함수
참조
복사생성자

함수의 인자 전달 방식

값에 의한 호출

```
void swap(int a, int b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int m=2, n=9;
  swap(m, n);
  cout << m << ' ' << n;
}
```

주소에 의한 호출

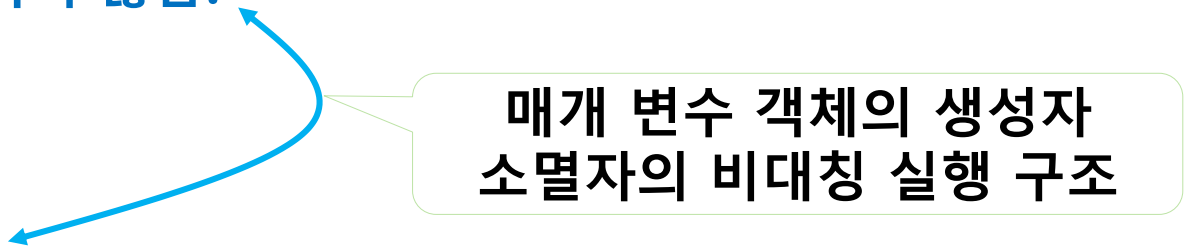
```
void swap( int *a, int *b ) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int m=2, n=9;
  swap(&m, &n);
  cout << m << ' ' << n;
}
```

‘값에 의한 호출’로 객체 전달

- 함수를 호출하는 쪽에서 객체 전달
  - 객체 이름만 사용.

- 함수의 매개 변수 객체 생성
  - 매개 변수 객체의 공간이 스택에 할당.
  - 호출하는 쪽의 객체가 매개 변수 객체에 그대로 복사됨.
  - **매개 변수 객체의 생성자는 호출되지 않음.**

- 함수 종료
  - **매개 변수 객체의 소멸자 호출.**

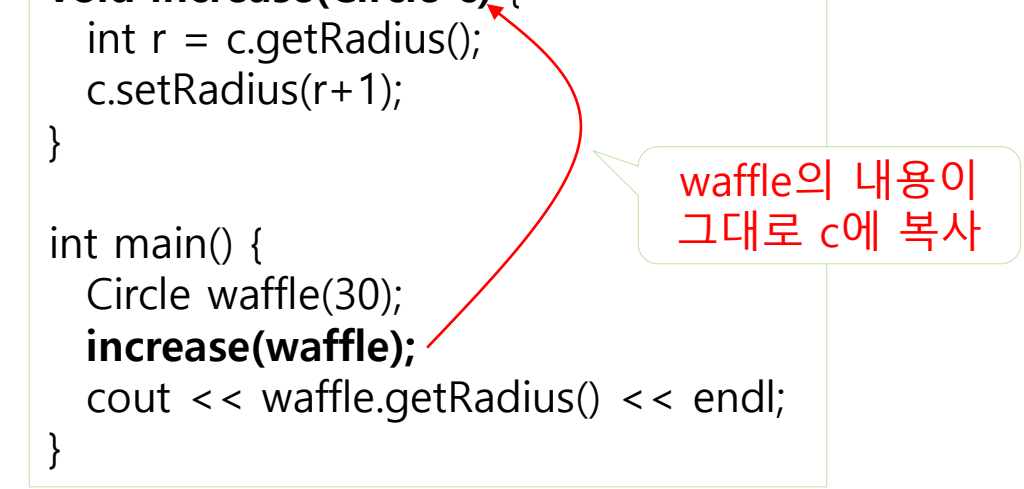


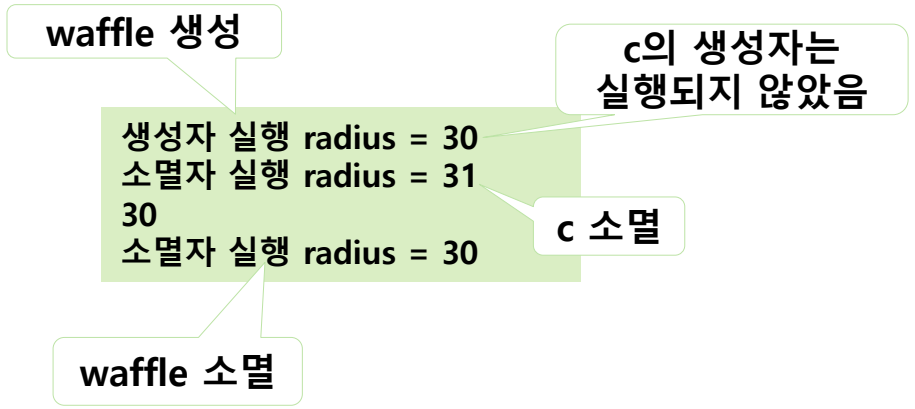
- *값에 의한 호출 시 매개 변수 객체의 생성자가 실행되지 않는 이유?*
  - 호출되는 순간의 실 인자 객체 상태를 매개 변수 객체에 그대로 전달하기 위함.

'값에 의한 호출'시 매개 변수의 생성자는 실행되지 않음

```
class Circle {
private:
   int radius;
public:
   Circle();
   Circle(int r);
   ~Circle();
   double getArea()  { return 3.14*radius*radius; }
   int getRadius() { return radius; }
   void setRadius(int radius) { this->radius = radius; }
};

Circle::Circle() : Circle(1) { }
Circle::Circle(int radius) {
   this->radius = radius;
   cout << "생성자 실행 radius = "  << radius << endl;
}
Circle::~Circle() {
   cout << "소멸자 실행 radius = " << radius << endl;
}
```

