윈도우 프로그래밍

8. 객체 활용(2)

2018. 5. 4.

심미나 교수

목 차

I. 객체 배열과 this포인터

객체 배열과 this포인터

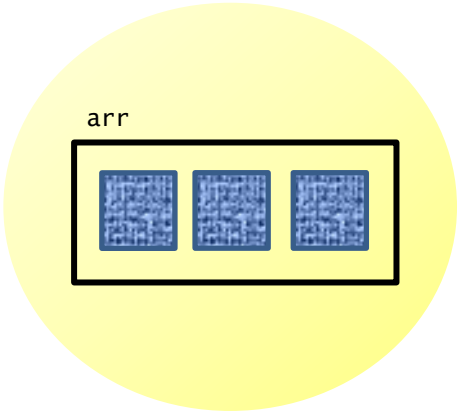
객체 배열과 객체 포인터 배열

- **객체 배열**
  - 객체로 이루어진 배열
  - 배열 생성시, 객체가 함께 생성
    - 호출되는 생성자는 void 생성자

```
Person arr[3];
Person *parr = new Person[3];
```

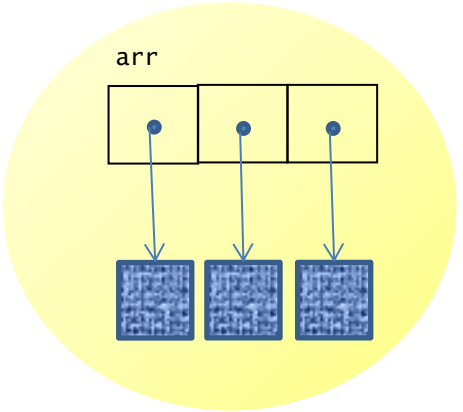
ⓘ Person객체 3개 묶인 배열

ⓘ Delete []parr; 로 삭제
ⓘ Delete parr[3]; (x)



- **객체 포인터 배열**
  - 객체를 저장할 수 있는 포인터 변수로 이루어진 배열
  - 별도의 객체 생성 과정 필요

```
Person *arr[3];
arr[0]=new Person(name, age);
arr[1]=new Person(name, age);
arr[2]=new Person(name, age);
```



※객체배열을 선언할지, 객체포인터배열을 선언할지 먼저 결정해야 함!

객체 배열과 this포인터

this 포인터의 이해

- this 포인터 - 사용된 객체 자신의 주소값을 갖는 포인터

```
int main(void)
{
    SoSimple sim1(100);
    SoSimple * ptr1=sim1.GetThisPointer();
    cout<<ptr1<<", ";
    ptr1->ShowSimpleData();
    SoSimple sim2(200);
    SoSimple * ptr2=sim2.GetThisPointer();
    cout<<ptr2<<", ";
    ptr2->ShowSimpleData();
    return 0;
}
```

1) 객체 생성

2) 객체 주소값 반환

3) 반환된 주소값 출력

4) 객체(주소)의 데이터 출력

실행 결과

```
Num=100, address=0012FF60
0012FF60, 100
Num=200, address=0012FF48
0012FF48, 100
```

```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    {
        cout<<"num="<<num<<", ";
        cout<<"address="<<this<<endl;
    }
    void ShowSimpleData()
    {
        cout<<num<<endl;
    }
    SoSimple * GetThisPointer()
    {
        return this;
    }
};
```

① 반환형(포인터)

① sim 자기자신

객체 배열과 this포인터

this 포인터의 활용

- 객체의 주소 값(this)으로 멤버변수에 접근 가능

```
class TwoNumber
{
private:
    int num1;
    int num2;
public:
    TwoNumber(int num1, int num2)
    {
        this->num1=num1;
        this->num2=num2;
    }
```

ⓘ 멤버변수

ⓘ 매개변수

ⓘ 멤버변수 접근 시 this 사용
this->num1 : 멤버변수 num1
즉, 객체주소(this)는 매개변수 접근에 사용 못하고, 멤버변수 접근 시에만 사용

```
TwoNumber(int num1, int num2)
    : num1(num1), num2(num2)
{
    // empty
}
```

ⓘ 멤버변수

ⓘ 이니셜라이저는 this-> 표현 사용 불가

객체 배열과 this포인터

Self-reference의 반환

- Self-reference - **객체 자신을 참조하는 참조명**

```
int main(void)
{
    SelfRef obj(3);
    SelfRef &ref=obj.Adder(2);
    obj.ShowTwoNumber();
    ref.ShowTwoNumber();
    ref.Adder(1).ShowTwoNumber().Adder(2).ShowTwoNumber();
    return 0;
}
```

ⓘ 객체자신을 반환

```
class SelfRef
{
private:
    int num;
public:
    SelfRef(int n) : num(n)
    {
        cout<<"객체생성"<<endl;
    }
    SelfRef& Adder(int n)
    {
        num+=n;
        return *this;
    }
    SelfRef& ShowTwoNumber()
    {
        cout<<num<<endl;
        return *this;
    }
};
```

