객체 포인터
객체 배열
객체의 동적생성

학습 목표

- 객체 포인터
- 객체 배열의 생성, 초기화, 소멸
- 동적 메모리 할당. 초기화, 반환
- this 포인터
- 스마트 포인터
  - unique_ptr
  - shared_ptr
  - weak_ptr
- 객체 포인터 배열

객체 포인터

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

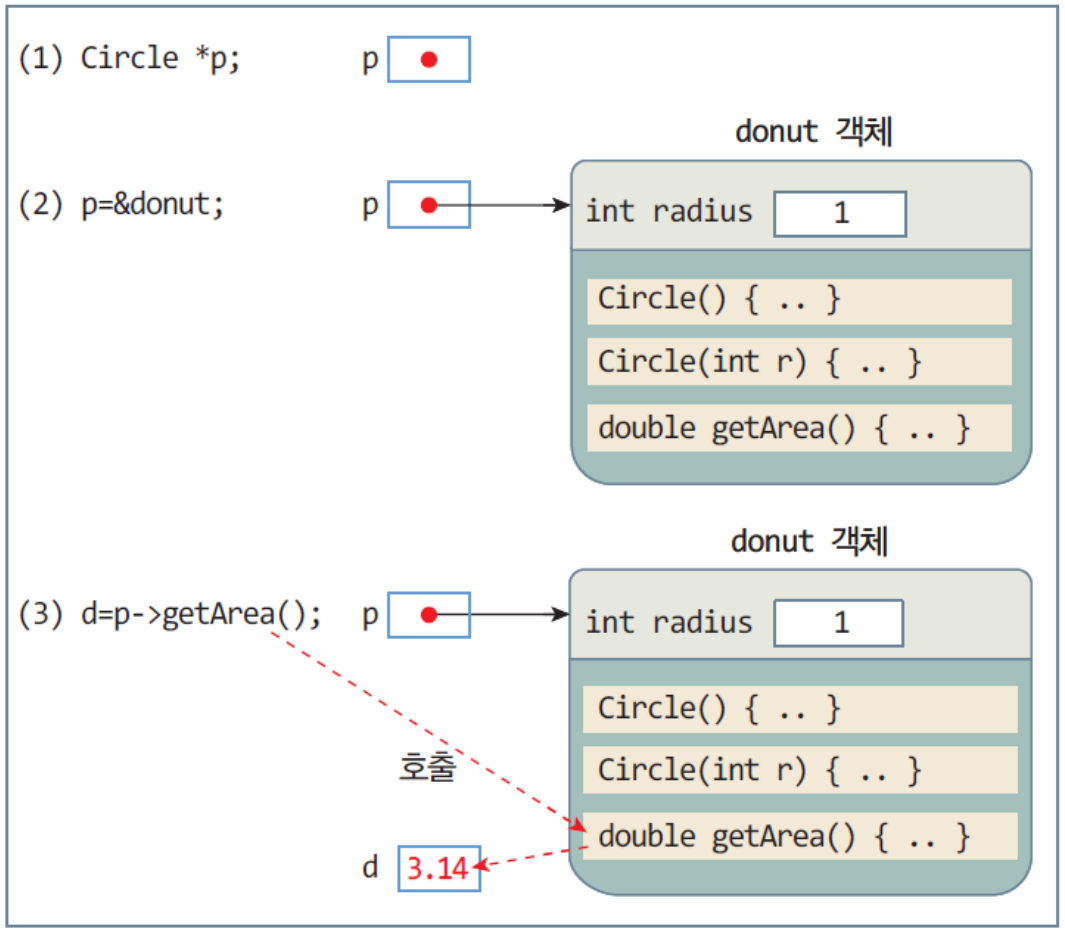
- 포인터로 멤버를 접근할 때
  - 객체 포인터 → 멤버
  - (*객체포인터).멤버

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

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

객체에 대한 포인터 선언

포인터에 객체 주소 저장

멤버 함수 호출



nullptr

- Null Pointer를 의미
  - 포인터가 아무것도 가리키고 있지 않음
- 널 포인터를 NULL 매크로나 상수 0 사용 시 문제점
  - NULL 매크로나 상수 0을 사용하여 함수에 인자로 넘기는 경우 int 타입으로 추론

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

void fa(int *a) {
    //cout << "fa) *a = " << *a << endl;
    cout << "fa) a = " << a << endl;
}
void fb(double *p) {
    //cout << "fb) *p = " << *p <<endl;
    cout << "fb) p = " << p <<endl;
}
int main() {
    double *p = nullptr;
    fa(NULL);
    fb(p);
}
```

객체 배열의 생성 및 소멸

- 객체 배열 선언 가능, 기본 타입 배열 선언과 형식 동일
  - int n[3]; // 정수형 배열 선언
  - Circle c[3]; // Circle 타입의 배열 선언

- 객체 배열 선언
  1. 객체 배열을 위한 공간 할당
  2. 배열의 각 원소인 객체마다 디폴트 생성자 호출
     - c[0]의 생성자, c[1]의 생성자, c[2]의 생성자 순서로 실행
     - **매개 변수 없는 디폴트 생성자 호출**(매개 변수 있는 생성자는 호출할 수 없음)
     - Circle circleArray[3](5); // 오류

- 배열 소멸
  - 배열의 각 객체마다 소멸자 호출. 생성의 반대 순서로 소멸
    - c[2]의 소멸자, c[1]의 소멸자, c[0]의 소멸자 순서로 실행

객체 배열의 선언 및 활용

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

class Circle {
  int radius;

public:
   Circle() = default;

   Circle(int r) : radius(r) { }

   void setRadius(int r) { radius = r; }
   int getRadius(int r) { return radius }
   double getArea();
};

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

```
int main() {
  Circle cirarr[2]; //Circle 객체 배열 생성, 디폴트 생성자가 없으면 오류

  for(int i=0; i<2; i++) //배열의 각 원소인 객체의 멤버 접근
    cout << "Circle " << i << "의 radius=" << cirarr[i]. getRadius() << endl;

  cirarr[0].setRadius(10); //배열의 각 원소인 객체의 멤버 접근
  cirarr[1].setRadius(20);

  for(int i=0; i<2; i++)
    cout << "Circle " << i << "의 면적=" << cirarr[i].getArea() << endl;

  for (auto obj : cirarr) // auto & 범위 기반 for
    cout << "Circle 면적= >> " << obj.getArea() << endl;

