8장 포인터와 동적 객체 생성

2020. 10. 8

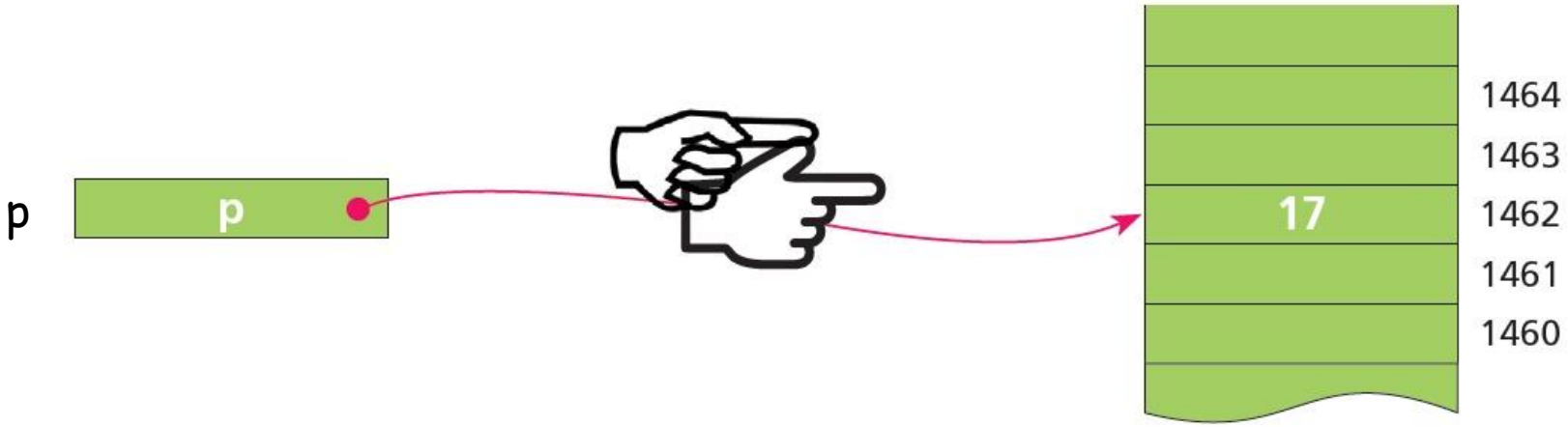
순천향대학교 컴퓨터 공학과

내용

- 포인터 리뷰
- 동적 메모리 할당 리뷰
- 객체 포인터
- 동적 객체 생성 및 반환
- this 포인터
- 스마트 포인터

포인터란?