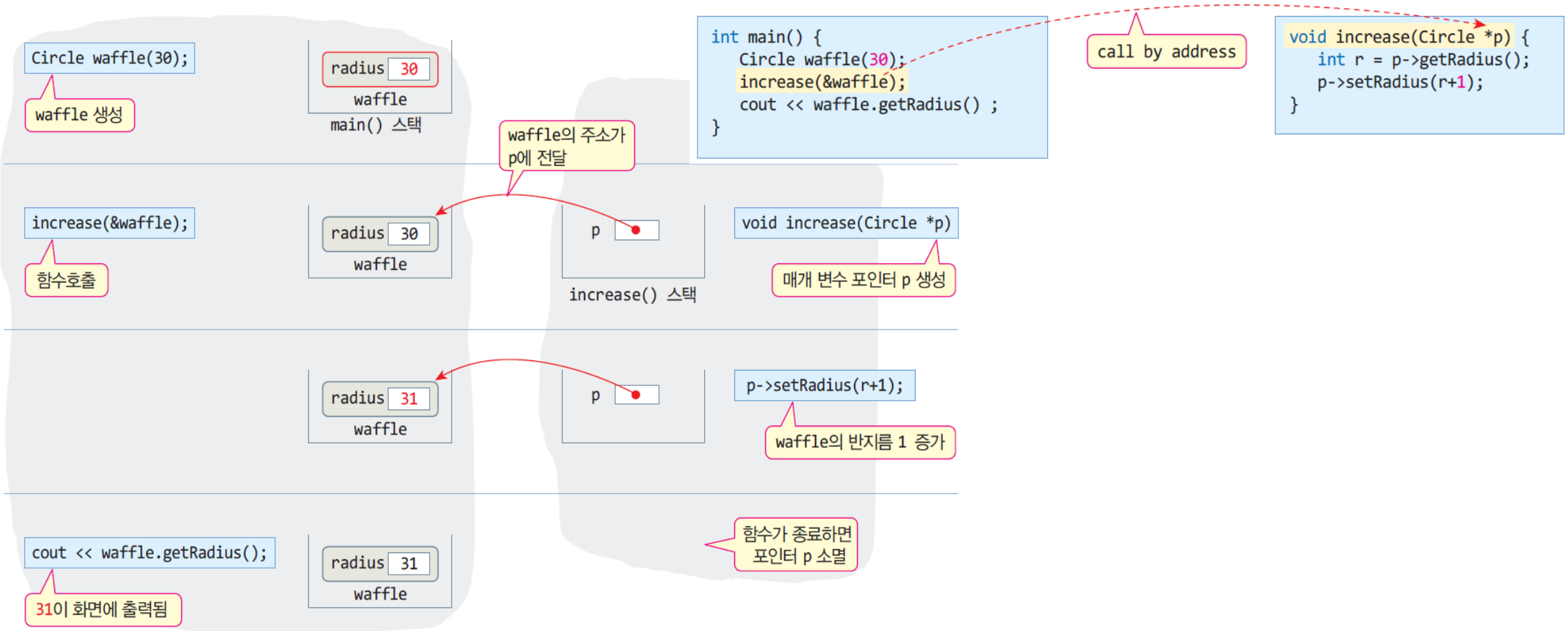
```
void increase(Circle c) {
   int r = c.getRadius();
   c.setRadius(r+1);
}

int main() {
   Circle waffle(30);
   increase(waffle);
   cout << waffle.getRadius() << endl;
}
```





함수에 객체 전달 – '주소에 의한 호출'

- 함수 호출 시 객체의 주소 전달
  - 함수의 매개 변수는 객체에 대한 포인터 변수로 선언.
  - 함수 호출 시 생성자/소멸자가 실행되지 않는 구조.



함수에 객체 전달 – '주소에 의한 호출' 실행 예

```
void increase(Circle c) {
    int r = c.getRadius();
    c.setRadius(r+1);
}
int main() {
    Circle waffle(30);
    increase(waffle);
    cout << waffle.getRadius() << endl;
}
```

```
생성자 실행 radius = 30
소멸자 실행 radius = 31
30
소멸자 실행 radius = 30
```

```
void increase(Circle *c) {
    int r = c->getRadius();
    c->setRadius(r+1);
}
int main() {
    Circle waffle(30);
    increase(&waffle);
    cout << waffle.getRadius() << endl;
}
```

```
생성자 실행 radius = 30
31
소멸자 실행 radius = 31
```

객체 치환 및 객체 리턴

- 객체 치환
  - 동일한 클래스 타입의 객체끼리 치환 가능.
  - 객체의 모든 데이터가 비트 단위로 복사됨.

```
Circle c1(5);
Circle c2(30);
c1 = c2;  // c2 객체를 c1 객체에 비트 단위 복사. c1의 반지름 30됨
```

  - 치환된 두 객체는 현재 내용물만 같을 뿐 독립적인 공간 유지.

- 객체 리턴

```
Circle getCircle() {
        Circle tmp(30);
        return tmp;      // 객체 tmp 리턴
}
```

```
Circle c;               // c의 반지름 1
c = getCircle();        // tmp 객체의 복사본이 c에 치환. c의 반지름은 30이 됨
```

참조란?

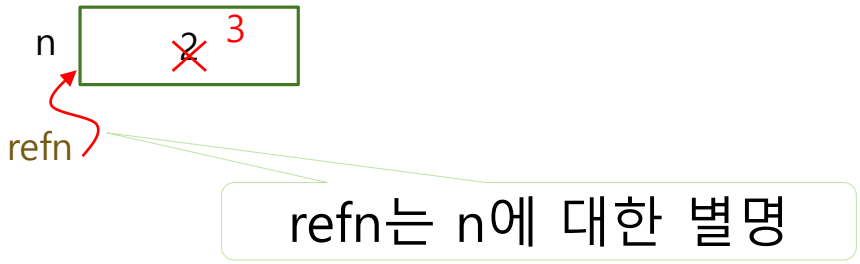
- 참조(reference)란 가리킨다는 뜻으로, 이미 존재하는 객체나 변수에 대한 별명.
- 참조 활용
  - **참조 변수**
  - **참조에 의한 호출**
  - **참조 리턴**
- 참조 변수 선언
  - 참조자 & 사용
  - 이미 존재하는 변수에 대한 다른 이름(별명)을 선언.
  - 참조 변수는 이름만 생기며 **참조 변수에 새로운 공간을 할당하지 않음.**
  - **초기화로 지정된 원본 변수의 공간 공유.**
  - **선언 시 반드시 원본 변수로 초기화** - **초기화 없으면 오류 발생.**
  - 참조 변수의 배열은 만들 수 없음.

```
int &refn[10];  //오류
```