  Circle *p;
  p = cirarr;
  for(int i=0; i<2; i++) {    // 객체 포인터로 배열 접근
    cout << "Circle " << i << "의 면적= " << p->getArea() << endl;
    p++;
   }
}
```

객체 배열의 초기화

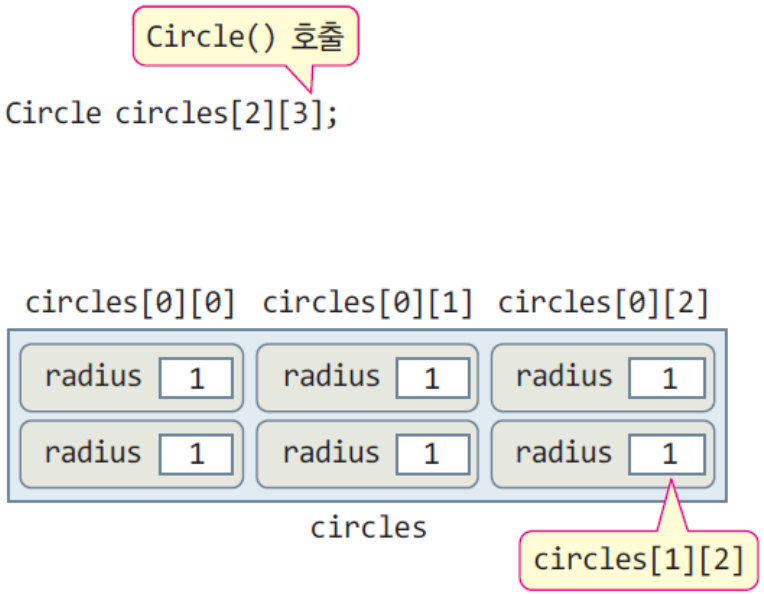
```
class Circle {
  int radius;
public:
  Circle() : Circle(1) { }
  Circle(int r) : radius(r) { }
  void setRadius(int r)  { radius = r; }
};
```

- 객체 배열 초기화 방법
  - 배열의 각 원소인 객체 각각에 생성자를 지정하는 방법

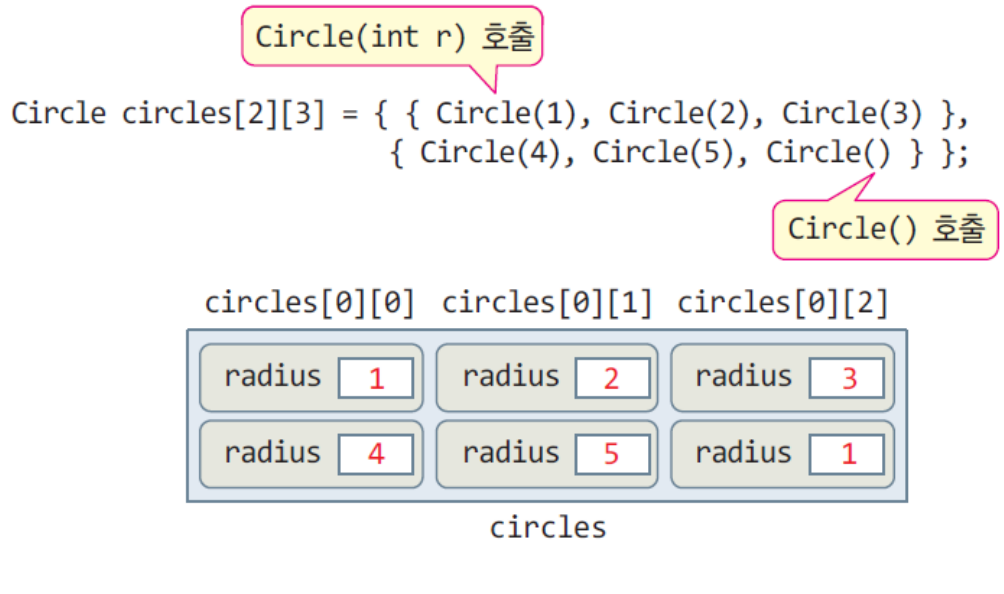
```
Circle circleArray[3] = { Circle(10), Circle(20), Circle() };
```

  - 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

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

2차원 객체 배열의 선언 및 활용

```
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
```

stack vs heap

- stack vs heap
  - function life cycle – stack frame vs pointer 관리
  - memory size – small vs large
  - memory allocation – compile time vs runtime(dynamic)

- stack memory allocation
  - 변수 선언을 통해 필요한 메모리 할당.
  - 많은 양의 메모리는 배열 선언을 통해 할당.
  - int, float, double,
  - small array [(int a[200]; //800b), (array<int, 300> a; //1.2kb)], 객체 몇 개.

- heap memory allocation
  - 필요한 양이 예측되지 않는 경우는 프로그램 작성시 메모리를 할당 받을 수 없음(dynamic).
  - 실행 중 운영체제로부터 힙(heap) 메모리를 할당 받음. 힙 메모리는 운영체제가 소유하고 관리.
  - large array[(array<int, 500000> a; //2mb), (vector<int> a(500000); //2mb)], 몇 백kb이상의 객체,
  - runtime(dynamic) 할당 시

동적 메모리 할당 및 반환

- C 언어의 동적 메모리 할당
  - malloc()/free() 라이브러리 함수 사용

```
int *a = (int *)malloc(sizeof(int));
*a = 100;
free(a);

//heap int array : c style
int *b = (int *)malloc(sizeof(int)*3);
b[0] = 100;
free(b);
```

- C++의 동적 메모리 할당/반환
  - new 연산자
    - 기본 타입 메모리 할당, 배열 할당, 객체 할당, 객체 배열 할당
    - 객체의 동적 생성 - 힙 메모리에 객체를 위한 메모리 할당 요청
    - **객체 할당 시 생성자 호출**
  - delete 연산자
    - New로 할당 받은 메모리 반환
    - **객체의 동적 소멸, 소멸자 호출 뒤 객체를 힙에 반환**

new와 delete 연산자

- C++의 기본 연산자
- 크기와 형 변환 필요 없음
- new/delete 연산자의 사용 형식

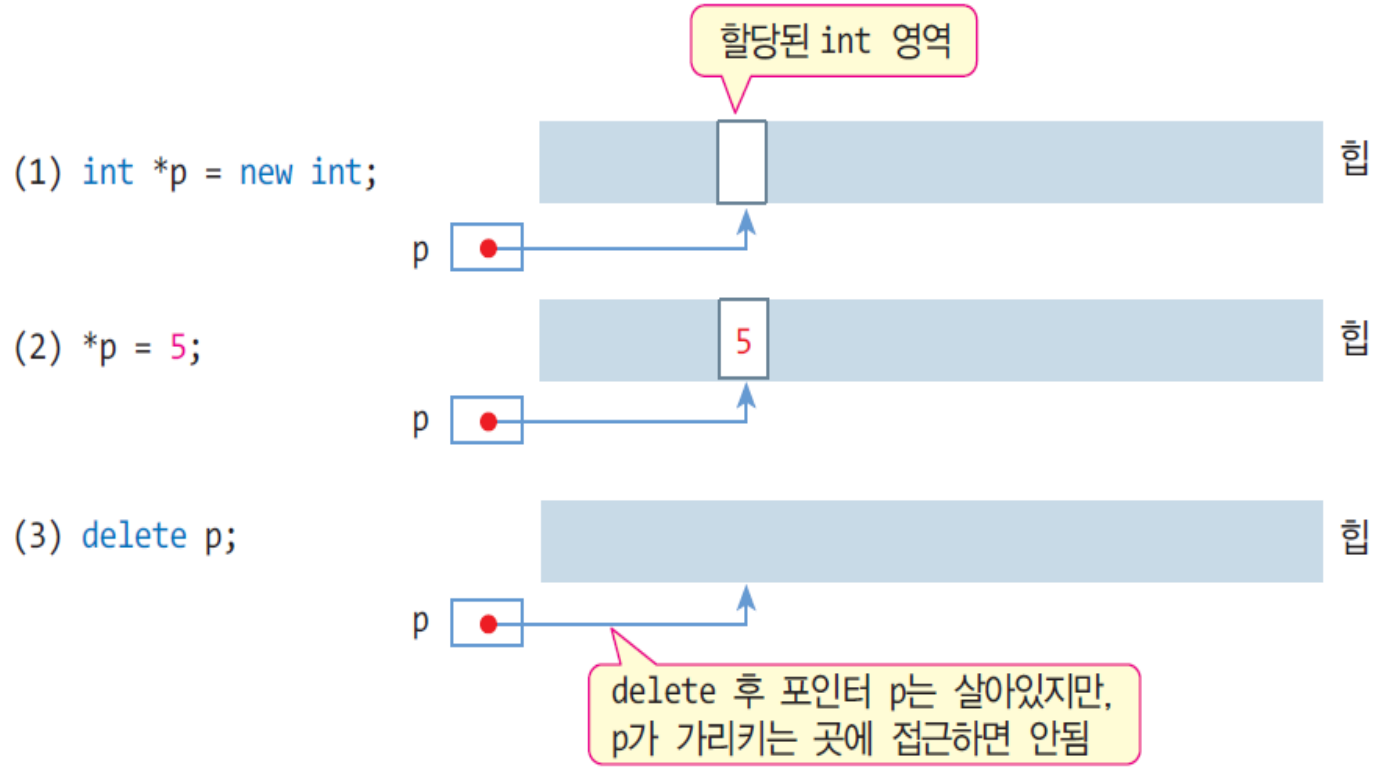
```
데이터타입 *포인터변수 = new 데이터타입 ;
delete 포인터변수;
```

- 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