- 포인터 타입 변수이며, 그 값은 null이나 주소임



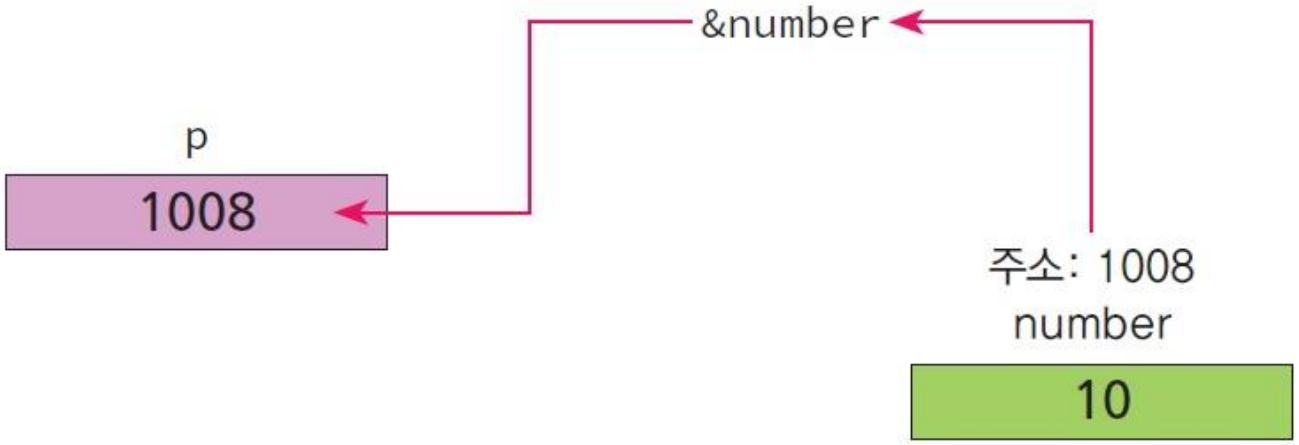
포인터 선언

문법 8.1 포인터 선언


```
int *p1;
double *p2;
char *p3;
short *p4;
```

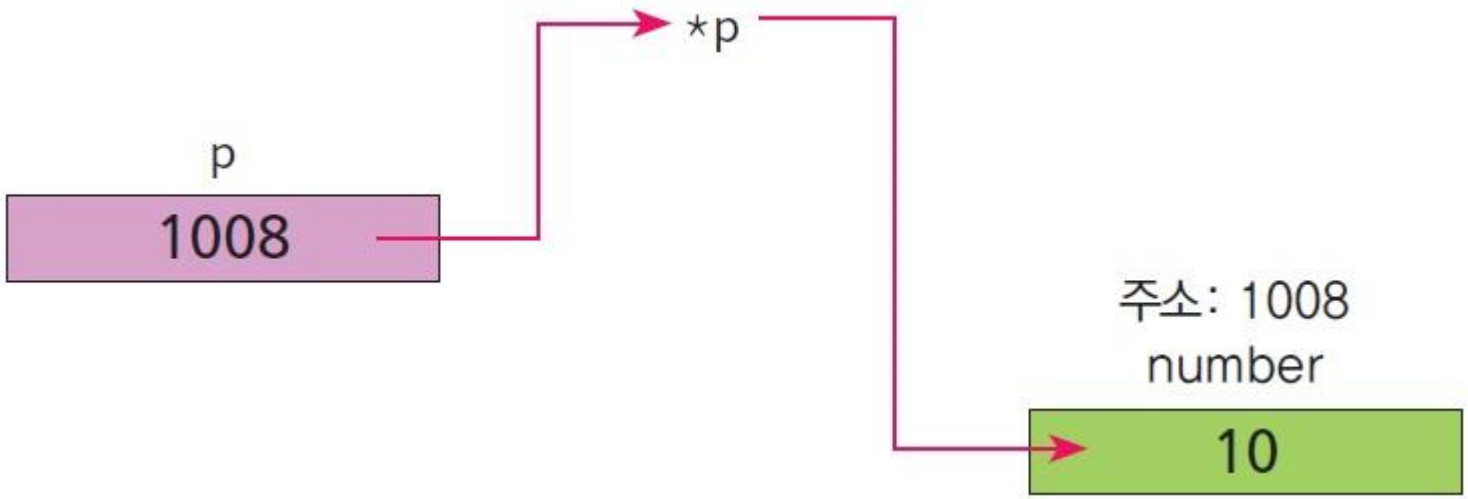
주소 연산자 &

```
int number = 10;
int *p;

p = &number;
```



간접참조 연산자 *

```
int number = 10;
int *p;

p = &number;
cout << *p << endl;
```



예제

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

int main()

{
        int number = 10;
        int *p = &number;

        cout << *p << endl;

        return 0;
}
```

널 포인터

- 포인터 선언시 널 포인터로 초기화하는 것이 필요함

- 다음 코드에서 문제는?

```
int *p;

*p = 100;
```

널 포인터 (2)

- 널 포인터 초기화
  - int *p = nullptr; // C++11 이후
  - int *p = NULL; // NULL은 정수 0임에 유의