ⓘ 단, 참조형으로 반환

ⓘ 객체자신을 반환

실행 결과

```
객체생성
5
5
6
8
```

객체 배열과 this포인터

Self-reference의 반환

- Self-reference - 객체 자신을 참조하는 참조명

```
int main(void)
{
    SelfRef obj(3);
    SelfRef &ref=obj.Adder(2);
    obj.ShowTwoNumber();
    ref.ShowTwoNumber();
    ref.Adder(1).ShowTwoNumber().Adder(2).ShowTwoNumber();
    return 0;
}
```

① 객체자신을 반환

```
class SelfRef
{
private:
    int num;
public:
    SelfRef(int n) : num(n)
    {
        cout<<"객체생성"<<endl;
    }
    SelfRef& Adder(int n)
    {
        num+=n;
        return *this;
    }
    SelfRef& ShowTwoNumber()
    {
        cout<<num<<endl;
        return *this;
    }
};
```

① 단, 반환형(참조자)

```
(ref.Adder(1)).ShowTwoNumber().Adder(2).ShowTwoNumber();
```

(1) ref참조자 반환

(2) (ref참조자).ShowTwoNumber()

(3) (ref참조자).Adder(2)

(4) (ref참조자).ShowTwoNumber()

II. 복사생성자

복사생성자

복사생성자의 이해 - C++의 대입연산

- **복사생성자의 필요성**
  - 일반 자료형과 객체의 대입연산 비교

```
int num1 = 10;
int num2 = num1;
```

```
int num2;
num2 = num1;
```

VS.

```
A   =   B;   //객체 대입
```

ⓘ 공간을 동절할당하고 참조하는 형태로 객체가 존재
ⓘ 자료형처럼 객체간 대입이 가능한지 판단하기 어려움
ⓘ 따라서 객체간 대입연산 문제는 프로그래머가 직접 정의

- **객체의 대입연산**

```
(1) ABC obj1 = obj2;
```

ⓘ 새로운 객체를 생성할 때, 기존 객체로 초기화할 때 일어나는 대입연산의 결과를 정의하는 것. 즉, "복사생성자" 정의!

```
(2) ABC obj1
    Obj1 = obj2;
```

ⓘ 이미 존재하는 객체간 대입은 "연산자 오버로딩" 개념
ⓘ 기존 객체를 기존에 생성된 객체로 초기화할 때, 연산의 결과를 정의하는 것

복사생성자

복사생성자의 이해 - C++의 초기화

- C와 C++의 초기화

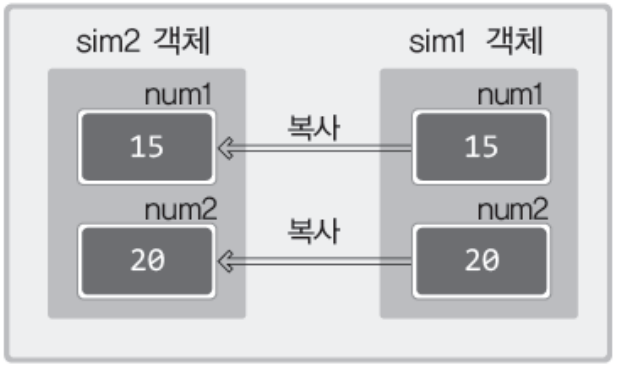
ⓘ C

```
int num = 20;
int &ref = num;
```

→

ⓘ C++

```
int num(20);
int &ref(num);
```



  - 따라서, Sosimple sim2=sim1; 과
    SoSimple sim2(sim1); 은 동일하게 해석!

- SoSimple sim2(sim1); 의미
  - SoSimple 클래스의 객체 sim2를 생성함
  - 이때, sim1을 인자로 받을 수 있는 형태의 생성자를 호출함을 의미!
  - 즉, 대입연산(=)에서 해야 할 일을 생성자를 통해 명시하도록 문법을 만듬

복사생성자

복사생성자의 이해 - C++의 초기화

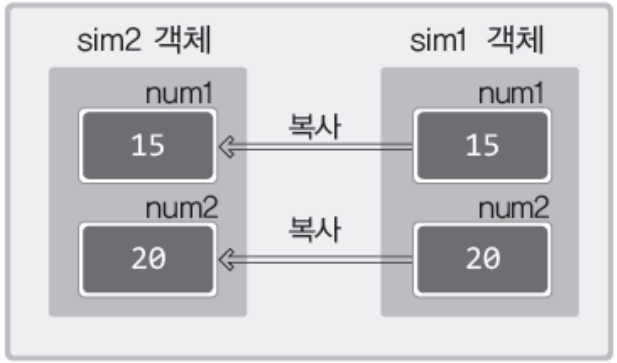
- 대입연산 의미처럼 멤버간 복사 진행

```
class SoSimple
{
private:
    int num1;
    int num2;
public:
    SoSimple(int n1, int n2) : num1(n1), num2(n2)
    { }
    void ShowSimpleData()
    {
        cout<<num1<<endl;
        cout<<num2<<endl;
    }
};
```