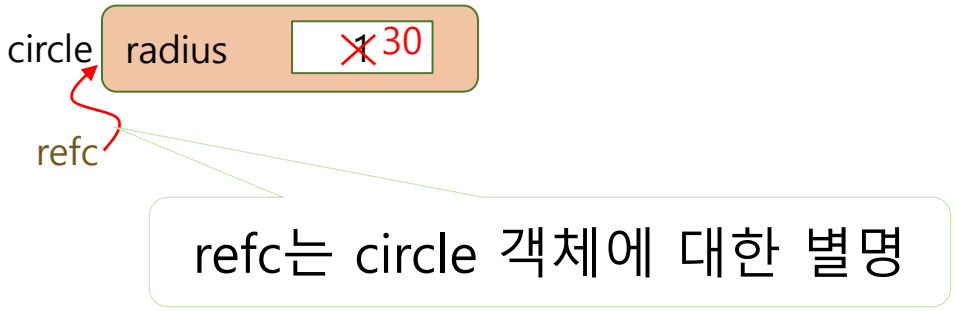
```
int &refn;  //오류
```

참조 변수 선언 및 사용 사례

```
int n = 2;
int &refn =n; // 참조 변수 refn 선언. refn은 n에 대한 별명, refn과 n은 동일한 변수
refn = 3;
```




```
Circle circle;
Circle &refc = circle; // 참조 변수 refc 선언,  refc는 circle에 대한 별명
refc.setRadius(30);     //refc->setRadius(30); 으로 하면 안 됨
```



기본 타입 변수에 대한 참조

```
int main() {
  cout << "i" << '₩t' << "n" << '₩t' << "refn" << endl;
  int i = 1;
  int n = 2;

  int &refn = n;   // 참조 변수 refn 선언. refn은 n에 대한 별명
  n = 4;
  refn++;          // refn=5, n=5
  cout << i << '₩t' << n << '₩t' << refn << endl;

  refn = i;        // refn=1, n=1
  refn++;          // refn=2, n=2
  cout << i << '₩t' << n << '₩t' << refn << endl;

  int *p = &refn;  // p는 n의 주소를 가짐, 참조에 대한 포인터 변수 선언
  *p = 20;         // refn=20, n=20
  cout << i << '₩t' << n << '₩t' << refn << endl;
}
```

객체에 대한 참조

```
class Circle {
  int radius;
public:
  Circle() : Circle(1) { }
  Circle(int radius) { this->radius = radius; }
  void setRadius(int radius) { this->radius = radius; }
  double getArea() { return 3.14*radius*radius; }
};

int main() {
  Circle circle;
  Circle &refc = circle;  //circle 객체에 대한 참조 변수 refc  선언

  refc.setRadius(10);
  cout << refc.getArea() << " " << circle.getArea();
}
```

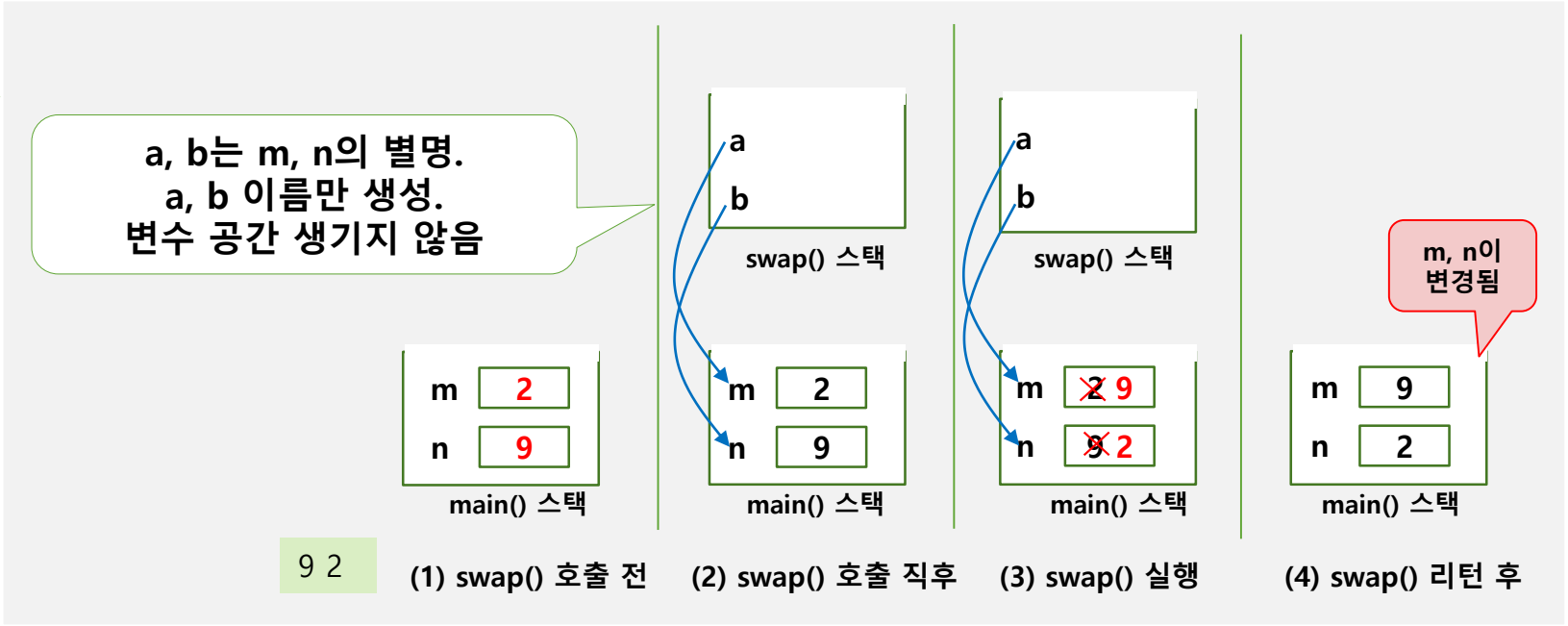
참조를 사용한 함수 호출

- 참조를 가장 많이 활용하는 사례
- call by reference라고 부름.

