



함수와 참조 & 복사 생성자

함수와 참조 & 복사생성자

- 한 주간 잘 지내셨나요?
- 오늘은
 - 함수 인자 전달 방식
 - 참조 – rvalue & lvalue
 - 복사 생성자
 - Move Semantics에 대하여 알아보려고 합니다.
- 어렵기는 하지만 C++다운 프로그램을 위해 꼭 필요한 개념입니다.
- 준비 되셨나요?

학습 목차

- 함수 인자 전달 방식
 - 값에 의한 호출
 - 주소에 의한 호출
- 참조란?
- 참조에 의한 호출
- 얇은 복사 & 깊은 복사
- 복사 생성자
- Rvalue Reference & Lvalue Reference
- Move Semantics

학습 목표

- 함수 인자 전달 방식을 이해하고 활용할 수 있다
- Reference 개념을 이해하고 활용할 수 있다
- Rvalue참조와 Lvalue 참조에 대하여 이해하고 구분 할 수 있다
- 복사 생성자의 필요성을 이해하고 활용할 수 있다
- 이동 생성자의 필요성을 이해하고 활용할 수 있다.
- Move Semantics을 이해하고 활용할 수 있다

함수의 인자 전달 방식 리뷰

- 인자 전달 방식
 - 값에 의한 호출, call by value
 - 함수가 호출되면 매개 변수가 스택에 생성됨
 - 호출하는 코드에서 값을 넘겨줌
 - 호출하는 코드에서 넘어온 값이 매개 변수에 복사됨
 - 주소에 의한 호출, call by address
 - 함수의 매개 변수는 포인터 타입
 - 함수가 호출되면 포인터 타입의 매개 변수가 스택에 생성됨
 - 호출하는 코드에서는 명시적으로 주소를 넘겨줌
 - 기본 타입 변수나 객체의 경우, 주소 전달
 - 배열의 경우, 배열의 이름
 - 호출하는 코드에서 넘어온 주소 값이 매개 변수에 저장됨

함수의 인자 전달 방식 리뷰

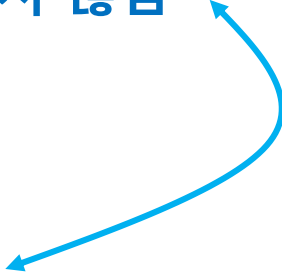
값에 의한 호출

```
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**
- 
- 매개 변수 객체의 생성자
소멸자의 비대칭 실행 구조
- *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 생성자 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() {
    radius = 1;
    cout << "생성자 실행 radius = " << radius << endl;
}

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;
}
```