ⓘ 생성자

```
int main(void)
{
    SoSimple sim1(15, 20);
    SoSimple sim2=sim1;
    sim2.ShowSimpleData();
    return 0;
}
```

ⓘ 객체간 대입연산
SoSimple sim2(sim1); 수행
ⓘ But, 다음 형태의 생성자 없음
SoSimple(const S_Simple &ref)
{
…
}
ⓘ 이 경우, "**디폴트 복사생성자**" 자동생성
(객체 멤버간 데이터 복사)



복사생성자

복사생성자의 이해 - 객체의 대입연산

- SoSimple sim2(sim1); 의미!
  - 객체 이름을 sim2로 하는 SoSimple형 객체 생성
  - Sim1을 인자로 받을 수 있는 생성자 호출하여 객체생성 완성

```
class SoSimple
{
    . . . .
public:
    SoSimple(int n1, int n2)
        : num1(n1), num2(n2)
    {
        // empty
    }
    SoSimple(SoSimple &copy)
        : num1(copy.num1), num2(copy.num2)
    {
        cout<<"Called SoSimple(SoSimple &copy)"<<endl;
    }
    . . . .
};
```

ⓘ 복사생성자

ⓘ Sim1 값을 sim2에 복사한 후 sim1데이터가 변형되지 않아야 하므로 const선언 필요

```
int main(void)
{
    SoSimple sim1(15, 30);
    cout<<"생성 및 초기화 직전"<<endl;
    SoSimple sim2=sim1;    // SoSimple sim2(sim1); 으로 변환!
    cout<<"생성 및 초기화 직후"<<endl;
    sim2.ShowSimpleData();
    return 0;
}
```

ⓘ 왼쪽과 같이 묵시적으로 해석!

실행 결과

```
생성 및 초기화 직전
Called SoSimple(SoSimple &copy)
생성 및 초기화 직후
15
30
```

복사생성자

복사생성자의 이해 - 디폴트 복사생성자

- 복사생성자가 미정의될 경우, 디폴트 복사생성자 자동 삽입
  - 객체의 멤버 대 멤버로 복사가 진행

```
class SoSimple
{
private:
    int num1;
    int num2;
public:
    SoSimple(int n1, int n2) : num1(n1), num2(n2)
    { }
    . . . .
};
```

자동생성

```
class SoSimple
{
private:
    int num1;
    int num2;
public:
    SoSimple(int n1, int n2) : num1(n1), num2(n2)
    { }
    SoSimple(const SoSimple &copy) : num1(copy.num1), num2(copy.num2)
    { }
};
```

ⓘ 복사생성자

복사생성자

복사생성자의 이해 - explicit

- **묵시적 형 변환을 막아 실수를 방지**

```
SoSimple sim2=sim1;
```

묵시적 형 변환 →

```
SoSimple sim2(sim1);
```

```
explicit SoSimple(const SoSimple &copy)
            : num1(copy.num1), num2(copy.num2)
{
    // empty!!
}
```

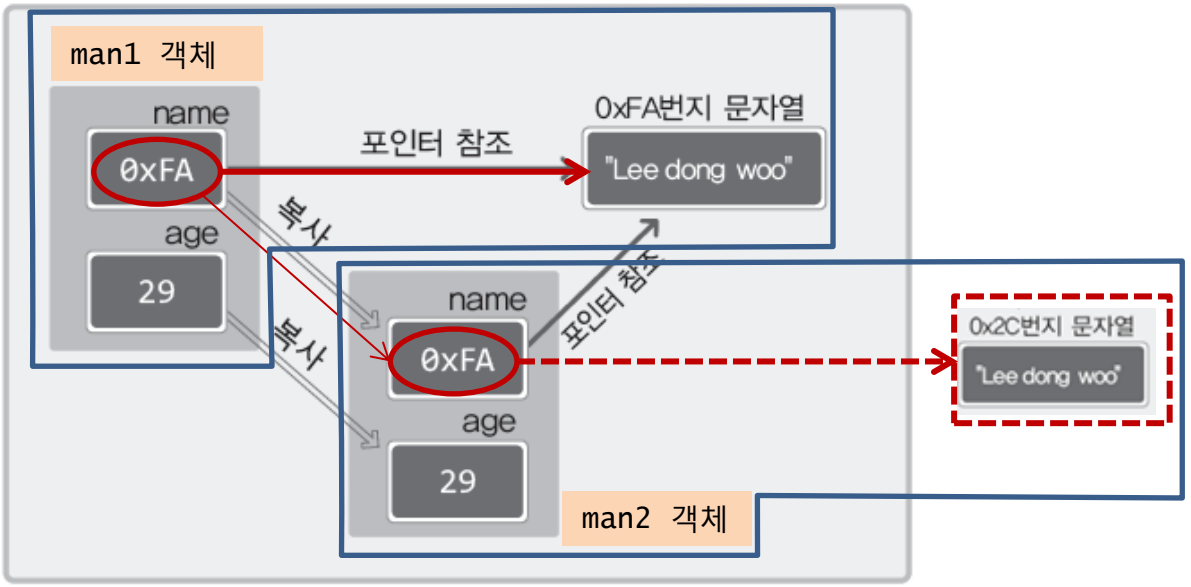
```
class AAA
{
private:
    int num;
public:
explicit AAA(int n) : num(n) { }
    . . . . .
};
```