```
#include <iostream>
using namespace std;
int main() {
    int *p = nullptr;
    p = new int;
    if (!p) {
        cout << "메모리를 할당할 수 없습니다.";
        return 0;
    }

    *p = 5;
    int n = *p;
    cout << "*p = " << *p << '₩n';
    cout << "n = " << n << '₩n';

    delete p;     //할당 받은 메모리 반환
    p = nullptr;
}
```



(4) p = nullptr; //해제된 메모리를 다시 사용할 수 있는 실수 방지

delete 사용 시 주의 사항

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

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

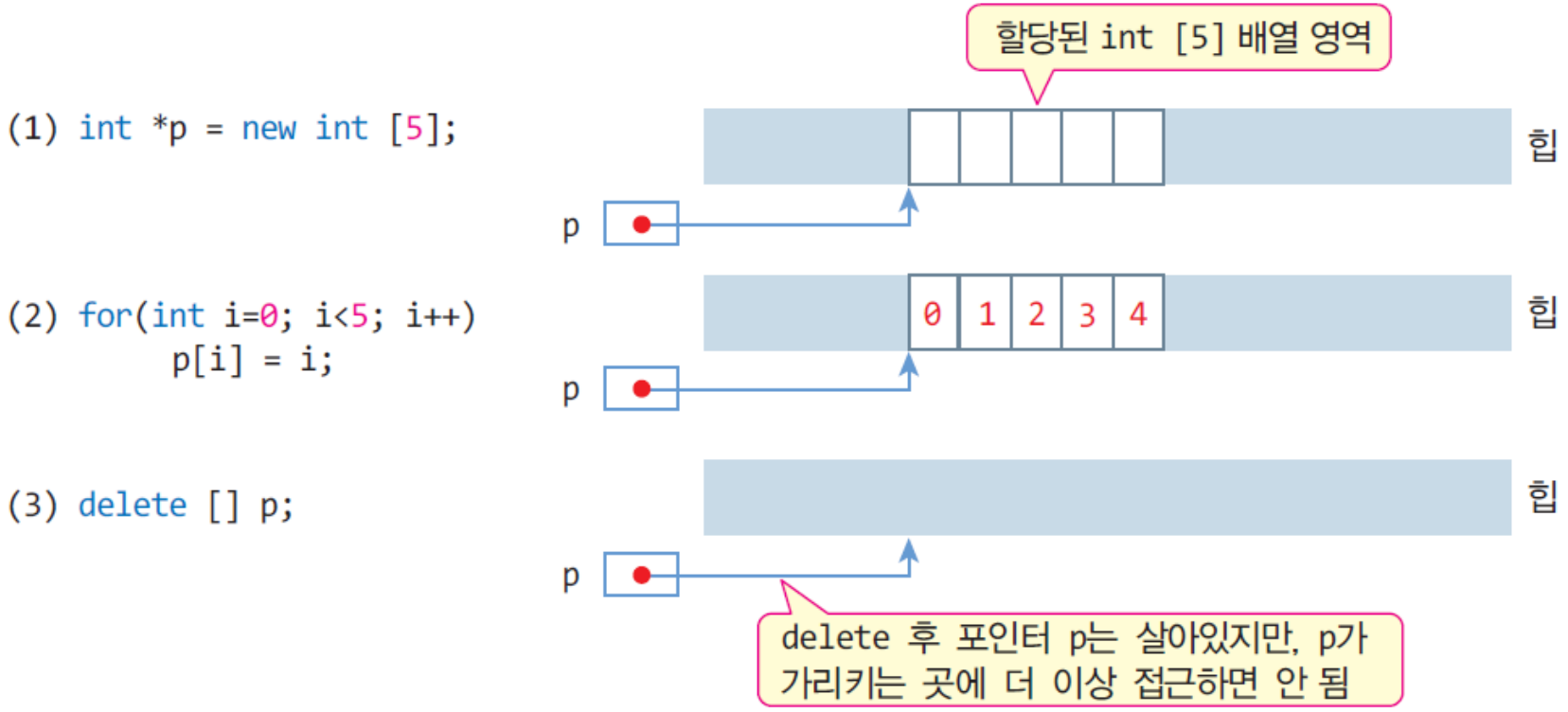
  - 동일한 메모리 두 번 반환 - 오류

```
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
  - **배열 형태로 동적 생성한 것은 배열 형태로 삭제**
  - **배열 인텍스 표시는 생략 불가**

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



동적 할당 메모리의 초기화 및 delete시 유의 사항

- 동적 할당 시 메모리 초기화

```
int *p = new int{ 20 }; // 또는 int *p = new int(20);
cout << "*p = " << *p << endl;

char *ch = new char{ 'c' };  //또는 char *ch = new char('c');
cout << "*ch = " << *ch << endl;
```