waffle의 내용이
그대로 c에 복사

waffle 생성

c의 생성자
실행되지 않았음

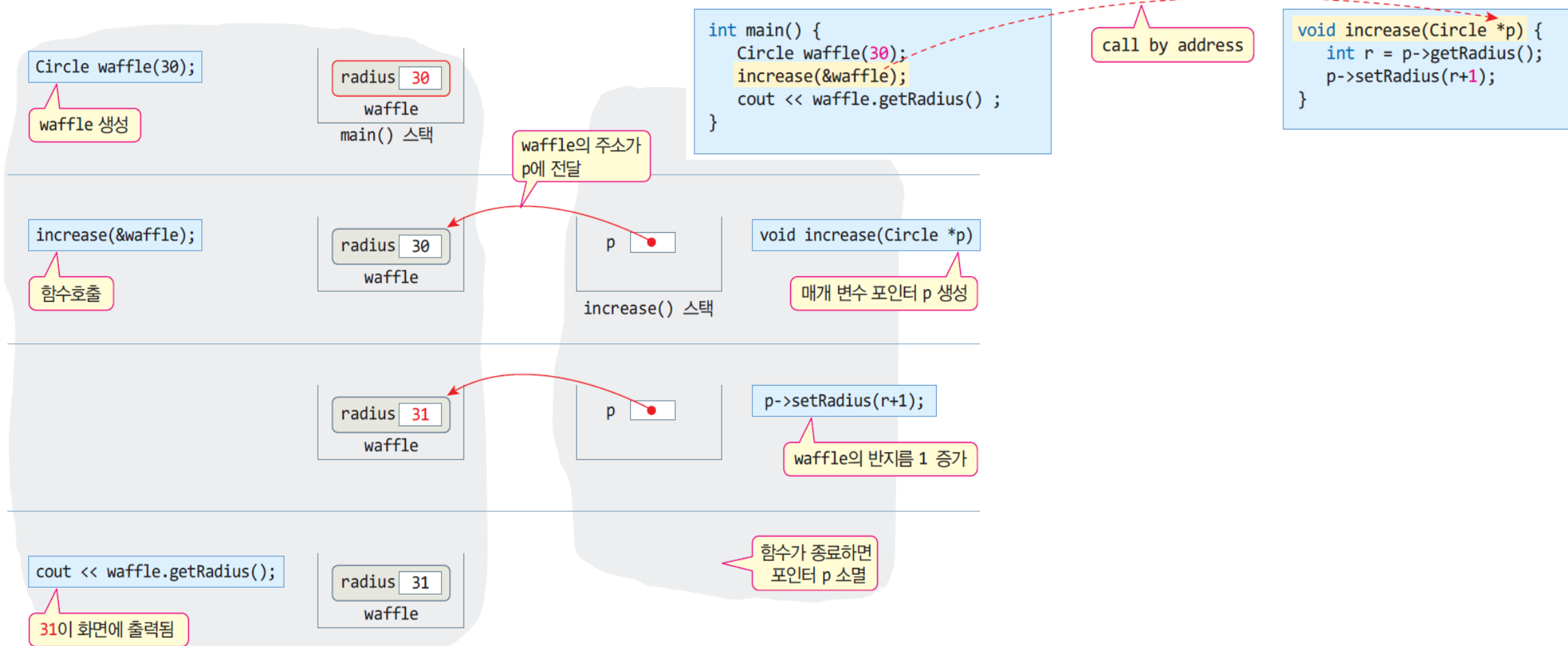
생성자 실행 radius = 30
소멸자 실행 radius = 31
30
소멸자 실행 radius = 30

c 소멸

waffle 소멸

함수에 객체 전달 - '주소에 의한 호출'

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



객체 치환 및 객체 리턴

- 객체 치환

- 동일한 클래스 타입의 객체끼리 치환 가능
- 객체의 모든 데이터가 비트 단위로 복사

```
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  
}
```

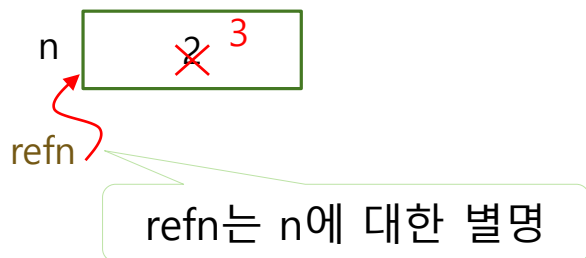
```
Circle c;    // c의 반지름 1  
c = getCircle();    // tmp 객체의 복사본이 c에 치환. c의 반지름은 30이 됨
```

참조란?

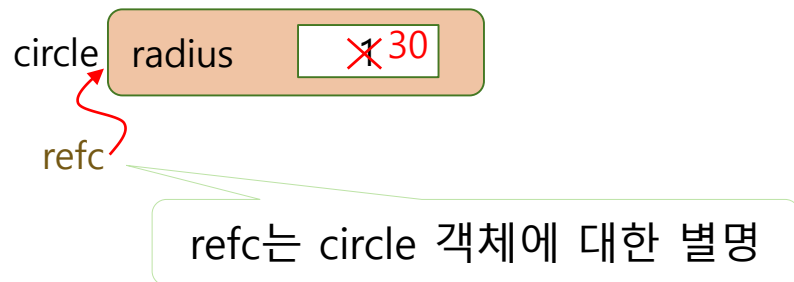
- 참조(reference)란 가리킨다는 뜻으로 이미 존재하는 객체나 변수에 대한 별명
- 참조 활용
 - 참조 변수
 - 참조에 의한 호출
 - 참조 리턴
- 참조 변수 선언
 - 참조자 &의 도입
 - 이미 존재하는 변수에 대한 다른 이름(별명)을 선언
 - 참조 변수는 이름만 생기며 **참조 변수에 새로운 공간을 할당하지 않음** - 초기화로 지정된 **원본 변수의 공간 공유**
 - 선언 시 반드시 원본 변수로 초기화** - **초기화 없으면 오류 발생** `int &refn; //오류`
 - 참조 변수의 배열을 만들 수 없다
`int &refn[10]; //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); //오류 : refc->setRadius(30);
```



기본 타입 변수에 대한 참조