ⓘ explicit 추가시,
  AAA obj=3;
  AAA obj(3) 묵시적 형 변환 방지
ⓘ 즉, AAA obj=3;과 같은 형태로 객체생성 불가

복사생성자

얕은 복사(Swallow Copy)

- 컴파일러에 의해 제공되는 '디폴트 복사생성자'에 의해 멤버변수 간에 값이 복사되는 것
  - 디폴트 복사생성자의 문제: 객체 소멸 시, 문제가 발생할 수 있음

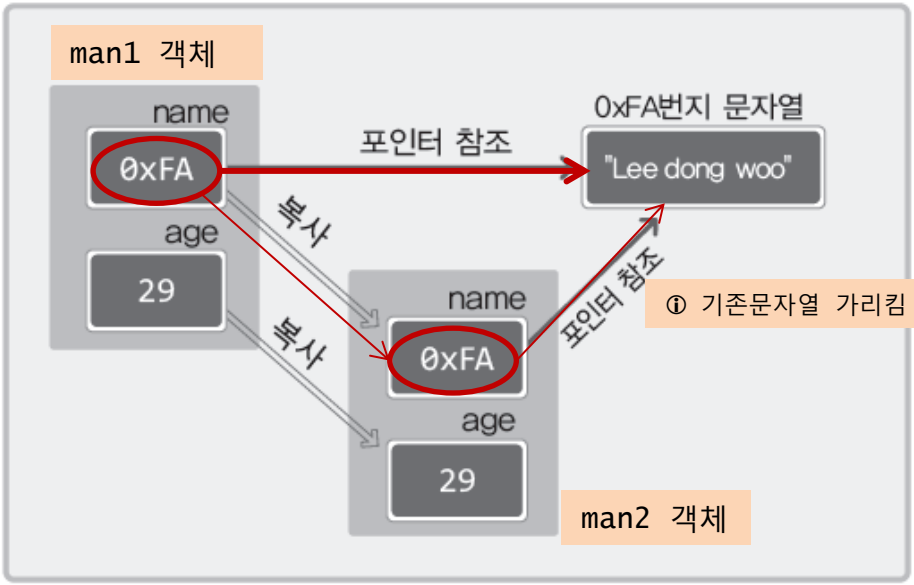


복사생성자

얕은 복사(Swallow Copy)

- 컴파일러에 의해 제공되는 '디폴트 복사생성자'에 의해 멤버변수 간에 값이 복사되는 것
  - 디폴트 복사생성자의 문제: 객체 소멸 시, 문제가 발생할 수 있음
    → 특히 멤버변수가 포인터일 때 문제 발생(얕은 복사 문제!)

- 객체 소멸 시 문제
  - (man1 소멸 시) man1 객체 소멸되고 이 객체가 가리키는 문자열 공간도 소멸됨
  - (man2 소멸 시) 소멸자 호출됨. 이때, 0xFA 주소값에 대해 소멸자 호출되면 이미 지운 공간이므로 메모리 오류가 발생함
  - 즉, 객체 소멸 불완전 발생!



복사생성자

얕은 복사(Shallow Copy)

- **디폴트 복사생성자의 문제**

```
int main(void)
{
    Person man1("Lee dong woo", 29);
    Person man2=man1;
    man1.ShowPersonInfo();
    man2.ShowPersonInfo();
    return 0;
}
```

실행 결과

```
이름: Lee dong woo
나이: 29
이름: Lee dong woo
나이: 29
Called destructor!
```

ⓘ 객체소멸 불완전

```
class Person
{
private:
    char * name;
    int age;
public:
    Person(char * myname, int myage)
    {
        int len=strlen(myname)+1;
        name=new char[len];
        strcpy(name, myname);
        age=myage;
    }
    . . . .
    ~Person()
    {
        delete []name;
        cout<<"called destructor!"<<endl;
    }
};
```

ⓘ 생성자에서 동적할당

ⓘ 복사생성자 없는 형태
ⓘ 따라서 디폴트 복사생성자 자동 생성!