```
void f(int i) {
        cout <<“f(int)”<<endl;
}

void f(char *p) {
        cout<<“f(char *)”<<endl;
}

fun(NULL); // 출력은?
fun(nullptr); //  출력은?
```

동적 메모리 할당

- **동적 메모리 할당(dynamic memory allocation)**이란 프로그램이 실행 도중에 동적으로 메모리를 할당받는 것을 말한다.

동적 메모리 사용 절차

- 메모리 할당 및 반환을 사용자가 요청해야 함

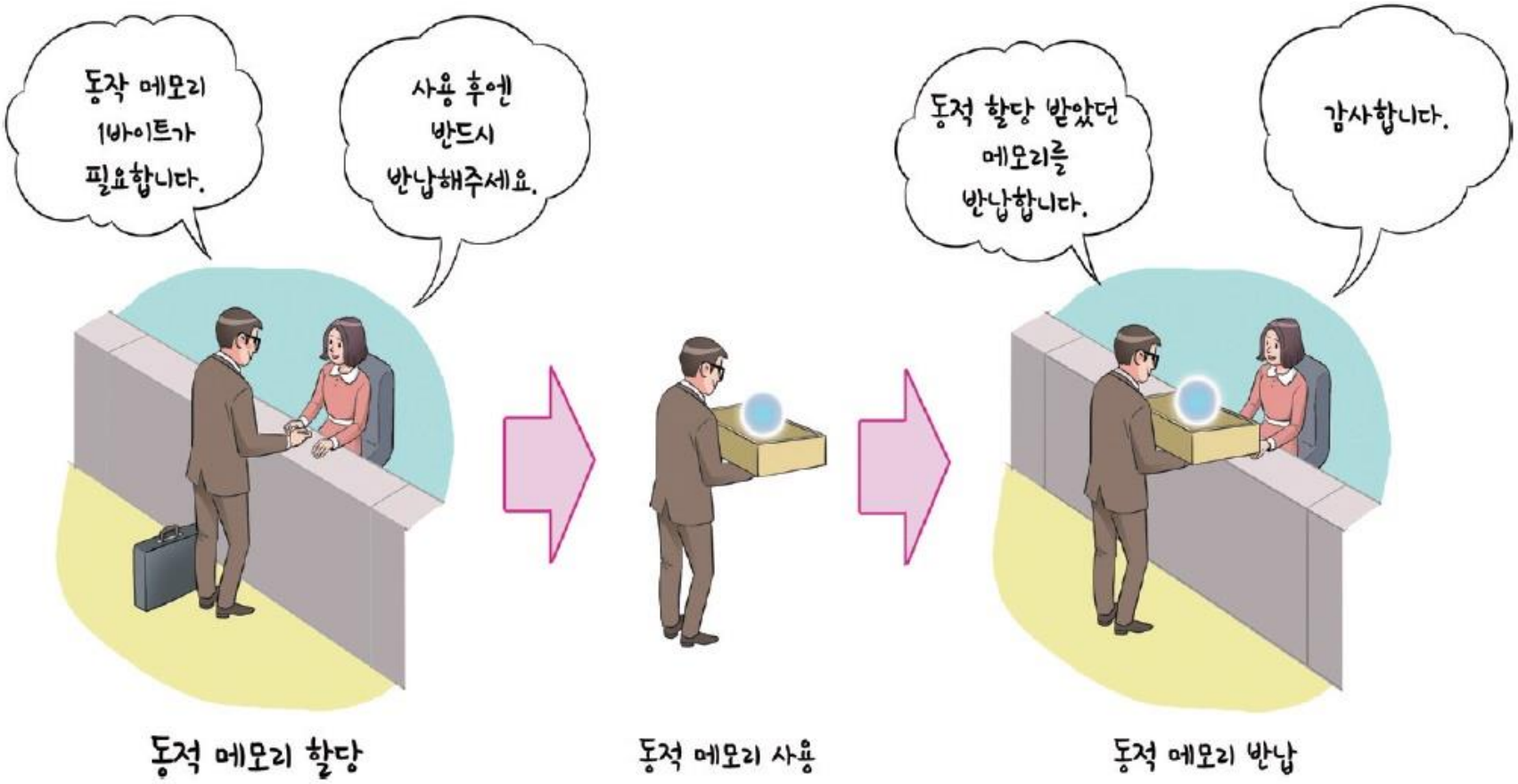


그림 8.1 동적 메모리 사용 단계

동적 메모리 할당 및 반환

- C 의 동적 메모리 할당/반환
  - malloc()/free() 라이브러리 함수 사용
- C++의 동적 메모리 할당/반환
  - new 연산자
    - 객체의 동적 생성 - 힙 메모리로부터 객체를 위한 메모리 할당 요청
    - 객체 할당 시 생성자 호출
    - 기본 타입 메모리 할당, 배열 할당, 객체 할당, 객체 배열 할당
  - delete 연산자
    - new로 할당 받은 메모리 반환
    - 객체의 동적 소멸 - 소멸자 호출 뒤 객체를 힙에 반환

동적 메모리 할당

문법 8.3 new 연산자

동적 메모리의 타입