- 함수의 매개 변수를 참조 타입으로 선언 시
  - 참조 매개 변수(reference parameter)라고 부름.
  - 참조매개 변수의 이름만 생기고 공간이 생기지 않음.
  - 참조 매개 변수는 실 인자 변수의 공간을 공유하며 값을 참조.
  - 참조 매개 변수에 대한 조작은 실 인자 변수의 조작 효과를 가짐.

참조에 의한 함수 호출 예

```
void swap( int &a, int &b ) {
  int tmp;
  tmp = a;   a = b;   b = tmp;  //참조 매개 변수를 보통 변수처럼 사용
}
int main() {
  int m=2, n=9;
  swap(m, n);   //함수가 호출되면 m,n에 대한 참조 변수 a,b가 생김
  cout << m << ' ' << n;
}
```



참조 매개변수가 필요한 사례

- 다음 코드에 어떤 문제가 있을까?
  - average() 함수의 동작 : 계산에 오류가 있으면 0 리턴, 아니면 평균 리턴
  - 만일 average()가 리턴 한 값이 0 이라면?
    - 평균이 0인 거야? 아니면 오류가 발생한 거야?

```
int average(int a[], int size) {
  if(size <= 0) return 0;
  int sum = 0;
  for(int i=0; i<size; i++) sum += a[i];
  return sum/size;
}
```

참조 매개변수로 해결

```
int x[ ]={1,2,3,4};
int avg = average(x, 4);  // avg는 2
```

흠, 평균이 2군.
알았어!

```
int x[ ]={1,2,3,4};
int avg = average(x, -1); // avg는 0
```

평균이 0인 거야,
아니면 오류가 난 거야?

참조 매개 변수로 평균을 리턴 하는 예

```
bool average(int a[], int size, int& avg) {  //참조 매개 변수 avg에 평균 값 전달
	if (size <= 0)
		return false;   //함수의 성공 여부를 bool type으로 반환
	int sum = 0;
	for (int i = 0; i < size; i++)
		sum += a[i];
	avg = sum / size;
	return true;	//함수의 성공 여부를 bool type으로 반환
}
void write_avg(const int &avg) {  //const 참조자 : 값 변경 금지
	cout << "평균은 " << avg << endl;
}
void write_error() {
	cout << "매개 변수 오류 " << endl;
}
int main() {
	int x[] = { 0,1,2,3,4,5 };  int avg;
	average(x, 6, avg) ? write_avg(avg) : write_error(); //avg는 평균, average()는 true 리턴, write_avg() 실행
	average(x, -2, avg) ? write_avg(avg) : write_error(); //false 반환, write_error() 실행
}
```

```
평균은 2
매개 변수 오류
```

참조에 의한 호출로 Circle 객체에 참조 전달

```
void increaseCircle(Circle &c) {
  int r = c.getRadius();
  c.setRadius(r+1);
}

int main() {
  Circle waffle(30);

  //참조에 의한 호출
  increaseCircle(waffle);

  cout << waffle.getRadius() << endl;
}
```

waffle 객체 생성

```
생성자 실행 radius = 30
31
소멸자 실행 radius = 31
```

waffle 객체 소멸

```
class Circle {
  private:
    int radius;
  public:
    Circle();
    Circle(int r);
    ~Circle();
    double getArea()  { return 3.14*radius*radius; }
    int getRadius() { return radius; }
    void setRadius(int radius) { this->radius = radius; }
};
Circle::Circle() : Circle(1) { }
Circle::Circle(int radius) {
  this->radius = radius;
  cout << "생성자 실행 radius = "  << radius << endl;
}
Circle::~Circle() {
  cout << "소멸자 실행 radius = " << radius << endl;
}
```

1. 참조 매개변수로 이루어진 연산은 원본 객체에 대한 연산
2. 참조 매개변수는 이름만 생성, 생성자와 소멸자는 실행되지 않음

참조 리턴

- C++의 함수 리턴
  - 함수는 값 외에 참조 리턴도 가능.

  - 참조 리턴
    - **변수의 값을 리턴 하는 것이 아님.**
    - 변수 등과 같이 현존하는 **공간에 대한 참조를 리턴함.**

값을 리턴하는 함수 vs. 참조를 리턴하는 함수

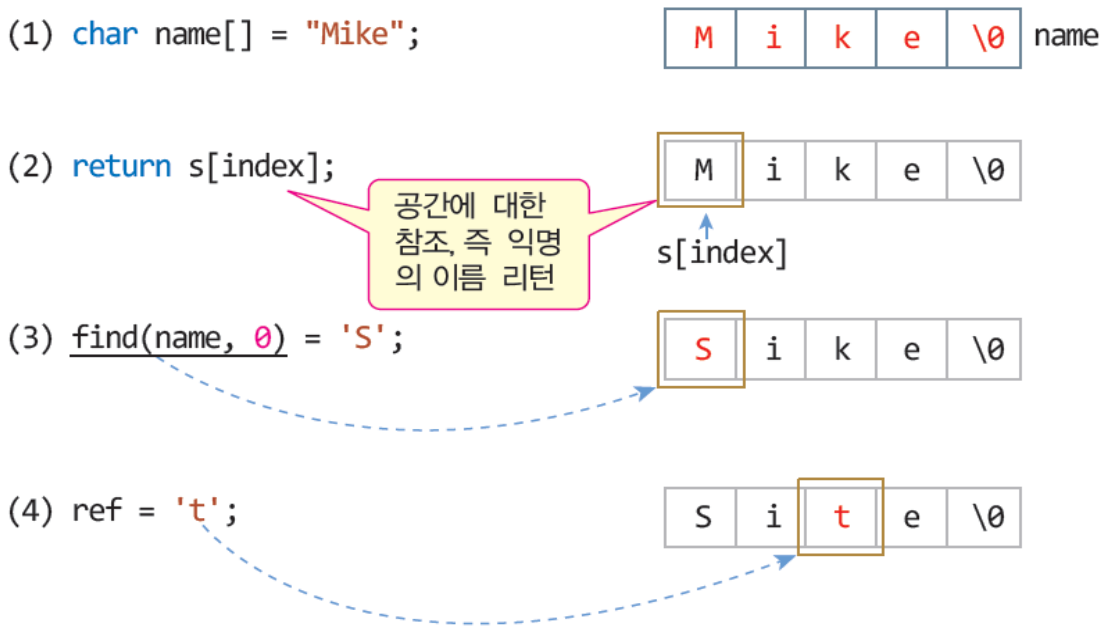
- char 타입의 참조(공간)을 리턴하는 find()

```
char ch1 = 'a';

char& find() {              // char 타입의 참조 리턴
  return ch1;               // 변수 ch1에 대한 참조 리턴, char 타입의 공간에 대한 참조 리턴
}
char ch2 = find();          // ch2='a'가 됨

char &ref = find();         // ref는 ch1에 대한 참조
ref = 'M';                  // ch1='M'

find() = 'b';               // ch1='b'가 됨, find()가 리턴한 공간에 'b' 문자 저장
```

```
char ch1 = 'a';
char get() {                // char 리턴
    return ch1;             // 변수 ch1의 문자('a') 리턴
}
char ch2 = get();           // ch2='a'가 됨
get() = 'b'; // 컴파일 오류
```