복사생성자

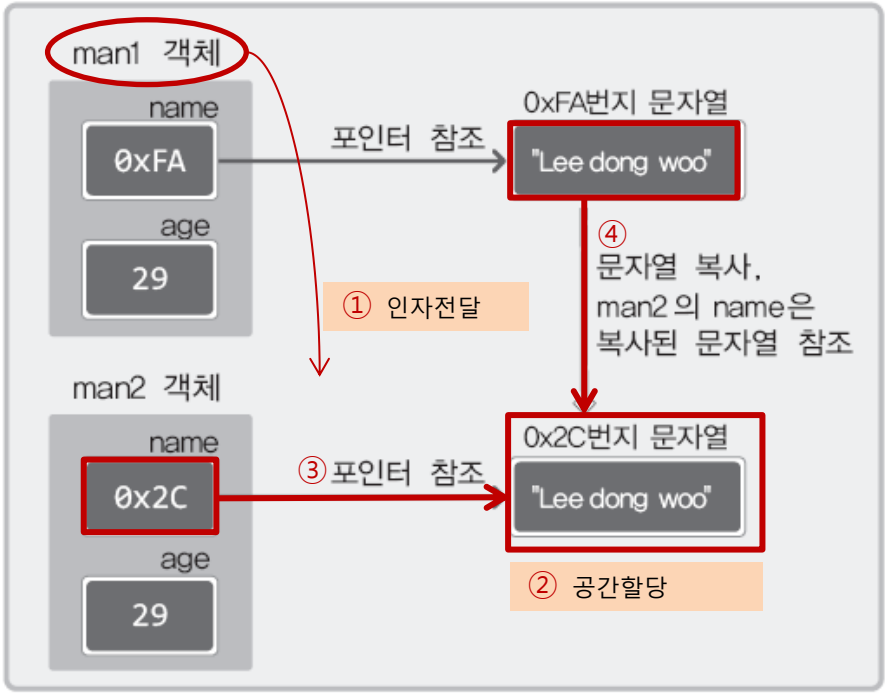
깊은 복사(Deep Copy)

- 얕은 복사 문제해결을 위해 깊은 복사 가능하도록 복사생성자 정의

```
Person(const Person& copy) : age(copy.age)
{
    name=new char[strlen(copy.name)+1];
    strcpy(name, copy.name);
}
```

ⓘ 생성자의 동적할당 코드 활용

* Man2 객체 생성시
① Man1 객체의 인자 전달
② 문자열 복사할 공간할당
③ Name에 새로 할당한 공간의 주소 복사
④ 문자열 복사



복사생성자

복사생성자의 구현 - 호출 시점

- **복사생성자가 호출되는 3가지 경우**
  - 메모리공간에 할당과 초기화가 동시에 일어나는 경우

사례1) 기 생성된 객체를 이용하여 신규 객체를 초기화

```
Person man1("Lee dong woo", 29);
Person man2=man1;①
```

사례2) 함수호출 시 객체를 인자로 전달하고, 값으로 받는 경우(Call-by-value)

사례3) 객체 반환 시, 참조형이 아닌 값으로 반환하는 경우

복사생성자

복사생성자의 구현 - 호출 시점

- **복사생성자가 호출되는 3가지 경우**
  - 사례2) 함수호출 시 객체를 인자로 전달하고, 값으로 받는 경우(Call-by-value)
  - 사례3) 객체 반환 시, 참조형이 아닌 값으로 반환하는 경우

* 일반 자료형의 경우

```
int SimpleFunc(int n)
{
    …
    return n;
}
int main(void)
{
    int num = 10;
    cout<<SimpleSunc(num)<<endl;
    …
}
```

반환 시, ③ 메모리공간 할당 동시에 초기화

② 인자 전달 시, 선언과 동시에 초기화

ⓘ 반환값은 register 값으로 전달됨
ⓘ 메모리에 이 값이 임시변수로 저장된 후 출력됨

* 객체의 경우

```
SoSimple SimpleFuncObj(SoSimple ob)
{
    . . . .
    return ob;
}
int main(void)
{
    SoSimple obj;
    SimpleFuncObj(obj);
    . . . .
}
```

③ 객체 반환 시, 복사생성자 호출

② 객체를 인자로 전달 시, 복사생성자 호출

복사생성자

복사생성자의 구현 - 호출 시점

- 사례2) Call-by-value 방식 함수호출 시 객체를 인자로 전달받은 경우

```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    { }
    SoSimple(const SoSimple& copy) : num(copy.num)
    {
        cout<<"Called SoSimple(const SoSimple& copy)"<<endl;
    }
    void ShowData()
    {
        cout<<"num: "<<num<<endl;
    }
};

void SimpleFuncObj(SoSimple ob)
{
    ob.ShowData();
}
```

③ 복사생성자 호출

② 인자 전달 시,
객체ob위한 메모리공간 할당

```
int main(void)
{
    SoSimple obj(7);
    cout<<"함수호출 전"<<endl;
    SimpleFuncObj(obj);
    cout<<"함수호출 후"<<endl;
    return 0;
}
```

