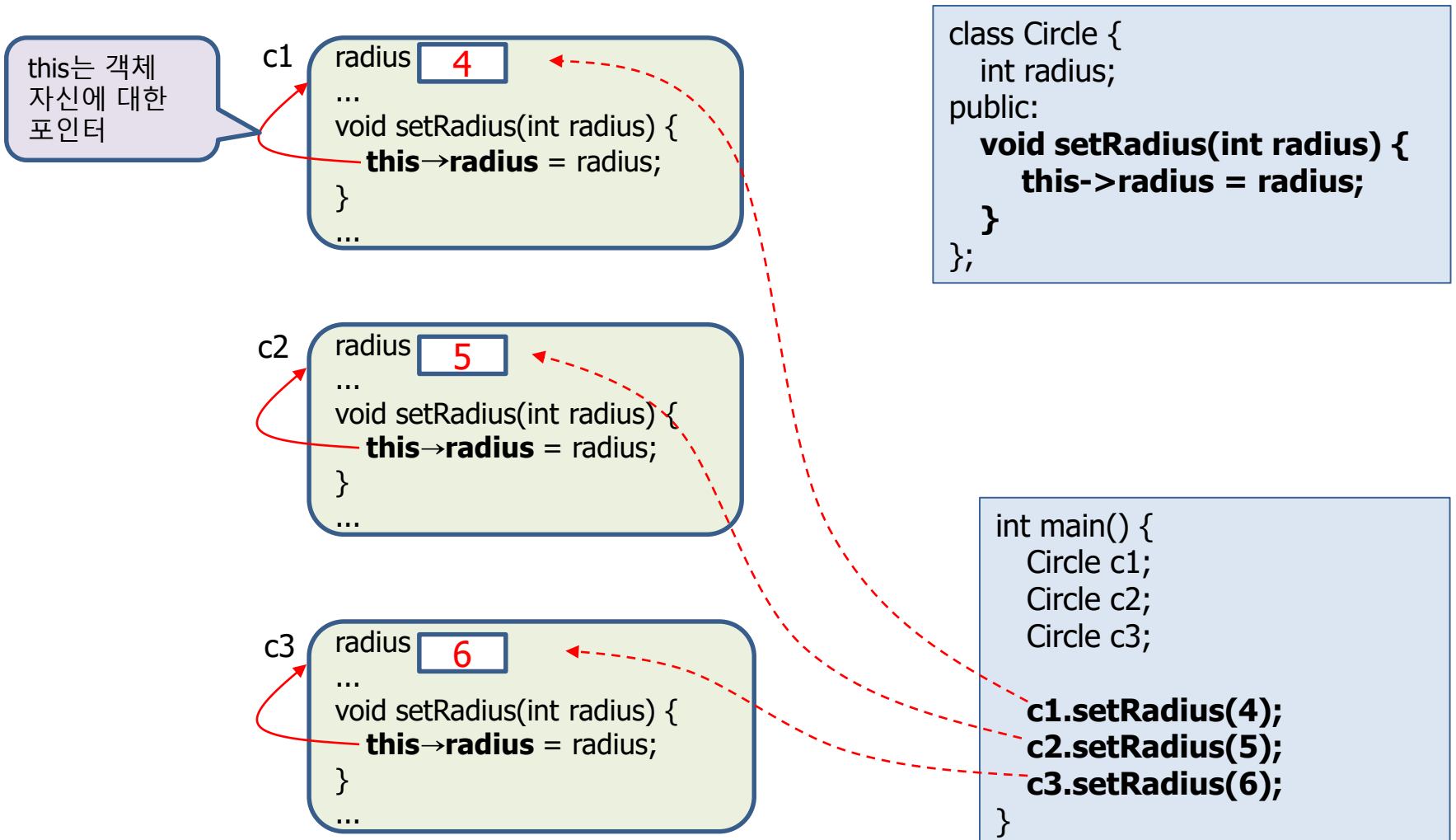
this 포인터

- this
  - **포인터,** 멤버 함수를 소유한 객체를 가리키는 포인터
  - 클래스의 멤버 함수 내에서만 사용
  - 개발자가 선언하는 변수가 아니고, 컴파일러가 선언한 변수
    - 멤버 함수에 컴파일러에 의해 묵시적으로 삽입 선언되는 매개 변수

```
class Circle {
   int radius;
public:
   Circle() { this→radius=1; }
   Circle(int radius) { this→radius = radius; }
   void setRadius(int radius) { this→radius = radius; }
   ....
};
```