```
p = new T;
p = new T[N];
p = new T[N] {initializer1, ..., initializerN};
```

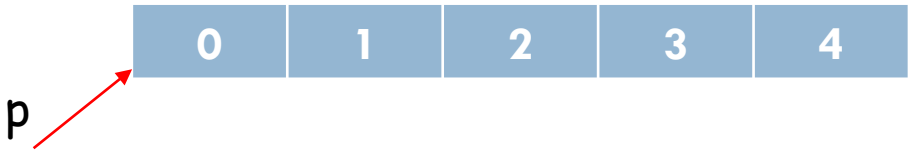
동적 메모리를 가리키는 포인터

동적 메모리의 개수

동적 메모리의 초기값들

```
int *p;
p = new int[5];
```

```
int *p = new int[5] { 0, 1, 2, 3, 4 };
```



동적 메모리 해제

문법 8.4 delete 연산자

```
delete p;
delete [] p;
```

예제

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

int main() {
        int *ptr;

        srand(time(NULL));
        ptr = new int[10];

        for (int i = 0; i<10; i++)
                ptr[i] = rand();

        for (int i = 0; i<10; i++)
                cout << ptr[i] << " ";

        cout << endl;
        delete[] ptr;
        return 0;
}
```

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

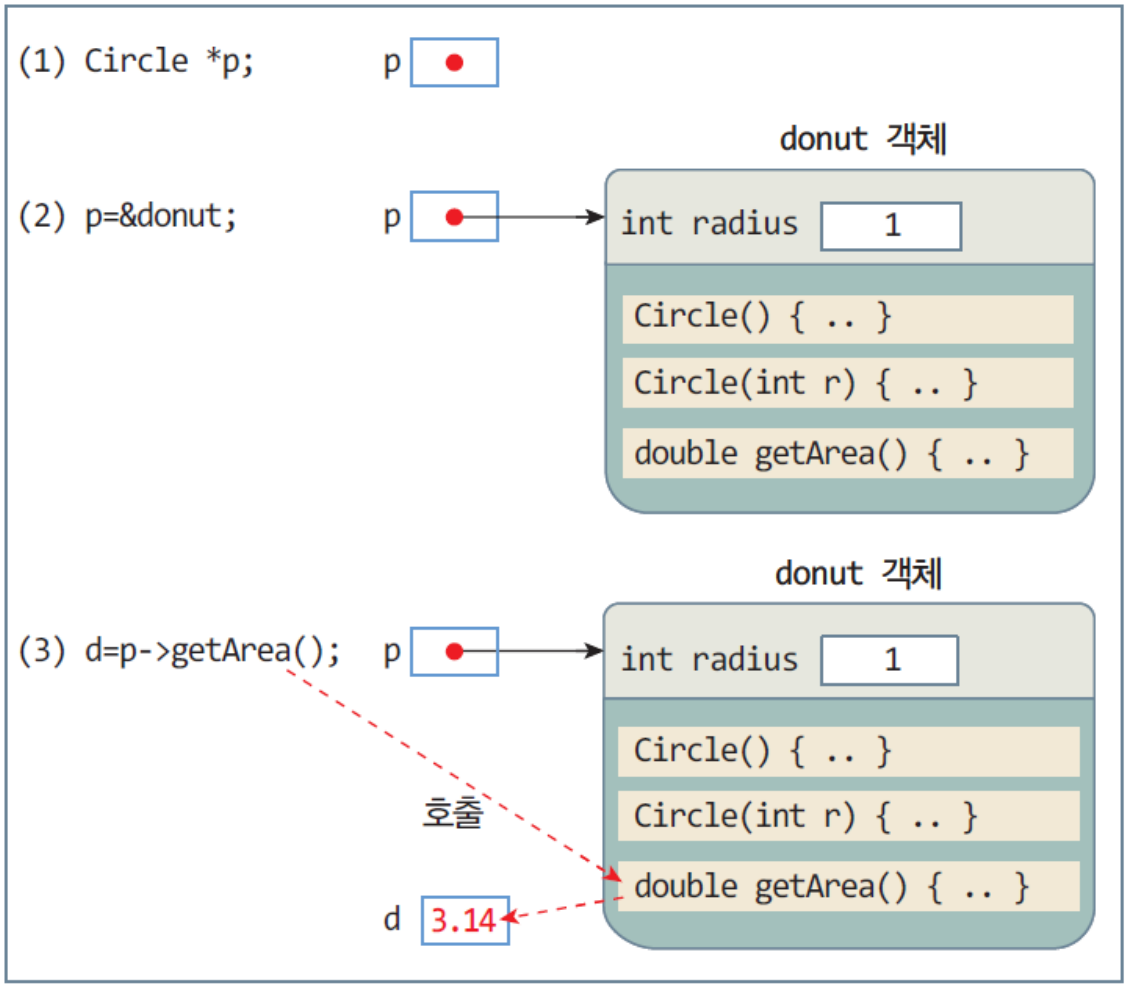
```
Circle donut;
double d = donut.getArea();

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

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)
```



예제: 객체 포인터


```
#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;
   cout << (*p).getArea() <<endl;

   p = &pizza;
   cout << p->getArea() << endl;
   cout << (*p).getArea() << endl;
}
```

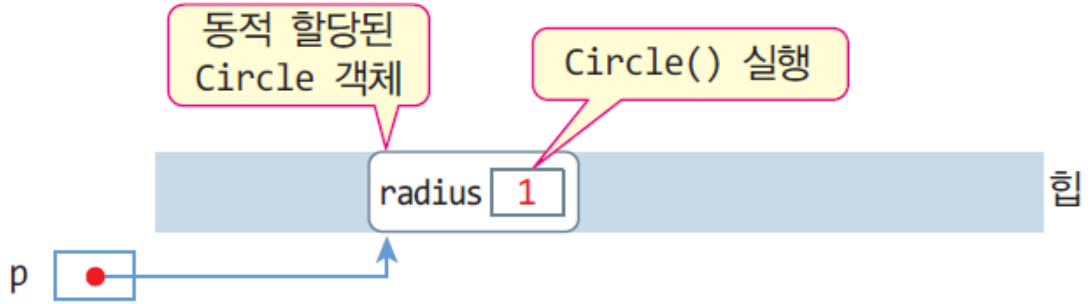
객체의 동적 생성 및 반환


```
클래스이름 *포인터변수 = new 클래스이름; // 힙 객체 생성

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

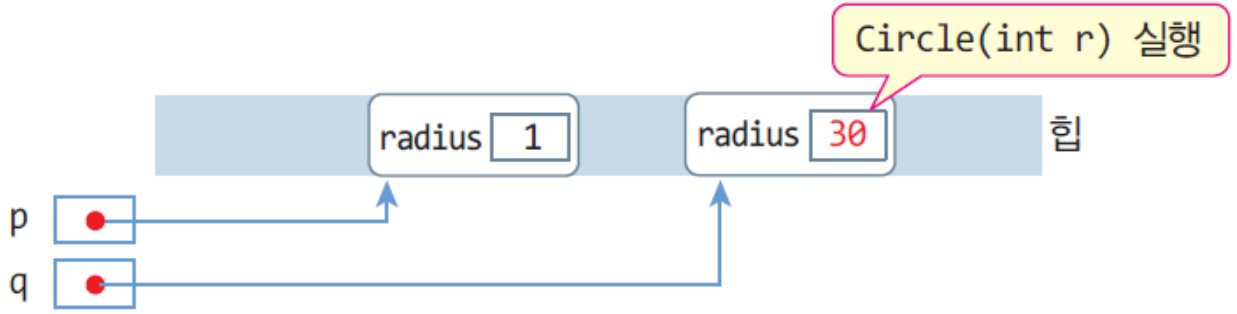
```
Circle *p = new Circle;
Circle *q = new Circle(30);
...
delete p;
delete q;
```

예제: 객체의 동적 생성 및 반환

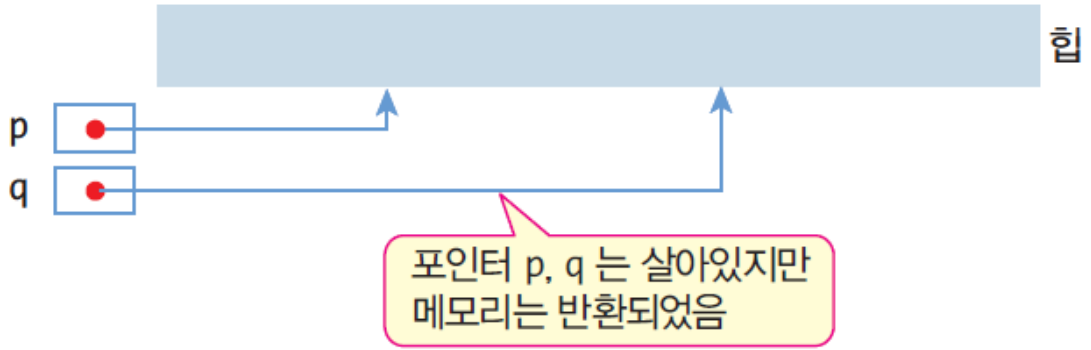