① 생성자 호출

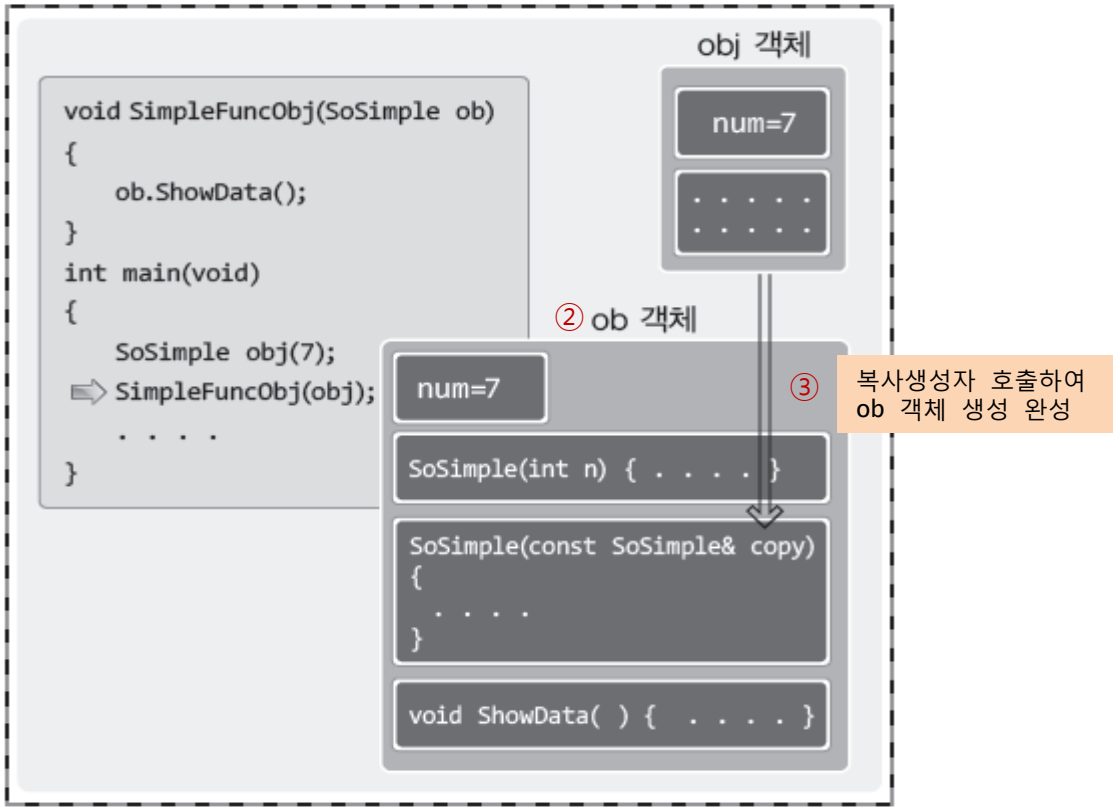
실행 결과

```
함수호출 전
Called SoSimple(const SoSimple& copy)
Num: 7
함수호출 후
```

복사생성자

복사생성자의 구현 - 호출 시점

- 사례2) Call-by-value 방식 함수호출 시 객체를 인자로 전달받은 경우



복사생성자

복사생성자의 구현 - 호출 시점

- 사례3) 객체 반환 시, 참조형으로 반환하지 않는 경우

```
int main(void)
{
    SoSimple obj(7);
    SimpleFuncObj(obj).AddNum(30).ShowData();
    obj.ShowData();
    return 0;
}
```

ob 객체 전달되며,
임시객체의 복사생성자 호출 ③

```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    { }
    SoSimple(const SoSimple& copy) : num(copy.num)
    {
        cout<<"Called SoSimple(const SoSimple& copy)"<<endl;
    }
    SoSimple& AddNum(int n)
    {
        num+=n;
        return *this;
    }
    void ShowData()
    {
        cout<<"num: "<<num<<endl;
    }
};

SoSimple SimpleFuncObj(SoSimple ob)
{
    cout<<"return 이전"<<endl;
    return ob;
}
```