```
int main() {  
    cout << "i" << 'Wt' << "n" << 'Wt' << "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 << 'Wt' << n << 'Wt' << refn << endl;  
  
    refn = i;         // refn=1, n=1  
    refn++;           // refn=2, n=2  
    cout << i << 'Wt' << n << 'Wt' << refn << endl;  
  
    int *p = &refn;    // p는 n의 주소를 가짐, 참조에 대한 포인터 변수 선언  
    *p = 20;          // refn=20, n=20  
    cout << i << 'Wt' << n << 'Wt' << refn << endl;  
}
```

객체에 대한 참조

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

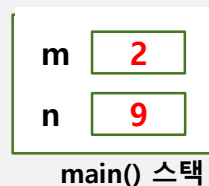
참조에 의한 호출

- 참조를 가장 많이 활용하는 사례
- call by reference라고 부름
- 함수 형식
 - 함수의 매개 변수를 참조 타입으로 선언
 - 참조 매개 변수(reference parameter)라고 부름
 - 참조 매개 변수는 실 인자 변수를 참조함
 - 참조매개 변수의 이름만 생기고 공간이 생기지 않음
 - 참조 매개 변수는 실 인자 변수 공간 공유
 - 참조 매개 변수에 대한 조작은 실 인자 변수 조작 효과

참조에 의한 호출 사례

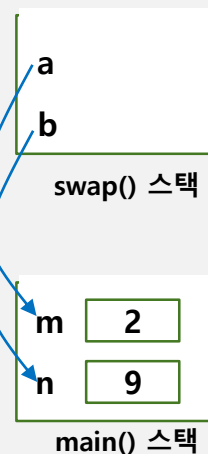
```
void swap( int &a, int &b ) {
    int tmp;
    tmp = a; a = b; b = tmp; //참조 매개 변수를 보통 변수처럼 사용
}
int main() {
    int m=2, n=9;
    swap(m, n); //함수가 호출되면 m,n에 대한 참조 변수 a,b가 생김
    cout << m << ' ' << n;
}
```

a, b는 m, n의 별명.
a, b 이름만 생성.
변수 공간 생기지 않음

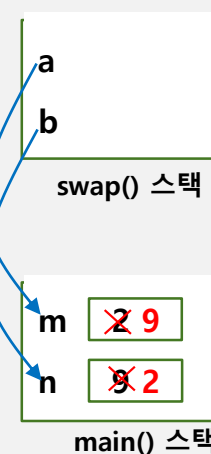


9 2

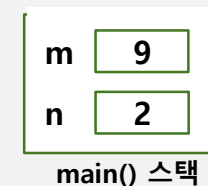
(1) swap() 호출 전



(2) swap() 호출 직후



(3) swap() 실행



(4) swap() 리턴 후

참조 매개변수가 필요한 사례

- 다음 코드에 어떤 문제가 있을까?
 - average() 함수의 작동
 - 계산에 오류가 있으면 0 리턴, 아니면 평균 리턴
 - 만일 average()가 리턴 한 값이 0이라면?
 - 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 평균 리턴하기

```

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 : 참조만 하고 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(); //avg에 평균이 넘어오고, average()는 true 리턴
    average(x, -2, avg) ? write_avg(avg) : write_error(); //false 반환
}

```

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 언어의 함수 리턴
 - 함수는 반드시 값만 리턴
 - 기본 타입 값 : int, char, double 등
 - 포인터 값
- C++의 함수 리턴
 - 함수는 값 외에 참조 리턴 가능
 - 참조 리턴
 - 변수 등과 같이 현존하는 공간에 대한 참조 리턴
 - 변수의 값을 리턴 하는 것이 아님

값을 리턴하는 함수 vs. 참조를 리턴하는 함수

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

```
char c = 'a';  
char& find() { // char 타입의 참조 리턴  
    return c; // 변수 c에 대한 참조 리턴, char 타입의 공간에 대한 참조 리턴  
}  
  
char a = find(); // a = 'a'가 됨  
  
char &ref = find(); // ref는 c에 대한 참조  
ref = 'M'; // c = 'M'  
  
find() = 'b'; // c = 'b'가 됨, find()가 리턴한 공간에 'b' 문자 저장
```

간단한 참조 리턴 사례

```
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;
}
```

(1) `char name[] = "Mike";`

M	i	k	e	\0
---	---	---	---	----

name

(2) `return s[index];`

공간에 대한
참조, 즉 익명
의 이름 리턴

M	i	k	e	\0
---	---	---	---	----

s[index]

(3) `find(name, 0) = 'S';`

S	i	k	e	\0
---	---	---	---	----

(4) `ref = 't';`

S	i	t	e	\0
---	---	---	---	----

const 참조자를 이용한 상수 참조

//const 참조자를 사용하여 상수를 참조하면 임시 변수를 만들어 참조자가 참조하게 함