```
(1) Circle *p = new Circle;
```




```
(2) Circle *q = new Circle(30);
```




```
(3) delete p;
    delete q;
```



```
int main() {
   Circle *p, *q;

   p = new Circle;
   q = new Circle(30);
   cout << p->getArea() << endl << q->getArea() << endl;
   delete p;
   delete q;
}
```

힙 객체는 생성한 순서에 관계 없이 원하는 순서대로 삭제 가능

```
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;
}
```

```
class Dog {
private:
  int *pWeight;
  int *pAge;

public:
  Dog() {
    pAge = new int{1};
    pWeight = new int{10};
  }
  ~Dog() {
    delete pAge;
    delete pWeight;
  }

  int getAge() { return *pAge; }
  int getWeight() { return *pWeight; }

  void setAge(int age) { *pAge = age; }
  void setWeight(int weight) { *pWeight = weight; }
};
```

```
int main() {
  Dog * pDog = new Dog;

  cout << "강아지의 나이: " << pDog->getAge() << endl;

  pDog->setAge(5);
  cout << "강아지의 나이: " << pDog->getAge() << endl;

  delete pDog;
  return 0;
}
```

힙 객체 배열

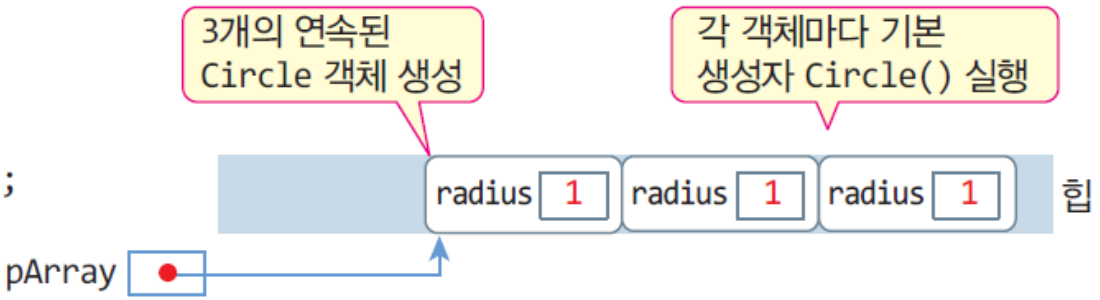
- 생성 및 반환:

```
클래스이름 *포인터변수 = new 클래스이름 [배열 크기];

delete [] 포인터변수; // 포인터변수가 가리키는 객체 배열을 반환
```

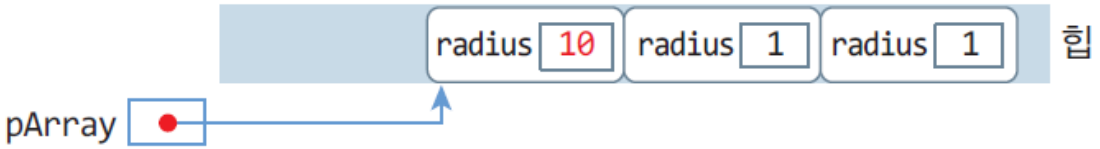
```
Circle *p = new Circle[3];

p[0].setRadius(10);
...
delete [] p;
```

예제

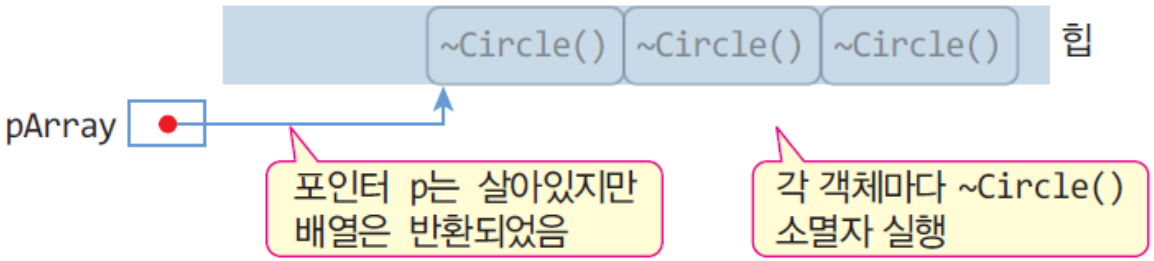