obj 전달되며,
② ob의 복사생성자 호출

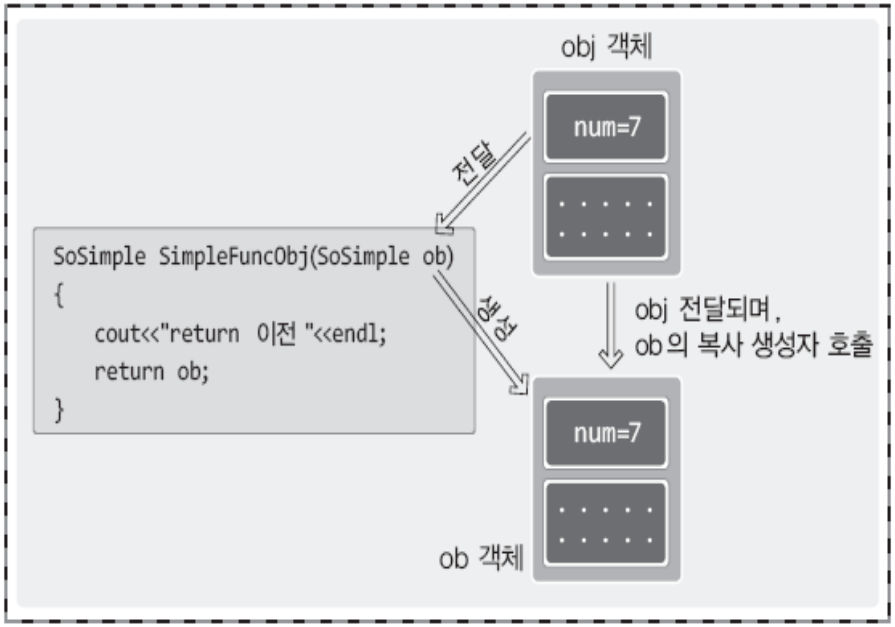
실행 결과

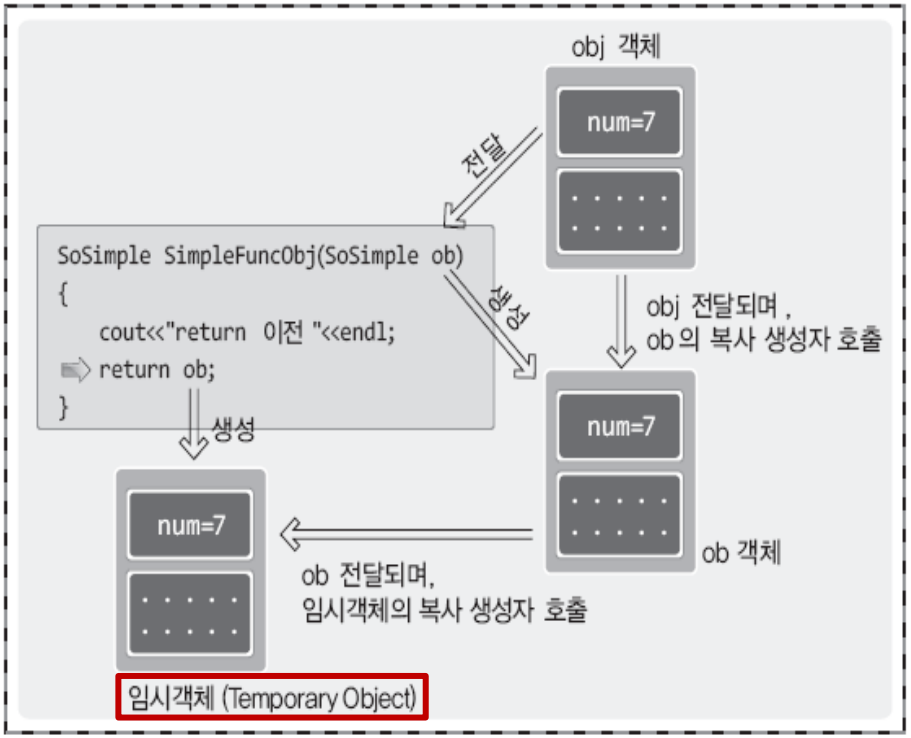
```
Called SoSimple(const SoSimple& copy)
return 이전
Called SoSimple(const SoSimple& copy)
num: 37
num: 7
```

복사생성자

복사생성자의 구현 - 호출 시점

- 사례3) 객체 반환 시, 참조형으로 반환하지 않는 경우





ⓘ 임시객체는 임시상수와 같이 반환 후 소멸
ex) int num = 3+4;
ex) cout<<fct(); // 반환값도 임시상수

복사생성자

복사생성자의 구현 - 객체소멸 시점

- **임시객체의 소멸**
  - 임시객체는 바로 다음 행에서 소멸
  - 단, 참조된 형태의 임시객체는 프로그램 종료 시 소멸

```
class Temporary
{
private:
    int num;
public:
    Temporary(int n) : num(n)
    {
        cout<<"create obj: "<<num<<endl;
    }
    ~Temporary()
    {
        cout<<"destroy obj: "<<num<<endl;
    }
    void ShowTempInfo()
    {
        cout<<"My num is "<<num<<endl;
    }
};
```

(임시객체의 참조 값).ShowTempInfo();

ⓘ 참조자 &ref가 참조하고 있어 main함수 종료시까지 유지됨
ⓘ 임시객체에 별칭을 붙인 개념으로 이해

```
int main(void)
{
    Temporary(100);
    cout<<"********** after make!"<<endl<<endl;

    Temporary(200).ShowTempInfo();
    cout<<"********** after make!"<<endl<<endl;

    const Temporary &ref=Temporary(300);
    cout<<"********** end of main!"<<endl<<endl;

    return 0;
}
```

실행 결과

```
create obj: 100
destroy obj: 100
********** after make!

create obj: 200
My num is 200
destroy obj: 200
********** after make!

create obj: 300
********** end of main!

destroy obj: 300
```

III. const객체와 friend함수

const객체와 friend함수

const객체 - 함수호출

- const객체로 선언된 객체의 데이터(멤버변수) 변경을 불허
  - const로 선언된 객체를 대상으로는 const 선언되지 않은 멤버함수 호출이 불가능함

```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    { }
    SoSimple& AddNum(int n)
    {
        num+=n;
        return *this;
    }
    void ShowData() const
    {
        cout<<"num: "<<num<<endl;
    }
};
```

```
int main(void)
{
    const SoSimple obj(7);
    // obj.AddNum(20);
    obj.ShowData();
    return 0;
}
```