```
int sum(const int& d1, const int& d2) {
    return d1 + d2;
}
```

```
int main() {
    int d1 = 30, d2 = 44;
    cout << "sum(d1, d2) = " << sum(d1, d2) << endl;
    //const 참조 매개변수를 사용하지 않으면 다음 문장은 오류
    cout << "sum(45, 12) = " << sum(45, 12) << endl;
}
```

//1. 배열에서 범위 기반 for문을 사용하여 값 변경하거나 복사를 피하고 싶으면 - 참조자 사용

```
int arr[] = { 1,2,3,4,5,6,7,8,9,10 };
for (int& elem : arr) //참조자를 사용 값 변경
    elem += 1;
```

//2. 값 변경, 복사 방지 배열 요소 사용 : const 와 참조 사용

```
for (const int& elem : arr)
    cout << elem << " ";
```

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 얇은 복사와 깊은 복사

- 얇은 복사(shallow copy)
 - 객체 복사 시, 객체의 멤버를 1:1로 복사
 - 객체의 멤버 변수에 동적 메모리가 할당된 경우
 - 사본은 원본 객체가 할당 받은 메모리를 공유하는 문제 발생
- 깊은 복사(deep copy)
 - 객체 복사 시, 객체의 멤버를 1:1대로 복사
 - 객체의 멤버 변수에 동적 메모리가 할당된 경우
 - 사본은 원본이 가진 메모리 크기 만큼 별도로 동적 할당
 - 원본의 동적 메모리에 있는 내용을 사본에 복사
 - 완전한 형태의 복사
 - 사본과 원본은 메모리를 공유하는 문제 없음

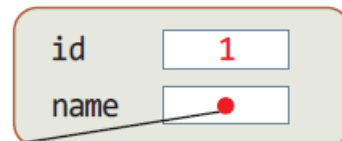
C++에서 객체의 복사

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

Person 타입 객체, 원본



복사본 객체



(a) 얕은 복사

name 포인터가 복사되었기 때문에
메모리 공유! - 문제 유발

Person 타입 객체, 원본



복사본 객체



(b) 깊은 복사

name 포인터의 메모리도
복사되었음

복사 생성자

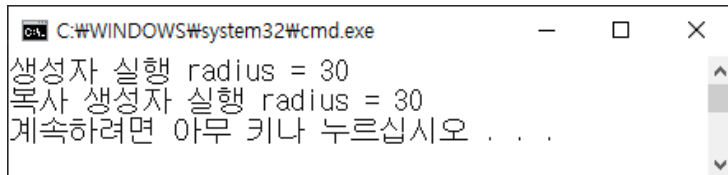
- 복사 생성자(copy constructor)란?
 - 객체의 복사 생성시 호출되는 특별한 생성자
- 특징
 - 한 클래스에 오직 한 개만 선언 가능
 - 복사 생성자는 보통 생성자와 클래스 내에 중복 선언 가능
 - 모양
 - 클래스에 대한 참조 매개 변수를 가지는 독특한 생성자
- 복사 생성자 선언

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

Circle의 복사 생성자와 객체 복사

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

```
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
}
```

디폴트 복사 생성자

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

복사 생성자가 없는 Book 클래스

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

컴파일러에 의해
디폴트 복사 생성자 삽입

Person hallym(software); //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 생성자를 사용하여 프로그램이 비정상 종료되는 경우(1/2)

```

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
}

void Person::changeName(const char* name) { // 이름 변경
    if(strlen(name) > strlen(this->name))
        return;
    strcpy(this->name, name);
}

```

얕은 복사 생성자를 사용하여 프로그램이 비정상 종료되는 경우(2/2)

```
int main() {  
    Person father(1, "Kitae");  
    Person daughter(father);    // 컴파일러가 삽입한 디폴트 복사 생성자 호출  
  
    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;    //daughter, father 순으로 소멸, father가 소멸할 때, 프로그램 비정상 종료됨  
}
```

깊은 복사 생성자를 가진 정상적인 Person 클래스(1/2)