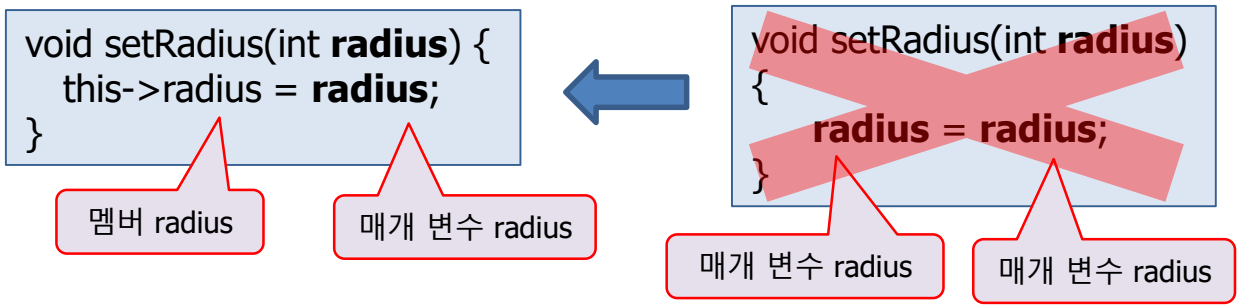
this와 객체

* 각 객체 속의 **this**는 다른 객체의 **this**와 다름



this가 유용한 경우

- 매개변수의 이름과 멤버변수의 이름이 같은 경우



- 멤버 함수가 객체 자신의 주소를 리턴할 때

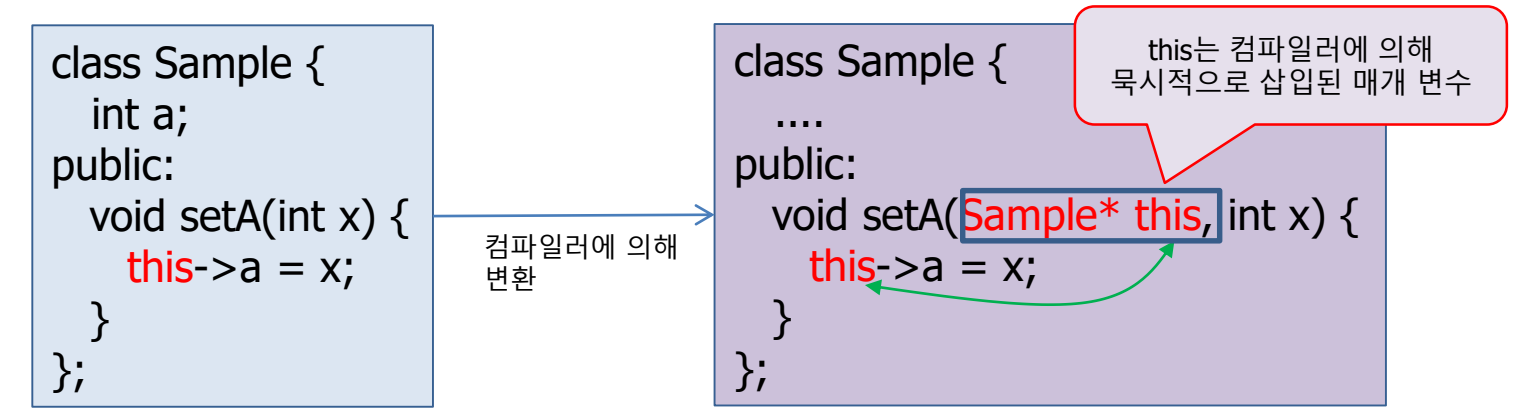
```
class Sample {
public:
   Sample* f() {
      ....
      return this;
   }
};
```

this의 제약 사항

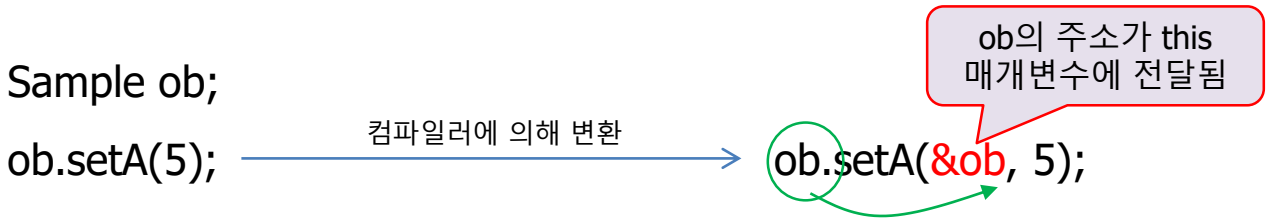
- 멤버 함수가 아닌 함수에서 this 사용 불가
  - 객체와의 관련성이 없기 때문

- static 멤버 함수에서 this 사용 불가
  - 객체가 생기기 전에 static 함수 호출이 있을 수 있기 때문에

this 포인터의 실체 – 컴파일러에서 처리



(a) 개발자가 작성한 클래스 (b) 컴파일러에 의해 변환된 클래스



(c) 객체의 멤버 함수를 호출하는 코드의 변환

다음 수업

- 함수와 참조, 복사생성자
  - 1_ 객체전달과 참조
  - 2_ 복사생성자