- 배열은 동적 할당 시 초기화 불가능

```
//유니폼 초기화를 사용하여 동적으로 할당되는 배열 초기화 가능
int *pArray = new int[4] {3, 4, 5, 6};

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

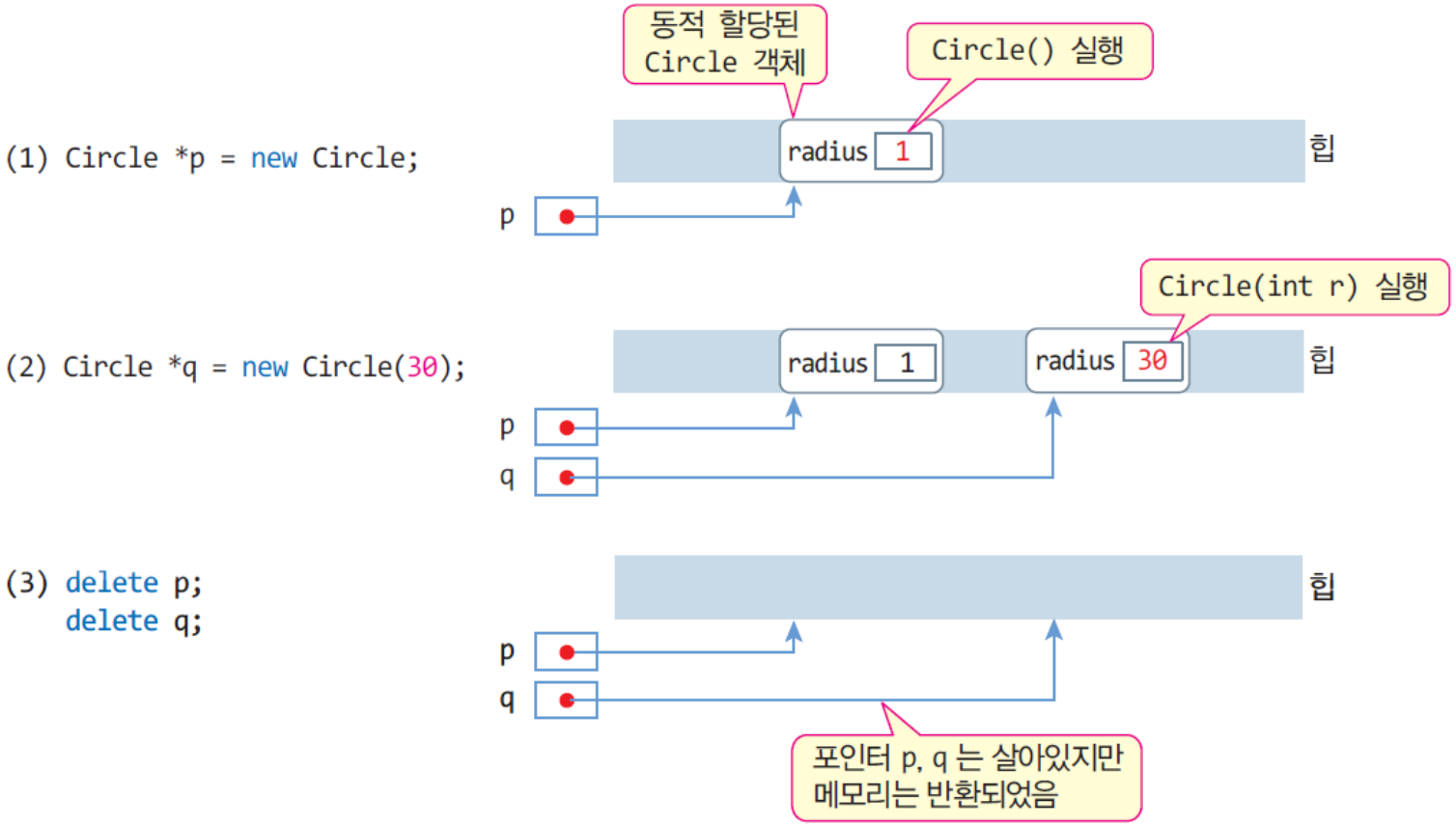
- delete시 [] 생략
  - 컴파일 오류는 아니지만 비정상적인 반환 예

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

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

객체의 동적 생성 및 반환

- new 연산자는 생성자를 호출하고, delete 연산자는 소멸자를 호출

클래스이름 *포인터변수 = **new** 클래스이름;
클래스이름 *포인터변수 = **new** 클래스이름(생성자 매개변수리스트);
delete 포인터변수;




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

객체의 동적 생성 및 반환 예

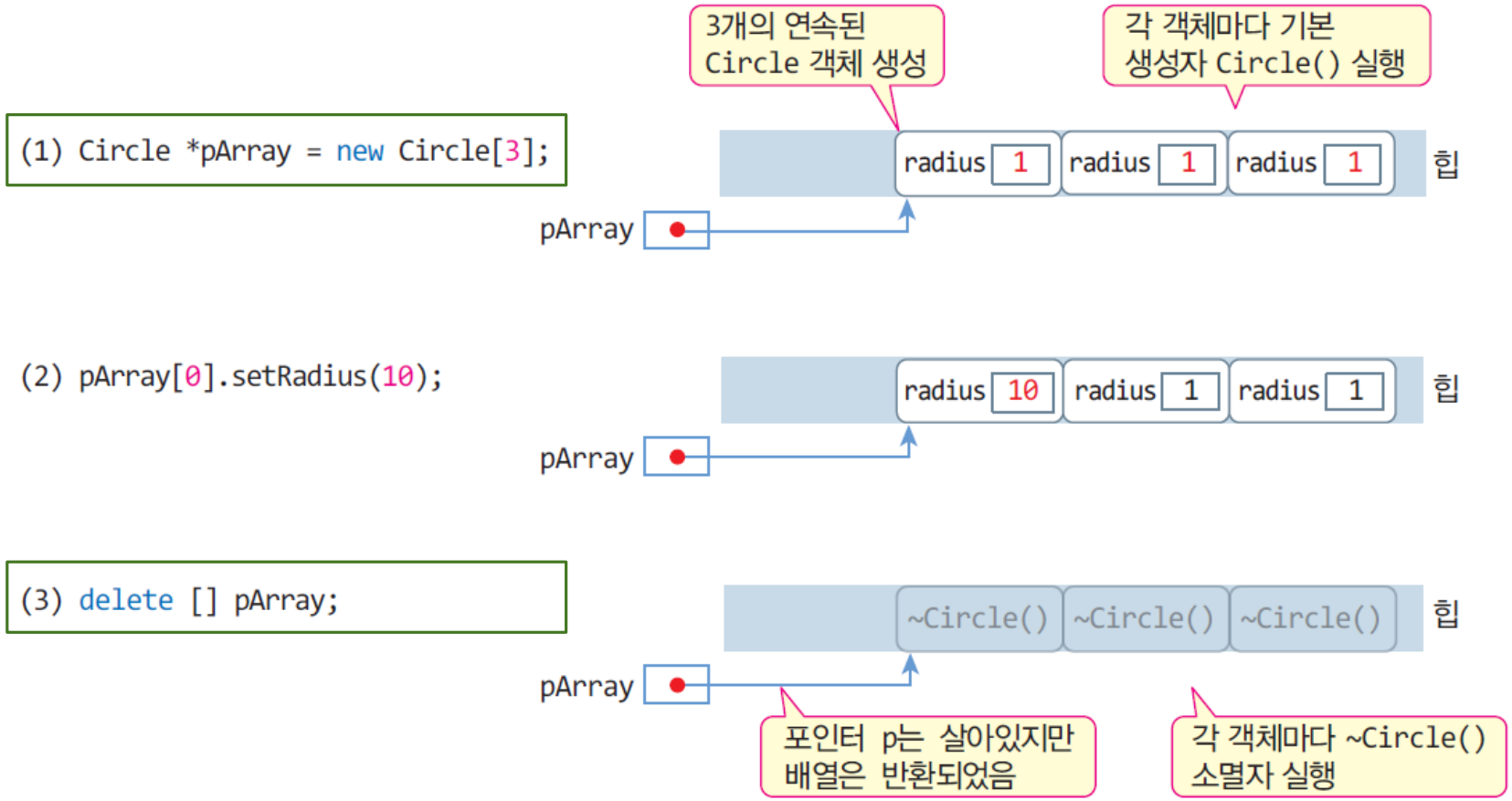
```
#include <iostream>
using namespace std;
class Color {
    int red, green, blue;
    string color;