ⓘ 객체의 데이터(멤버변수) 변경을 불허

ⓘ Const객체인 obj는 const함수가 아닌 AddNum함수 호출불가

const객체와 friend함수

const객체 - 함수 오버로딩

- const객체/참조자 대상 멤버함수 호출시, const 선언된 멤버함수 호출
  - 함수 오버로딩 조건
    - 매개변수의 개수 및 타입
    - 함수의 const선언 유무

```
void YourFunc(const SoSimple &obj)
{
    obj.SimpleFunc();
}
int main(void)
{
    SoSimple obj1(2);
    const SoSimple obj2(7);
    obj1.SimpleFunc();
    obj2.SimpleFunc();
    YourFunc(obj1);
    YourFunc(obj2);
    return 0;
}
```

ⓘ 전달형태에 무관하게 const참조자 형태로 받아 const함수 호출

```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    { }
    SoSimple& AddNum(int n)
    {
        num+=n;
        return *this;
    }
    void SimpleFunc()
    {
        cout<<"SimpleFunc: "<<num<<endl;
    }
    void SimpleFunc() const
    {
        cout<<"const SimpleFunc: "<<num<<endl;
    }
};
```

ⓘ 함수의 오버로딩 가능!

실행 결과

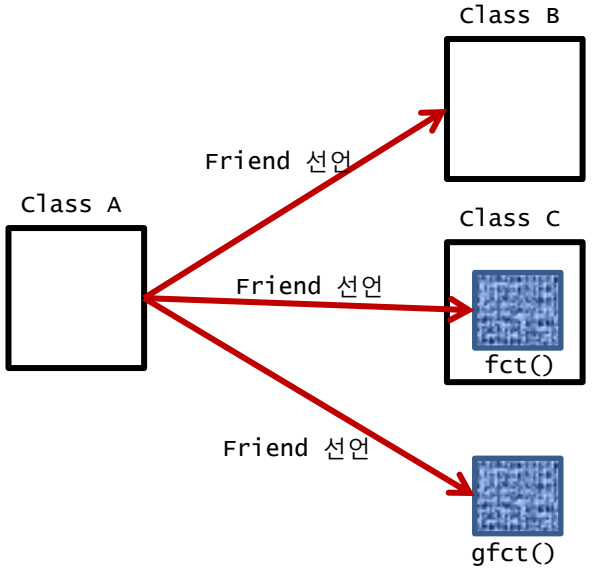
```
SimpleFunc: 2
const SimpleFunc: 7
const SimpleFunc: 2
const SimpleFunc: 7
```

const객체와 friend함수

클래스의 friend 선언

- friend선언으로 내 클래스의 private 멤버변수에 대한 직접 접근 허용
  - 반대의 경우는 성립 안되므로 원할 경우 역방향도 선언해야 함
  - 정보은닉에 반하는 선언(절대적으로 사용 지양)
    - 연산자 오버로딩에서 일부 사용 사례

- friend선언의 유형
  - 다른 클래스를 대상으로 선언
  - 다른 클래스 내의 멤버함수를 대상으로 선언
  - 전역함수를 대상으로 선언



const객체와 friend함수

클래스의 friend 선언

- 멤버변수를 대상으로 선언하여 private 멤버의 접근을 허용

```
class Boy
{
private:
    int height;
    friend class Girl;
public:
    Boy(int len) : height(len)
    { }
    . . . . .
};
```

```
class Girl
{
private:
    char phNum[20];
public:
    Girl(char * num)
    {
        strcpy(phNum, num);
    }
    void ShowYourFriendInfo(Boy &frn)
    {
        cout<<"His height: "<<frn.height<<endl;
    }
};
```

ⓘ Girl이 Boy의 friend이므로 private 멤버 height에 직접접근 가능

const객체와 friend함수

friend 함수 선언

- 멤버함수/전역함수 대상으로 선언하여 private 멤버 접근을 허용

```
class Point
{
private:
    int x;
    int y;
public:
    Point(const int &xpos, const int &ypos) : x(xpos), y(ypos)
    { }
    friend Point PointOP::PointAdd(const Point&, const Point&);
    friend Point PointOP::PointSub(const Point&, const Point&);
    friend void ShowPointPos(const Point&);
};
```

```
void ShowPointPos(const Point& pos)
{
    cout<<"x: "<<pos.x<<", ";
    cout<<"y: "<<pos.y<<endl;
}
```

```
Point PointOP::PointAdd(const Point& pnt1, const Point& pnt2)
{
    opcnt++;
    return Point(pnt1.x+pnt2.x, pnt1.y+pnt2.y);
}
Point PointOP::PointSub(const Point& pnt1, const Point& pnt2)
{
    opcnt++;
    return Point(pnt1.x-pnt2.x, pnt1.y-pnt2.y);
}
```

V. 실습

감사합니다

mnshim@sungkyul.ac.kr