```
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; }
};
```

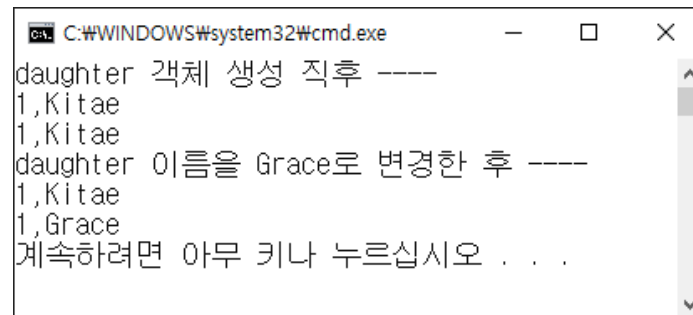
```
Person::Person(const Person& person) { //복사 생성자 구현
    this->id = person.id; // id 값 복사
    int len = strlen(person.name); // name의 문자 개수
    this->name = new char [len+1]; // name을 위한 공간 할당
    strcpy(this->name, person.name); // name의 문자열 복사
}
```

//생성자 이니셜라이저 사용

Person::Person(const Person& p) : Person(p.id, p.name) { }

깊은 복사 생성자를 가진 정상적인 Person 클래스(2/2)

```
int main() {  
    Person father(1, "Kitae");  
    Person daughter(father);    // 복사 생성자 호출  
  
    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;    //daughter, father 객체 소멸, 정상적 종료  
}
```



```
C:\WINDOWS\system32\cmd.exe  
daughter 객체 생성 직후 ----  
1,Kitae  
1,Kitae  
daughter 이름을 Grace로 변경한 후 ----  
1,Kitae  
1,Grace  
계속하려면 아무 키나 누르십시오 . . .
```

복사 생성자가 자동 호출되는 경우

//2. '값에 의한 호출'로 객체가 전달될 때, person 객체의 복사 생성자 호출

```
void f(Person person) {
    person.changeName("dummy");
}
```

```
Person g() {
    Person mother(2, "Jane");
    return mother;
}
```

※ 리턴값이 객체인 경우 - 컴파일러마다 다르게 처리

- 복사 생성자 호출
- 리턴값 최적화(RVO) 또는 복사 생략 적용

객체 반환 시 임시 객체를 없애고, 계산 결과값을 반환 값을 받는 객체에 대해 할당된 메모리에 직접 넣어 초기화 - 생성자 호출

```
Int main() {
    Person father(1, "Kitae");
    //1. 객체로 초기화하여 객체가 생성될 때, son 객체의 복사 생성자 호출
    Person son(father); //다른 객체를 동일하게 복사, Person son = father; 문장도 동일하게 실행
    f(father);
    Person mother = g();
}
```

```
C:\Windows\system32\cmd.exe
복사 생성자 실행 Kitae
복사 생성자 실행 Kitae
복사 생성자 실행 Jane
계속하려면 아무 키나 누르십시오 . . .
```

복사 생성자 실행 : Kitae
복사 생성자 실행 : Kitae

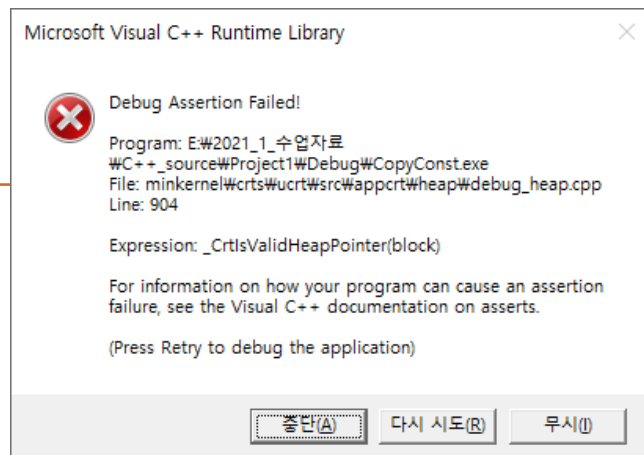
객체 대입 연산 실행

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

    daughter.changeName("Grace"); // (3) daughter의 이름을 "Grace"로 변경
    father = daughter;            // (4) father 객체에 daughter 대입 - 비정상적 종료
                                // 원인 : 디폴트 대입 연산 실행(얕은 복사)
                                // 해결 방법 : 대입 연산자 함수 정의(프렌드와 연산자함수에서 설명)

    father.show();

    return 0;
}
```