참조 리턴 예

```
char& find(char s[], int index) {
  return s[index];  //s[index] 공간의 참조 리턴
}

int main() {
  char name[] = "Mike";
  cout << name << endl;

  //find()가 리턴한 위치에 문자 'S' 저장, 즉 name[0]='S'로 변경
  find(name, 0) = 'S';  cout << name << endl;

  char& ref = find(name, 2); //ref는 name[2] 참조

  ref = 't'; // name = "Site"
  cout << name << endl;
}
```




```
Mike
Sike
Site
```

const 참조자를 이용한 상수 참조

```
//const 참조자를 사용하여 상수를 참조하면 임시 변수를 만들어 참조자가 참조하게 함
int sum(const int& fd1, const int& fd2) {
	return fd1 + fd2;
}
int main() {
	int d1=30, d2=44;

	cout << "sum(d1, d2) = " << sum(d1, d2) << endl;
	cout << "sum(45, 12) = " << sum(45, 12) << endl;
}
```

```
int arr[] = { 1,2,3,4,5,6,7,8,9,10 };

//1. 배열에서 범위 기반 for문 : 참조자를 사용한 값 변경
for (int& elem : arr)
	elem  += 1;

//2. 값 변경 및 복사 방지 배열 요소 사용 : const와 참조 사용
for (const int& elem : arr)
	cout << elem << "  "; //elem  += 1; 오류
```

동적으로 할당된 메모리 공간에 대한 참조자

```
class Point {
public:
        int x, y;
        void write_xy() { cout << "x=" << x << ",  y=" << y << " ]"<< endl; }
};
Point& sum(const Point& p1, const Point& p2) {
        Point *p=new Point;
        p->x = p1.x + p2.x;
        p->y = p1.y + p2.y;
        return *p;
}
int main() {
       Point *p1 = new Point{4, 5};
       Point *p2 = new Point{14, 15};
       Point& p3 = sum(*p1, *p2);

      cout << "p1 = [ "; p1->write_xy();
      cout << "p2 = [ "; p2->write_xy();
      cout << "p3 = [ "; p3.write_xy();
}
```

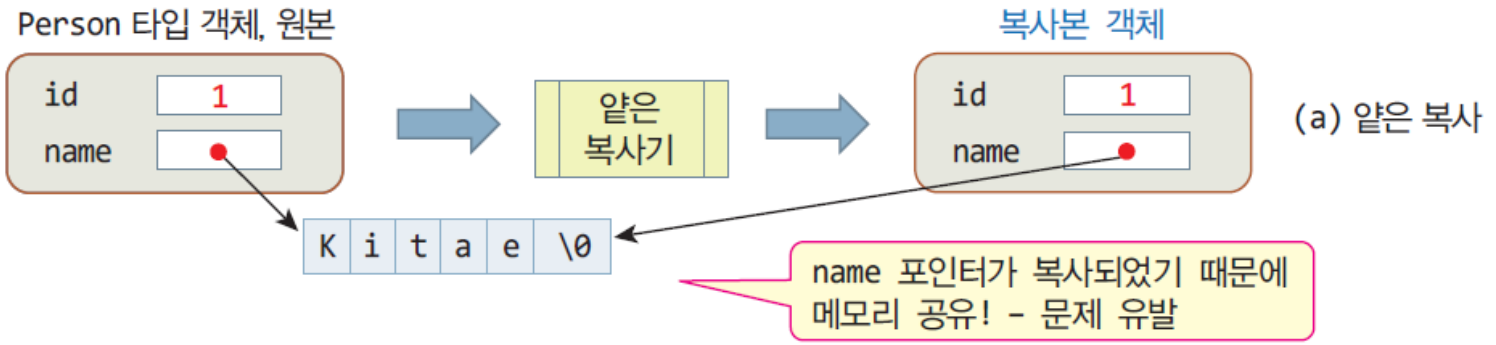
```
p1 = [ x=4,   y=5   ]
p2 = [ x=14,   y=15   ]
p3 = [ x=18,   y=20   ]
```