public:
    Color() : Color(0, 0, 0, "black") { }

    Color(int r, int g, int b, string c) : red(r), green(g), blue(b), color(c) { }

    ~Color() {  cout << color << " 객체 소멸" << endl;  }

    void setColor(int r, int g, int b, string c);
    void show() const;
};
```

```
int main() {
    Color *p=new Color;
    Color *q = new Color(0, 255, 0, "green");

    p->show();
    q->show();

    //생성한 순서에 관계 없이 원하는 순서대로 delete 할 수 있음.
    delete q;
    delete p;
    return 0;
}
```

객체 배열의 동적 생성 및 반환

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

class Circle {
   int radius;

public:
   Circle():Circle(1){ }
   Circle(int r) : radius(r){ }

   void setRadius(int r)  {
       radius = r; }

   double getArea();
};

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

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



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

```
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 반대순서

클래스 멤버의 동적 생성

- 클래스의 멤버도 동적 생성 가능
  - 단, 생성자에서 동적 할당되어야 하고 소멸자에서 동적 메모리를 해제해야 함

```
class Dog {
        string *name;
        int *age;
public:
        Dog(string n, int a) : name{ new string{n} }, age{ new int{a} } {}
        /*
        Dog(string n, int a){          //멤버 변수 동적 메모리 할당
            name = new string(n);   //name = new string{n};
            age = new int(a);          //age=new int{a}
        }
        */
        ~Dog() {
            delete name;  //동적 할당 된 메모리 해제
            delete age;
        }
        int getAge() { return *age; }
        void setAge(int a) { *age = a; }
};
```

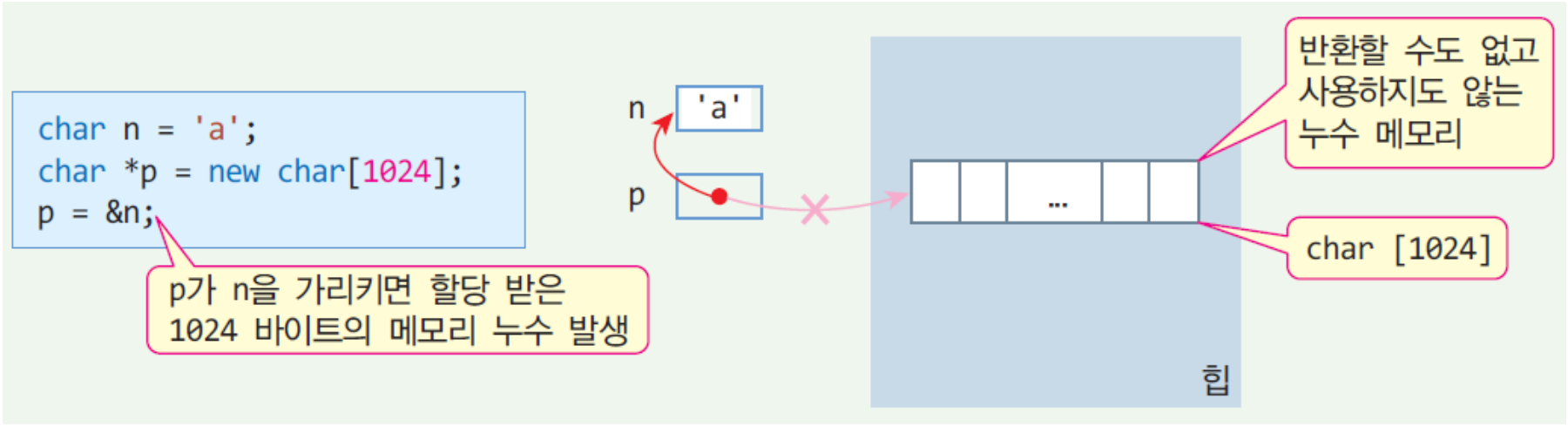
```
int main() {
    Dog *p=new Dog("강아지", 2);
    cout << "강아지 나이 = " << p->getAge() << endl;

    p->setAge(5);
    cout << "강아지 나이 = " << p->getAge() << endl;

    delete p;
}
```

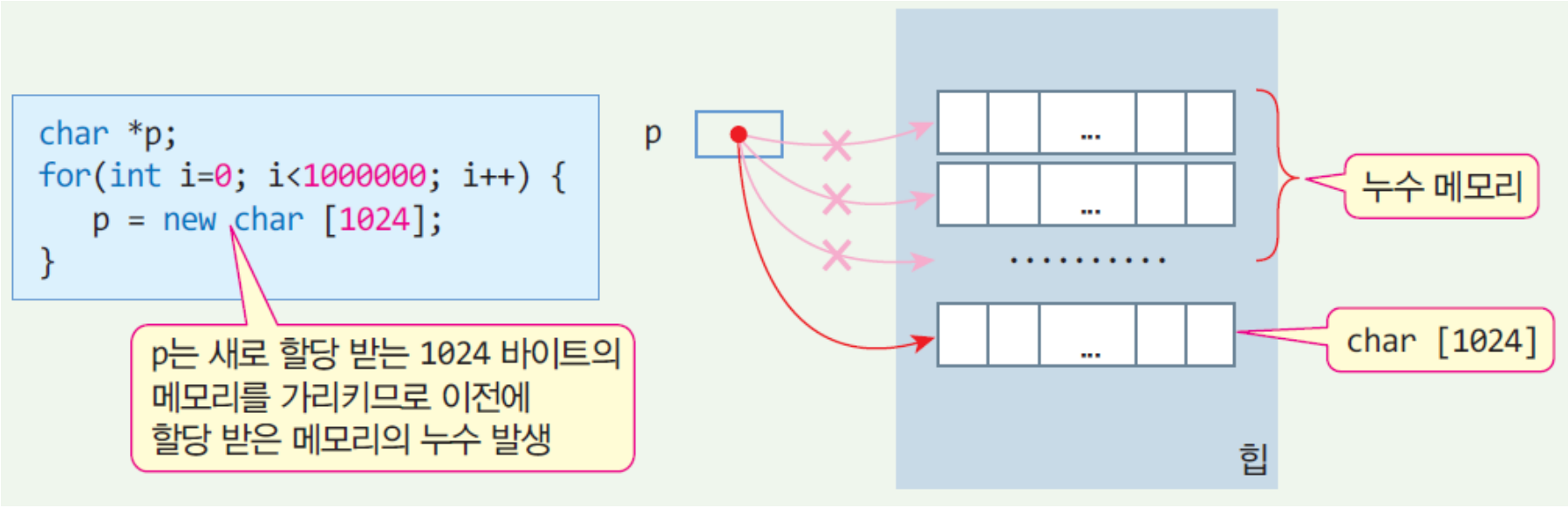
객체 포인터 배열

```
class Person {
        string name, tel;
public:
        Person(string n, string t) :name(n), tel(t) { };
        ~Person() { cout << "객체 소멸" << endl; };
        void disPlay() const { cout << "name=" << name << ", tel=" << tel << endl; }
};
int main() {
    Person *per[3]; //3개의 객체 주소를 저장할 수 있는 객체 포인터 배열 선언을 선언하여 Person 클래스 객체 처리
    string name, tel;

    int size = sizeof(per) / sizeof(per[0]); //배열 크기 계산
    for (int i = 0; i < size; i++) {
        cout << ">> name, tel : "; cin >> name;  cin >> tel;
        per[i] = new Person(name, tel); //생성된 객체의 주소를 저장
    }
    for (int i = 0; i < size; i++) {
        per[i]->disPlay(); //(*(per+i))->disPlay();
    }
    for (int i = 0; i < size; i++) {
        delete per[i];  //n번 객체 생성, n번 객체 소멸
    }
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
>> name, tel : aa 11-22
>> name, tel : bb 33-44
>> name, tel : cc 55-66
name=aa, tel=11-22
name=bb, tel=33-44
name=cc, tel=55-66
객체 소멸
객체 소멸
객체 소멸
```