Rvalue Reference & Lvalue Reference

- Lvalue & Rvalue
 - Lvalue : 변수처럼 이름과 주소를 가진 대상(지속되는 객체)
 - Rvalue : 리터럴, 임시객체처럼 더 이상 존재하지 않는 일시적인 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 댕글링 포인터 방지
- 이동 생성자와 이동 대입 연산자 구현에 Rvalue reference 사용 -> 원본 객체 삭제 시에 만 유용

Move Semantics

- 임시 객체(Rvalue)의 복사가 아닌 이동이 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 1

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

```
void helper(string&& para) {
    cout << para << endl;
}
```

```
void message(string& message) {
    cout << "lvalue reference : " << message << endl;
}
```

```
void message(string&& message) { //Rvalue Reference를 파라미터로 받는 함수
    cout << "rvalue reference : " << message << endl;
    //helper(message); //error, 매개변수 message는 lvalue
    helper(move(message)); //move() : lvalue->rvalue
}
```

```
int main() {
    string str = "message";
    string temp = "move";
    message(str); //message(move(str))로 호출하면?
    message(str + temp);
    message("reference");

    return 0;
}
```

```
lvalue reference : message
rvalue reference : messagemove
messagemove
rvalue reference : reference
reference
```

Move Semantics 구현 2

```
#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 << " == move constructor == " << endl;
    id = p.id;
    name = p.name;
    p.name = nullptr;
    p.id = 0;
}

void display(Person&& per) {
    per.show();
}
```

```
int main() {
    vector<Person> vp;
    vp.push_back(Person(3, "hallym")); //이동 생성자 호출
    vp.push_back(Person(5, "software"));

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

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

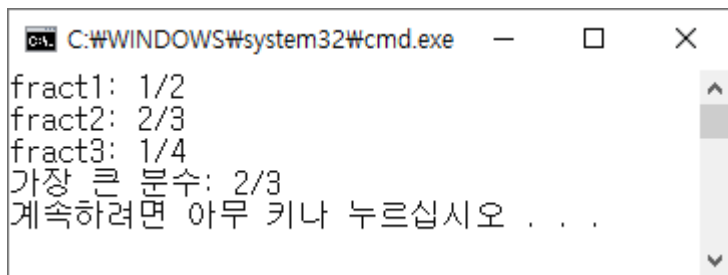
    //rvalue참조를 매개변수로 갖는 함수 호출
    display(Person(10, "bigdata"));
    return 0;
}
```

확인 1. 참조 활용

- Fraction 클래스에 참조로 전달과 참조로 리턴을 활용해서 3개의 분수 중에서 가장 큰 분수를 찾는 프로그램

```
Fraction& findLarger(Fraction& fr1, Fraction& fr2);
```

```
Fraction& findLargest(Fraction& fr1, Fraction& fr2, Fraction& fr3);
```



```
C:\WINDOWS\system32\cmd.exe
fract1: 1/2
fract2: 2/3
fract3: 1/4
가장 큰 분수: 2/3
계속하려면 아무 키나 누르십시오 . . .
```


확인 2. 복사 생성자

- 제시된 결과처럼 실행될 수 있도록 MyClass 클래스를 완성하시오

```
class MyClass{
    int size;
    int *element;
public:
    MyClass(int size);
    ~MyClass();
};
```

```
==== original(my) ====
0  0  0  0  0
==== change(my) ====
0  0  30  0  15
==== copy after ====
0  0  30  0  15
==== original(my) ====
0  0  30  0  15
==== change(copy) ====
0  23  30  0  61
```

```
MyClass::MyClass(int size) : size(size), element(new int[size]){
    for(auto i=0; i<size; i++){
        element[i] = 0;
    }
}
```

```
MyClass::~MyClass(){
    if(!element)
        delete[] element;
}
```

```
int main(){
    MyClass my{5};
    my.write("original(my)");
    my.change(2, 30);
    my.change(4, 15);
    my.write("change(my)");
    MyClass copy(my);
    copy.write("copy after");
    my.write("original(my)");
    copy.change(1, 23);
    copy.change(4, 61);
    copy.write("change(copy)");
}
```

Q & A

- “문자열 함수와 참조”에 대한 학습이 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
- 7주 과제와 퀴즈(제한 기간내 2회 응시 가능)가 있습니다.
- 수고하셨습니다.^^