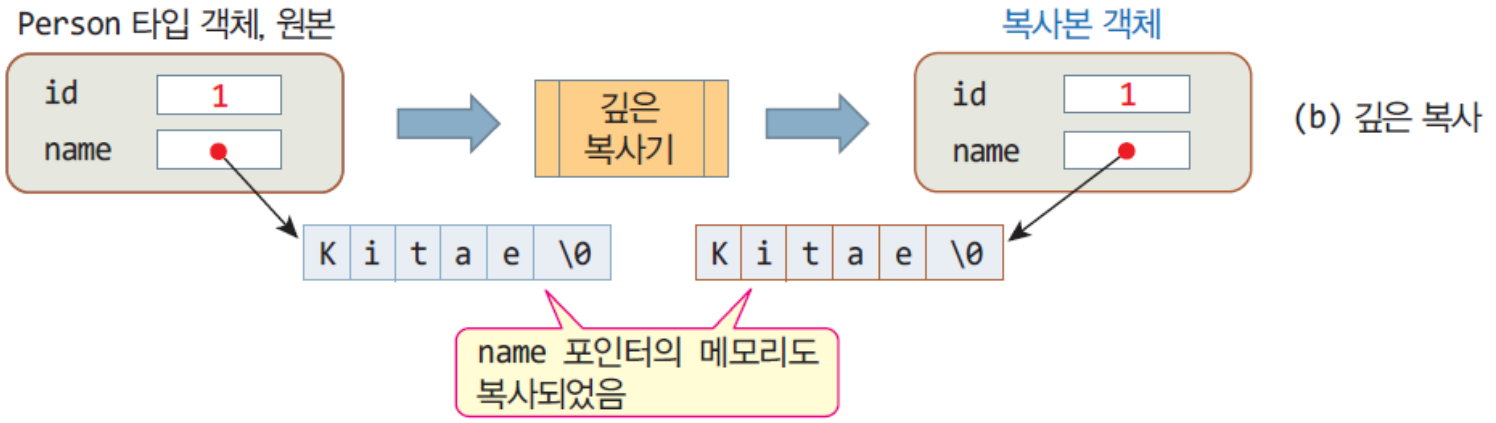
C++에서 얕은 복사와 깊은 복사

- 얕은 복사(shallow copy)
  - 객체 복사 시, 객체의 멤버를 1:1로 복사.
  - 객체의 **멤버 변수에 동적 메모리가 할당**된 경우
    - 사본은 원본 객체가 할당 받은 **메모리를 공유**하는 문제가 발생
- 깊은 복사(deep copy)
  - 객체 복사 시, 객체의 멤버를 1:1대로 복사
  - 객체의 **멤버 변수에 동적 메모리가 할당**된 경우
    - **사본은 원본이 가진 메모리 크기 만큼 별도로 동적 할당**
    - 원본의 동적 메모리에 있는 내용을 사본에 복사
  - 완전한 형태의 복사
    - 사본과 원본이 메모리를 공유하는 문제가 없어짐.

C++에서 객체의 복사

```
class Person {
   int id;
   char *name;
   ..................
};
```





복사 생성자

- 복사 생성자(copy constructor)란?
  - 객체의 복사 생성시 호출되는 특별한 생성자
- 특징
  - 한 클래스에 오직 한 개만 선언 가능
  - 복사 생성자는 보통 생성자와 클래스 내에 중복 선언 가능
  - 모양
    - 클래스에 대한 참조 매개 변수를 가지는 생성자

- 복사 생성자 선언

```
class Circle {
    Circle(Circle& c);    // 복사 생성자 선언
     //Circle& c : 자기 클래스에 대한 참조매개 변수
};
Circle::Circle(Circle& c) {   // 복사 생성자 구현
   ..............
}
```

Circle의 복사 생성자와 객체 복사

```
class Circle {
private:
   int radius;
public:
   Circle() : Circle(1) { }
   Circle(int radius) : radius(radius) {cout << "생성자 실행 radius = " << radius << endl; }
   Circle(Circle& c); // 복사 생성자 선언
   double getArea() { return 3.14*radius*radius; }
};

Circle::Circle(Circle& c) : radius(c.radius) { // 복사 생성자 구현
   cout << "복사 생성자 실행 radius = " << radius << endl;
}

int main() {
   Circle src(30);       // src 객체의  보통 생성자 호출

   Circle dest(src);   // dest 객체의 복사 생성자 호출

   cout << "원본의 면적 = " << src.getArea() << endl;
   cout << "사본의 면적 = " << dest.getArea() << endl;
}
```

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
생성자 실행 radius = 30
복사 생성자 실행 radius = 30
원본의 면적 = 2826
사본의 면적 = 2826
```

디폴트 복사 생성자

- 복사 생성자가 선언되어 있지 않는 클래스
  - 컴파일러는 자동으로 디폴트 복사 생성자 삽입

복사 생성자가 없는 Book 클래스

```
class Book {
    double price;    // 가격
    char *title      // 제목
    char *author;    // 저자이름
public:
    Book(double pr, char* t, char* a;);
    ~Book()
};
```

```
Book(Book& book) {
    //컴파일러가 삽입하는 디폴트 복사 생성자
    this->price = book.price;
    this->title = book.title;
    this->author = book.author;
}
```

```
Book dest(src);    //error ?
```

```
Book(Book& c) = default; //컴파일러가 생성한 복사 생성자를 명시적으로 디폴트로 함
Book(Book& c) = delete;  //복사 생성자 삭제, 객체를 값으로 전달하지 않을 때 설정
```

얕은 복사 생성자 사용 예

```
class Person {
   char *name;
    int id;
public:
   Person(int id, const char* name);
   ~Person();
   void changeName(const char *name);
   void show() { cout << id << ',' << name << endl; }
};
Person::Person(int id, const char* name) {
   this->id = id;
   int len = strlen(name);                  //name의 문자 개수
   this->name = new char [len+1]; //name 문자열 공간 할당
   strcpy(this->name, name);                //name에 문자열 복사
}
Person::~Person() {
   if(name)       //만일 name에 동적 할당된 배열이 있으면
      delete [] name; //동적 할당 메모리 소멸, name 메모리 반환
  cout << "소멸자 호출" << endl;
}
void Person::changeName(const char* name) {   //이름 변경
   if(strlen(name) > strlen(this->name)) return;
   strcpy(this->name, name); //#include <cstring>
}
```

```
Person::Person(Person& p) {
   this->id = p.id;
   this->name = p.name;
}
```

컴파일러에 의해
디폴트 복사 생성자 삽입