동적 메모리 할당과 메모리 누수(Memory leak)





* 프로그램이 종료되면, 운영체제는 누수 메모리를 모두 힙에 반환

스마트 포인터

- 포인터처럼 동작하는 클래스 템플릿으로, **사용이 끝난 메모리를 자동으로 해제**
- 메모리 누수(memory leak)로부터 프로그램의 안전성 보장을 위해 제공
- <memory>헤더 파일 필요

- 스마트 포인터 종류
  - unique_ptr
  - shared_ptr
  - weak_ptr

스마트 포인터 종류

- unique_ptr
  - 하나의 스마트 포인터만 객체를 소유
  - 스마트 포인터가 영역을 벗어나거나 리셋 되면 참조하던 resource 해제
  - 포인터에 대한 소유권을 이전(move)할 수는 있지만, 복사(copy)나 대입(assign)과 같은 공유(share)를 불허

- shared_ptr
  - 하나의 특정 객체를 참조하는 스마트 포인터의 개수(reference count)를 참조하는 스마트 포인터
  - 참조 카운트(reference count), use_count()
    - 해당 메모리를 참조하는 포인터가 몇개인지 나타내는 값
    - shared_ptr가 추가될 때 1씩 증가, 수명이 다하면 1씩 감소, 0이 되면 메모리 자동 해제

- weak_ptr
  - 하나 이상의 shared_ptr가 가리키는 객체를 참조할 수 있지만 reference count를 늘리지않는 스마트 포인터
  - 순환 참조를 제거하기 위해 사용
    - 순환 참조란 shared_ptr가 서로 상대방을 가리키는 것으로 reference count가 0이 되지 않아 메모리가 해제되지 않는 형태.

스마트 포인터 – unique_ptr

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

int main() {
    unique_ptr<int> p(new int);  //unique_ptr 사용법1
    *p = 99;  //포인터의 초기화
    cout << "*p = " << *p << endl;

    unique_ptr<int> ap = move(p); //unique_ptr<int> ip = p; //error, move()를 사용해 포인터 이동
    cout << "*ap = " << *ap << endl; //cout << "*p = " << *p << endl; //실행시간 오류

    unique_ptr<double> sp = make_unique<double>(23.5); //unique_ptr 사용법2
    cout << "*sp = " << *sp << endl;

    auto asp = make_unique<int[]>(5); //unique_ptr 사용법3, make_unique<T>() 사용을 권고
    for (int i = 0; i < 5; i++) {
        asp[i] = 10 + i;
        cout <<asp[i] << " ";
    }
    cout << endl;
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
*p = 99
*ap = 99
*sp = 23.5
10 11 12 13 14
```

스마트 포인터 - unique_ptr 동적 객체 생성

```cpp
#include <iostream>
#include <memory>
#include <string>
using namespace std;

class Person{
    string name;
    int age;
public:
    Person() = default;
    Person(string n, int t) : name(n), age(t) { };
    ~Person() { cout << " 객체 소멸" << endl; }
    void disPlay() { cout << "name = " << name << ", age = " << age << endl; }
};

int main() {
    auto p = make_unique<Person>("unique", 20);
    p->disPlay();
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
name = unique, age = 20
객체 소멸
```

스마트 포인터 - unique_ptr 동적 객체 멤버 생성

```cpp
#include <iostream>
#include <memory>
#include <string>
using namespace std;

class Person{
    unique_ptr<string> name;
    unique_ptr<int> age;

public:
    Person();
    Person(string n, int a);
    ~Person();
    void disPlay() const;
};

Person::~Person() {
    cout << *name << " 객체 소멸" << endl;
}
```

```cpp
Person::Person() : Person("null", 0) {}

Person::Person(string n, int a) :
name{make_unique<string>(n)}, age{make_unique<int>(a)} { };
/*
Person::Person(string n, int a) {
    name = make_unique<string>(n);
    age = make_unique<int>(a);
}
*/
void Person::disPlay() const {
    cout << "name = " << *name << ", age = " << *age << endl;
}

int main() {
    unique_ptr<Person> p=make_unique<Person>("java", 30);
    p->disPlay();
    return 0;
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
name = java, age = 30
java 객체 소멸
```

스마트 포인터 - shared_ptr, use_count()

```cpp
#include <memory>
class Person{
    string name;
    int age;
public:
    Person() = default;
    Person(string n, int a) : name(n), age(a) { };
    ~Person() { cout << "메모리 해제" << endl; };
    void disPlay() { cout << "name = " << name << ", age = " << age << endl; }
};
void show(shared_ptr<Person> sp) {
    sp->disPlay();
    cout << "show().sp.use_count() : " << sp.use_count() << endl;
}
int main() {
    auto sp1 = make_shared<Person>("unique", 17);
    show(sp1);
    cout << "sp1.use_count() : " << sp1.use_count() << endl;

    shared_ptr<Person> sp2 = sp1; // 또는 auto sp2=sp1;
    show(sp2);
    cout << "sp1.use_count() : " << sp1.use_count() << endl;
    cout << "sp2.use_count() : " << sp2.use_count() << endl;
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a

name = unique, age = 17
show().sp.use_count() : 2
sp1.use_count() : 1

name = unique, age = 17
show().sp.use_count() : 3
sp1.use_count() : 2
sp2.use_count() : 2
메모리 해제
```

스마트 포인터 - shared_ptr 메모리 leak