```
(1) Circle *pArray = new Circle[3];
```




```
(2) pArray[0].setRadius(10);
```




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



```
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;
}
```

각 원소 객체의 기본 생성자 Circle() 실행

```
int main() {
   Circle *pArray = new Circle [3];

   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;

   for(int i=0; i<3; i++) {
      cout << p->getArea() << '₩n';
      p++;
   }

   delete [] pArray;
}
```

소멸자는 생성의 역순으로 실행
pArray[2], pArray[1], Parray[0]의 순서로 소멸

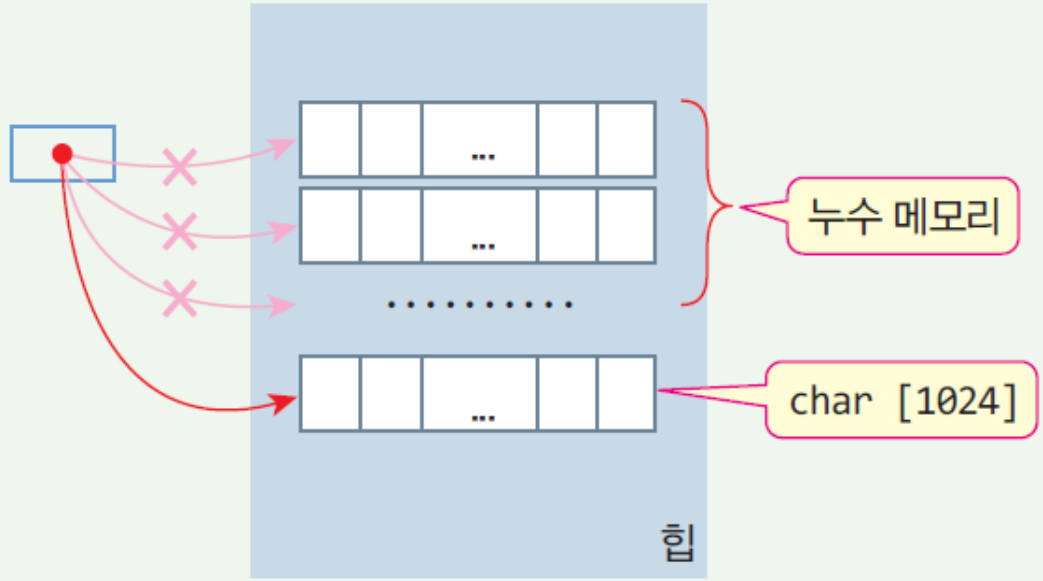
다음 코드의 문제점은?


```
char *p;

for (int i=0; i<1000000; i++) {
   p = new char[1024];
}
```

동적 메모리 할당과 메모리 누수

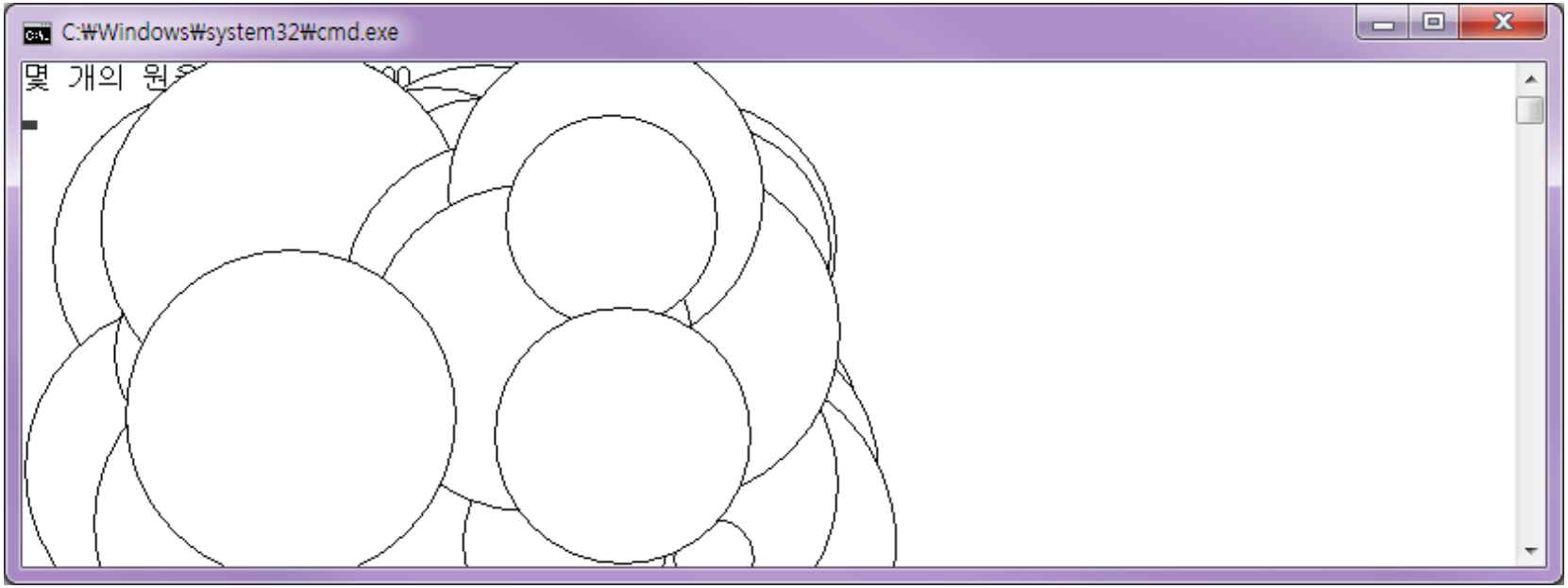