```
int main() {
   Person father(1, "Kitae");  //1. father 객체 생성
   Person daughter(father);  //2. daughter 객체 복사 생성
                             //명시적으로 복사 생성자 호출
                             //디폴트 복사 생성자 호출

   cout << "daughter 객체 생성 직후" << endl;
   father.show();            //3. father 객체 출력
   daughter.show();          //3. daughter 객체 출력

   daughter.changeName("Grace"); //4. 이름 변경

   cout << "daughter 이름을 Grace로 변경한 후" << endl;
   father.show();            //5. father 객체 출력
   daughter.show();          //5. daughter 객체 출력

   return 0;                 //6. daughter 객체 소멸
                             //7. father 객체 소멸
}
```

```
daughter 객체 생성 직후
1,Kitae
1,Kitae
daughter 이름을 Grace로 변경한 후
1,Grace
1,Grace
소멸자 호출
소멸자 호출
```

깊은 복사 생성자를 가진 정상적인 예

```
class Person { //Person 클래스 선언
   char *name;
  int id;
public:
   Person(int id, const char* name);  //생성자

   Person(const Person& person); // 복사 생성자

   ~Person(); //소멸자
   void changeName(const char *name);
   void show() { cout << id << ',' << name << endl; }
};
```

```
int main() {
   Person father(1, "Kitae");
   Person daughter(father);

   cout << "daughter 객체 생성 직후" << endl;
   father.show();
   daughter.show();

   daughter.changeName("Grace");

   cout << "daughter 이름을 Grace로 변경한 후" << endl;
   father.show();
   daughter.show();

   return 0;
}
```

```
복사 생성자 실행. 원본 객체의 이름 Kitae
daughter 객체 생성 직후
1,Kitae
1,Kitae
daughter 이름을 Grace로 변경한 후
1,Kitae
1,Grace
소멸자 호출
소멸자 호출
```

```
Person::Person(const Person& person) { //복사 생성자
   this->id = person.id;                //id 값 복사

   int len = strlen(person.name);       //name의 문자 개수
   this->name = new char [len+1];       //name을 위한 공간 할당
   strcpy(this->name, person.name);     //name의 문자열 복사

   cout << "복사 생성자 실행. 원본 객체의 이름 " << this->name << endl;
}
```

```
//생성자 이니셜라이저 사용 (멤버 하나씩 개별적으로)
Person::Person(const Person& p) : id(p.id) ..... { }
```

char *name을 string으로 변경

```
class Person {
    string name;
    int id;
public:
    Person(int id, const string name);
    ~Person();
    Person(const Person &person); // 복사 생성자
    void changeName(const string name);
    void show() { cout << id << ',' << name << endl; }
};
Person::Person(int id, const string name) {
    cout << "생성자 실행" << endl;
    this->id = id;
    this->name = name;
}
Person::Person(const Person& p) : name(p.name), id(p.id) { //생성자 이니셜라이저 사용
    cout << "복사 생성자 실행. 원본 객체의 이름 " << this->name << endl;
}
Person::~Person() {
    cout << "소멸자 호출" << endl;
}
void Person::changeName(string name) {
    this->name = name;
}
```

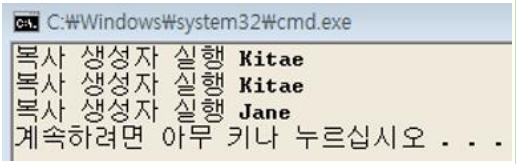
복사 생성자가 자동 호출되는 경우

```
void f(Person person) { //2. '값에 의한 호출'로 객체가 전달될 때, person 객체의 복사 생성자 호출
    person.changeName("dummy");
}
Person g() {
   Person mother(2, "Jane");
   return mother;
}
int main() {
    cout << "--A" << endl;
    Person father(1, "Kitae");

    cout << "--B" << endl;
    Person son = father; //1. 객체로 초기화하여 객체가 생성될 때, son 객체의 복사 생성자 호출

    cout << "--C" << endl;
    f(father);

    cout << "--D" << endl;
    Person rst = g(); //3. 컴파일러마다 다름(복사생성자를 호출하는 오버헤드가 크면 리턴 객체에 바로 값을 대입)
    cout << "g() 함수 리턴 객체 정보 : ";
    rst.show();
    cout << "--F" << endl;
}
```

//Visual Studo Code g++ 컴파일러 실행 시

```
PS C:\yanges\lecture\lecture_src\cpp> g++ cpptest.cpp
PS C:\yanges\lecture\lecture_src\cpp> ./a
--A
생성자 실행
--B
복사 생성자 실행. 원본 객체의 이름 Kitae
--C
복사 생성자 실행. 원본 객체의 이름 Kitae
소멸자 호출
--D
생성자 실행
g() 함수 리턴 객체 정보 : 2,Jane
--F
소멸자 호출
소멸자 호출
소멸자 호출
```

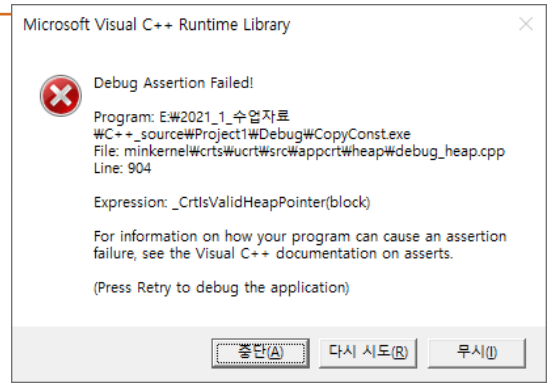
//Visual Studo 2017 실행 시



객체 대입 연산 실행