```cpp
#include <memory>
class Person{
    string name; int age;
public:
    Person(string n, int a) : name(n), age(a) { };
    ~Person() { cout << "메모리 해제" << endl; };
    void disPlay() { cout << "name = " << name << ", age = " << age << endl; }

    std::shared_ptr<Person> per; //해결방법 weak_ptr
};
void show(shared_ptr<Person> sp) {
    sp->disPlay();
    cout << "show().sp.use_count() : " << sp.use_count() << endl;
}
int main() {
    auto sp1 = make_shared<Person>("unique", 17);
    show(sp1);
    cout << "sp1.use_count() : " << sp1.use_count() << endl;

    sp1->per = sp1;
    show(sp1->per);
    cout << "sp1.use_count() : " << sp1.use_count() << endl;
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a

name = unique, age = 17
show().sp.use_count() : 2
sp1.use_count() : 1

name = unique, age = 17
show().sp.use_count() : 3
sp1.use_count() : 2
```

c & c++ 동적 메모리 할당

```
#include <stdlib.h>
#include <iostream>
using namespace std;

class Apple {
public:
    Apple() { cout << "yummy apple" << endl; }
    ~Apple() { cout << "finished apple" << endl; }
};
int main() {
    //heap int : c style
    int *a = (int *)malloc(sizeof(int));
    *a = 100;
    free(a);

    //heap int array : c style
    int *b = (int *)malloc(sizeof(int)*3);
    b[0] = 100;
    free(b);

    //heap Apple : c style
    Apple *c = (Apple *)malloc(sizeof(Apple));
    free(c);

    //heap Apple array : c style
    Apple *d = (Apple *)malloc(sizeof(Apple)*3);
    free(d);
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
```

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


class Apple {
public:
    Apple() { cout << "yummy apple" << endl; }
    ~Apple() { cout << "finished apple" << endl; }
};
int main() {
    //heap int : c++ style
    int *a = new int;
    *a = 100;
    delete a;

    //heap int array : c++ style
    int *b = new int[3];
    b[0] = 100;
    delete [] b;

    //heap Apple : c++ style
    Apple *c = new Apple;
    delete c;

    //heap Apple array : c++ style
    Apple *d = new Apple[3];
    delete [] d;
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
yummy apple
finished apple
yummy apple
yummy apple
yummy apple
finished apple
finished apple
finished apple
```

c++ & safer c++ 동적 메모리 할당

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


class Apple {
public:
    Apple() { cout << "yummy apple" << endl; }
    ~Apple() { cout << "finished apple" << endl; }
};
int main() {
    //heap int : c++ style
    int *a = new int;
    *a = 100;
    delete a;

    //heap int array : c++ style
    int *b = new int[3];
    b[0] = 100;
    delete [] b;

    //heap Apple : c++ style
    Apple *c = new Apple;
    delete c;

    //heap Apple array : c++ style
    Apple *d = new Apple[3];
    delete [] d;
}
```

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

class Apple {
public:
    Apple() { cout << "  yummy apple" << endl; }
    ~Apple() { cout << "  finished apple" << endl; }
};
int main() {
    cout << "== heap int : c++ style ==" << endl;
    unique_ptr<int> a = make_unique<int>();
    *a = 100;

    cout << "== heap int array : vector<int> ==" << endl;
    vector<int> b(3);
    b[0] = 55; //or b.at(0)

    cout << "== heap Apple : memory leak 해결 ==" << endl;
    unique_ptr<Apple> c = make_unique<Apple>();

    cout << "== heap Apple array : memory leak 해결 ==" << endl;
    int count;
    cout << "  vector size? ";
    cin >> count;
    vector<Apple> d(count);
}
```

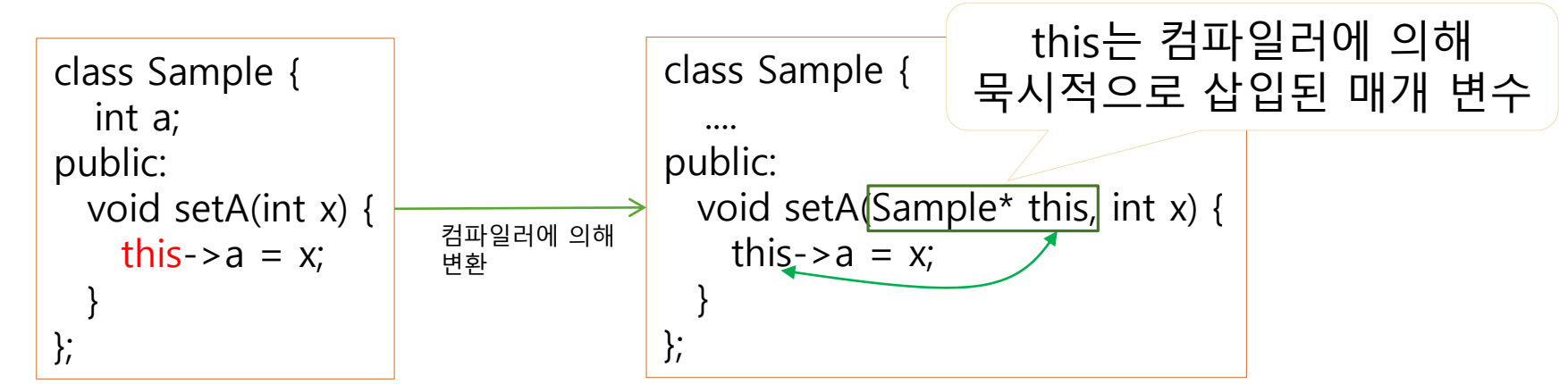
```
PS C:\yanges\lecture\lecture_src\cpp> g++ safer.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
== heap int : c++ style ==
== heap int array : vector<int> : c++ style ==
== heap Apple : memory leak 해결 ==
  yummy apple
== heap Apple array : memory leak 해결 ==
  vector size? 2
  yummy apple
  yummy apple
  finished apple
  finished apple
  finished apple
```