```
char *p;
for(int i=0; i<1000000; i++) {
    p = new char [1024];
}
```

p는 새로 할당 받는 1024 바이트의 메모리를 가리키므로 이전에 할당 받은 메모리의 누수 발생



Lab: 동적 원 생성

- 사용자가 입력한 개수만큼의 원을 동적으로 생성하여서 화면에 그리는 프로그램을 작성해 보자.



Lab: solution

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

class Circle {
public:
        int x, y, radius; // 원의 중심과 반지름
        string color;     // 원의 색상
        void draw();
};

void Circle::draw() {
 // 원을 화면에 그리는 함수
        HDC hdc = GetWindowDC(GetForegroundWindow());
        Ellipse(hdc, x - radius, y - radius, x + radius, y + radius);
}
```

```
int main() {
  int n;
  Circle *p;

  cout << "몇 개의 원? ";
  cin >> n;
  p = new Circle[n];

  for (int i = 0; i < n; i++) {
      p[i].x = 100 + rand() % 300;
      p[i].y = 100 + rand() % 200;
      p[i].radius = rand() % 100;
      p[i].draw();
  }
  delete[] p;

  return 0;
}
```

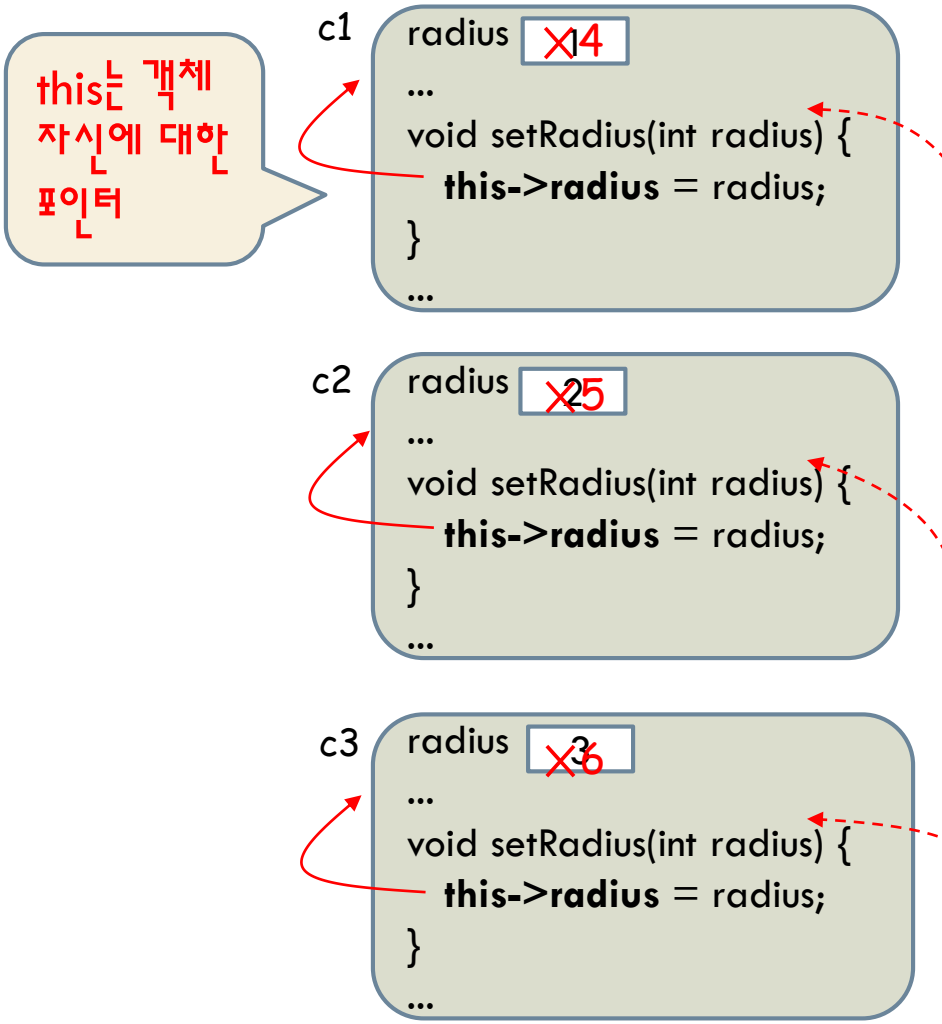
this 포인터

- this
  - 객체 자신에 대한 포인터
  - 클래스의 멤버 함수 내에서만 사용
  - 멤버 함수에 컴파일러에 의해 묵시적으로 삽입

```
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



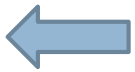

```
class Circle {
  int radius;
public:
  Circle() {
    this->radius=1;
  }
  Circle(int radius) {
    this->radius = radius;
  }
  void setRadius(int radius)
{
    this->radius = radius;
  }
};
```

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

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

this가 필요한 경우

- 매개변수와 멤버 변수의 이름이 같은 경우

```
Circle(int radius) {
  this->radius =
radius;
}
```



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

- 멤버 함수가 객체 자신의 주소를 리턴할 때
  - 연산자 중복 시 필요

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

const 포인터

- 다음 선언문의 차이는?