```
int main() {
    cout << "--A" << endl;
    Person father(1, "Kitae"); //father 객체 생성

    cout << "--B" << endl;
    Person sister(father);      //sister 객체 복사 생성, 복사 생성자 호출

    cout << "--C" << endl;
    sister.changeName("Grace"); //sister의 이름을 "Grace"로 변경
    father = sister;            //father객체에 sister 대입 – Visual Studo 2017 실행 시 비정상적 종료
                                //원인 : 디폴트 대입 연산 실행(얕은 복사)
                                //해결 방법 : 복사 대입 연산자 함수 정의(연산자함수에서 설명)
    father.show();

    return 0;
}
```

//Visual Studo 2017 실행 시



//Visual Studo Code g++ 컴파일러 실행 시

```
--A
생성자 실행
--B
복사 생성자 실행. 원본 객체의 이름 Kitae
--C
1,Grace
소멸자 호출
```

//복사대입연산자 적용 시

```
--A
생성자 실행
--B
복사 생성자 실행. 원본 객체의 이름 Kitae
--C
복사 대입 연산자 실행. 원본 객체의 이름 Grace
1,Grace
소멸자 호출
소멸자 호출
```

Rvalue Reference & Lvalue Reference

- Lvalue & Rvalue
  - Lvalue : 변수처럼 이름과 주소를 가진 대상(지속되는 객체)
  - Rvalue : 리터럴, 임시객체처럼 더 이상 존재하지 않는 임시적인 객체
    - **Rvalue 예**

```
int val = 3;
int x = val + 4;
int temp = x;
int* ptr = &x;
cout << "Hallym"
<< endl;
```

- Lvalue reference : lvalue만 참조(&)
- Rvalue reference : rvalue만 참조(&&)
  - 불필요한 rvalue복사와 이로 인한 오버헤드 방지
  - 메모리 소유권 전환 - 메모리 누수와 Dangling Pointer 방지

- 이동 생성자와 이동 대입 연산자 구현에 Rvalue reference 사용 -> 원본 객체 삭제 시에 만 유용

Move Semantics

- 임시 객체(Rvalue)는 복사가 아닌 이동 되는 것이 상식적 임
- C++11에 객체의 이동이라는 개념이 도입됨 → Move Semantics

- Move Semantics 구현
  - 이동 생성자와 이동 대입 연산자 구현
  - Rvalue Reference를 파라미터로 받는 함수 작성

- 컨테이너에 객체를 삽입할 때 더 이상 포인터를 넣지 않아도 됨
- vector 컨테이너와 같은 대규모 리소스를 반환하는 함수 작성 가능

Move Semantics 구현 예

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

void helper(string&& para) {
    cout << "helper : " << para << endl;
}
void message(string& msg) {
    cout << "lvalue reference : " << msg << endl;
}
void message(string&& msg) { //Rvalue Reference를 파라미터로 받는 함수
    cout << "rvalue reference : " << msg << endl;
    //helper(msg); //error, 매개변수 msg는 lvalue
    helper(move(msg)); //move() : lvalue->rvalue
}
int main() {
    string str = "apple";
    string temp = "banana";
    message(str); //message(move(str))로 호출하면?
    message(str + temp);
    message("orange");
    return 0;
}
```

```
lvalue reference : apple
rvalue reference : applebanana
helper : applebanana
rvalue reference : orange
helper : orange
```

Move Semantics 구현 (move constructor)

```
#include <iostream>
#include <vector>
using namespace std;
class Person {
    char* name;
    int id;
public:
    Person(int id, const char* name);  //구현 생략
    Person(const Person& p);  //구현 생략
    Person(Person&& p) noexcept; //이동 생성자
    ~Person(); //구현 생략
    void show() { cout << id << ',' << name << endl; }
};
Person::Person(Person&& p) noexcept {
    cout << "이동 생성자 실행" << endl;
    id = p.id;
    name = p.name;
    p.name = nullptr;
    p.id = 0;
}
void disPlay(Person&& per) {
    per.show();
}
```

```
int main() {
    vector<Person> vp; //

    cout << "=1==========" << endl;
    vp.push_back(Person(1, "python")); //이동

    cout << "=2==========" << endl;
    vp.push_back(Person(2, "java")); //이동

    cout << "=3==========" << endl;
    vp.emplace_back(3, "c");

    cout << "=4==========" << endl;
    vp.emplace_back(4, "c++");

    //vector<Person*> vp; //이동 생성자 없으면 포인터 사용
    //vp.push_back(new Person(3, "c"))

    for (auto &tmp : vp)
        tmp.show();

    cout << "=5==========" << endl;
    //rvalue참조를 매개변수로 갖는 함수 호출
    disPlay(Person(5, "web"));
    return 0;
}
```

```
=1==========
생성자 실행
이동 생성자 실행
소멸자 실행
=2==========
생성자 실행
이동 생성자 실행
이동 생성자 실행
소멸자 실행
소멸자 실행
=3==========
생성자 실행
이동 생성자 실행
이동 생성자 실행
소멸자 실행
소멸자 실행
=4==========
생성자 실행
1,python
2,java
3,c
4,c++
=5==========
생성자 실행
5,web
소멸자 실행
소멸자 실행
소멸자 실행
소멸자 실행
소멸자 실행
```

학습 정리 (1)

- 함수의 인자 전달 방식
  - 값, 주소, 참조
  - 함수에 객체 전달
  - 객체 치환
  - 객체 리턴

- 참조에 의한 함수 호출/리턴

- 얕은 복사와 깊은 복사

- 복사 생성자(copy constructor) // Person(const Person& p);

- 복사 생성자가 자동 호출되는 경우
  1. 객체로 초기화하여 객체가 생성될 때 //Person son = father
  2. '값에 의한 호출'로 객체가 전달될 때 //f(Person person)
  3. 객체를 리턴 받을 때는 컴파일러마다 다름 //Person rst = g();

학습 정리 (2)

- 객체 대입 연산 (복사 대입 연산자 함수 정의 필요)
  - father = sister;

- Rvalue & Lvalue
  - void message(string&& msg)
  - message("orange");

- Move Semantics
  - 이동 생성자(move constructor)

```
Person::Person(Person&& p) noexcept {
    cout << "이동 생성자 실행" << endl;
    id = p.id;
    name = p.name;
    p.name = nullptr;
    p.id = 0;
}
```

Q & A

- "C++ 참조와 복사생성자"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_06_참조와복사생성자_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.^^