this 포인터

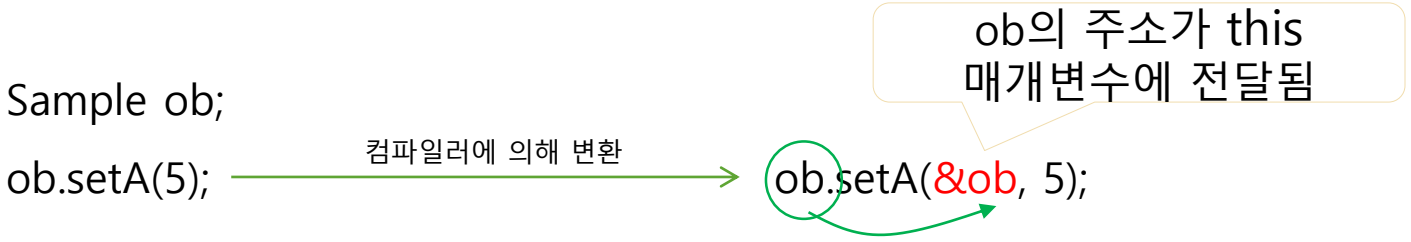
- this
  - 포인터, 객체 자신 포인터
  - 클래스의 멤버 함수 내에서만 사용
  - 개발자가 선언하는 변수가 아니고, 컴파일러가 선언한 변수
    - 컴파일러에 의해 묵시적으로 멤버 함수에 삽입/선언되는 매개 변수

```
class Circle {
  int radius;

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

this 포인터의 실체 – 컴파일러에서 처리



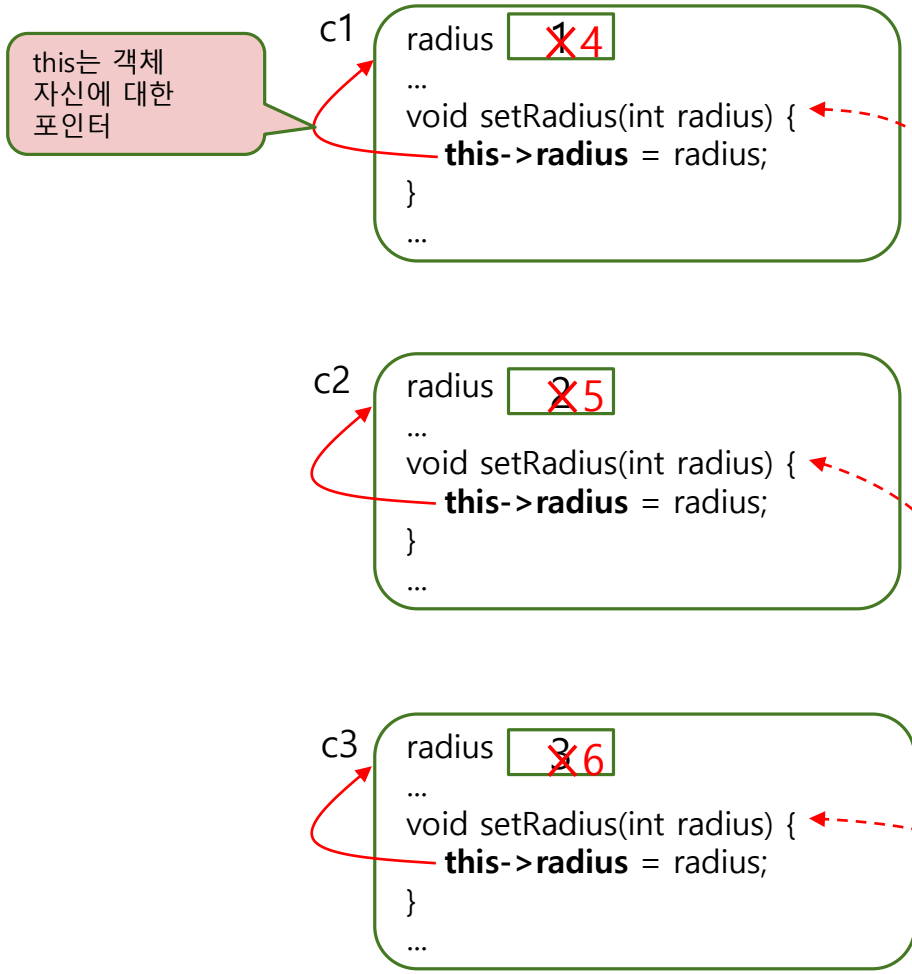
(a) 개발자가 작성한 클래스 (b) 컴파일러에 의해 변환된 클래스



(c) 객체의 멤버 함수를 호출하는 코드의 변환

this와 객체

* 각 객체 속의 this는 다른 객체의 this와 다름

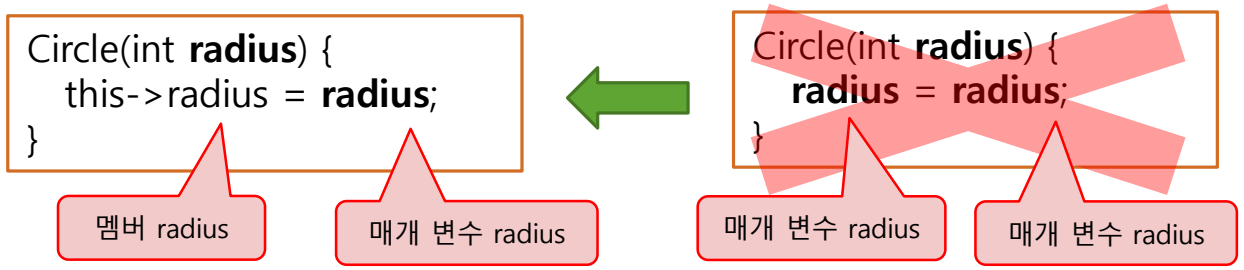



```
class Circle {
   int radius;
public:
   Circle() {
      this->radius=1;
   }
   Circle(int radius) {
      this->radius = radius;
   }
   void setRadius(int radius) {
      this->radius = radius;
   }
};
```

```
int main() {
   Circle c1;
   Circle c2(2);
   Circle c3(3);

   c1.setRadius(4);
   c2.setRadius(5);
   c3.setRadius(6);
}
```

this가 필요한 경우

- 매개변수의 이름과 멤버 변수의 이름이 같은 경우



- 멤버 함수가 객체 자신의 주소를 리턴할 때
  - 연산자 중복에서 자주 사용

```
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 때문

학습 정리 (1)

- 객체 포인터
- nullptr

- 객체 배열의 생성 및 소멸 : Circle cirarr[2]
- 객체 배열의 초기화 : Circle circleArray[3] = { Circle(10), Circle(20), Circle() };

- 동적 메모리 할당/반환
  - int *pInt = new int; delete pInt;
  - Circle *pCircle = new Circle(); delete pCircle;
  - int *p = new int [5]; delete [] p;

- 동적 할당 메모리의 초기화 및 반환
  - int *p = new int[4] {3, 4, 5, 6}; delete [] p
  - Circle *pCircle = new Circle[3]; delete [] pCircle;

학습 정리 (2)

- 클래스 멤버의 동적 생성

```
class Dog {
        string *name;
public:
         Dog(string n) : name{ new string{n} } {}
        ~Dog() { delete name; }
};
```

- 객체 포인터 배열 Dog *d[3];

- this 포인터

```
class Sample {
public:
        void setA(Sample* this, int x) { }
};
```

- this의 제약 사항

학습 정리 (3)

- 스마트 포인터 종류
  - unique_ptr
  - shared_ptr : 참조 카운트(reference count), use_count()
  - weak_ptr

- shared_ptr
  - auto sp1 = make_shared<Person>();
  - void show(shared_ptr<Person> sp) //show(sp1);
  - 순환구조 : weak_ptr로 해결

- shared_ptr과 메모리 leak

Q & A

- "C++ 객체포인터와 동적생성"에 대한 학습이 모두 끝났습니다.

- 새로운 내용이 많았습니다. 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.

- cpp_04_객체포인터와동적생성 _ex.pdf 에 확인 학습 문제들을 담았습니다.
- 이론 학습을 완료한 후 확인 학습 문제들로 학습 내용을 점검 하시기 바랍니다.

- 퀴즈와 과제가 출제되었습니다. 마감시간에 늦지 않도록 주의해 주세요.

- 수고하셨습니다.^^