```
const int *p1;                  // ①
int * const p2;                 // ②
const int * const p3;           // ③
```

const 포인터

```
const int *p1;              // ①
int * const p2;             // ②
const int * const p3;       // ③
```

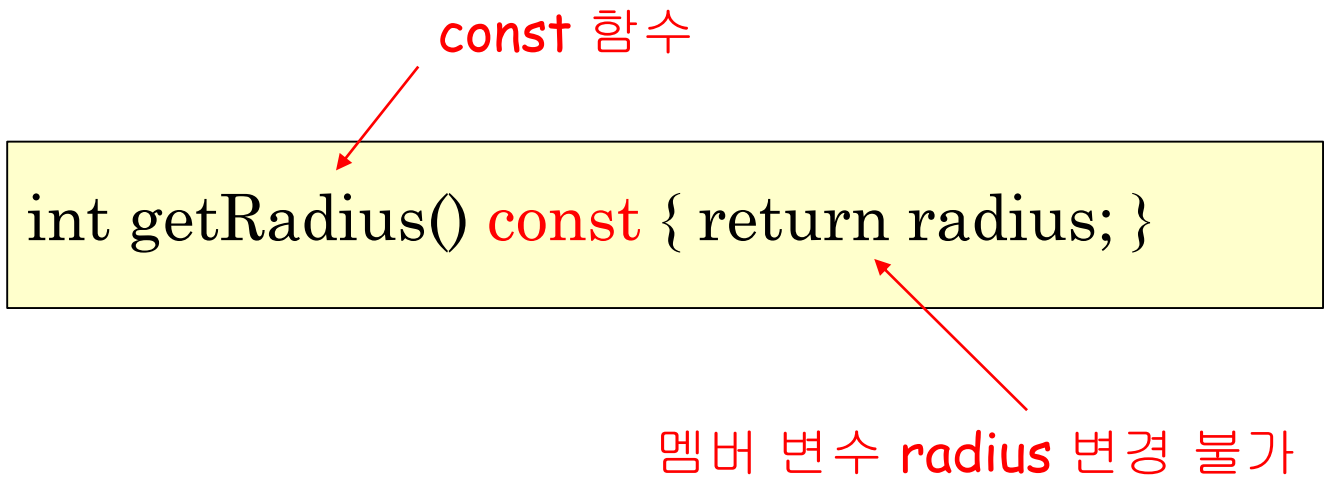
① p1은 변경되지 않는 정수 상수를 가리키는 포인터이다. 이 포인터를 통하여 참조되는 값은 변경이 불가능하다.

② p2는 정수에 대한 상수 포인터이다. 정수는 변경될 수 있지만 p2는 다른 것을 가리킬 수 없다.

③ p3는 상수에 대한 상수 포인터이다. 포인터가 가리키는 값도 변경이 불가능하고 포인터 p3도 다른 것을 가리키게끔 변경될 수 없다.

const 멤버 함수

- 멤버 함수를 const로 정의하면 함수 안에서 멤버 변수 변경 불가



- const 객체를 가리키는 포인터를 정의하면 이 포인터로 호출할 수 있는 함수는 const 함수 뿐이다.

예제

```
int main()
{
        Circle* p = new Circle();
        const Circle *pConstObj = new Circle();
        Circle *const pConstPtr = new Circle();

        cout << "pRect->radius: " << p->getRadius() << endl;
        cout << "pConstObj->radius: " << pConstObj->getRadius() << endl;
        cout << "pConstPtr->radius: " <<
                pConstPtr->getRadius() << endl<<endl;

        p->setRadius(30);
        pConstObj->setRadius(30);
        pConstPtr->setRadius(30);

        cout << "pRect->radius: " << p->getRadius() << endl;
        cout << "pConstObj->radius: " << pConstObj->getRadius() << endl;
        cout << "pConstPtr->radius: " << pConstPtr->getRadius() << endl;
        return 0;
}
```

스마트 포인터

- 스마트 포인터(C++11에서 도입)를 사용하면 프로그래머가 동적 메모리 할당 후에 잊어버려도 자동으로 동적 메모리가 삭제된다.
  - unique_ptr을 사용하여 스마트 포인터 표현
  - 이 포인터는 기본 포인터를 감싸서 객체로 구성
  - 그 객체에 소멸자가 포함되어서, 객체 소멸시 할당된 공간도 해제
  - 스마트 포인터는 자동으로 nullptr로 초기화된다.

```
int main() {
  unique_ptr<int> p(new int); // 포인터를 객체로 포장하여 p로 참조

  *p = 99;
}
```

예제: 스마트 포인터

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

int main()
{
	unique_ptr<int[]> buf(new int[10]);

	for (int i = 0; i<10; i++) {
		buf[i] = i;
	}

	for (int i = 0; i<10; i++) {
		cout << buf[i] << " ";
	}
	cout << endl;
	return 0;
}
```