



C++ 프로그래밍

김 형 기

hk.kim@jbnu.ac.kr

Lecture Plan

- OOP 개요
 - 클래스와 객체
 - 생성자와 소멸자
 - 생성자 오버로딩
- Part 1
-
- 복사 생성자
 - this, const, static, friend
- Part 2

복사 생성자

Copy Constructor

- 복사 생성자 개요
- 기본 복사 생성자
- 얕은 복사 vs 깊은 복사

- 아래 코드의 p 메모리 주소로 가보면, 오른쪽과 같음
- 즉, 객체를 메모리에 생성하면 멤버 변수의 값이 스택이나 힙에 저장되는 것
 - 멤버 함수는 코드 공간에 존재하며 접근 제한에 따라 일반 함수처럼 호출될 뿐임

```
1 class Point
2 {
3 private:
4     int x;
5     int y;
6 public:
7     Point(int x, int y)
8         : x{ x }, y{ y }
9     {}
10 };
11
12 int main()
13 {
14     Point p{ 2,3 };
15
16 }
```

0x003DFAA0	cc	cc	cc	cc	cc	cc	cc	cc	cc	cc
0x003DFAE0	cc	cc	cc	cc	cc	cc	cc	cc	cc	cc
0x003DFB20	02	00	00	00	03	00	00	00	cc	cc
0x003DFB60	00	00	00	00	00	00	00	00	00	00
0x003DFBA0	a8	fb	3d	00	6d	1e	29	00	b0	fb

● 복사 생성자

- 객체가 복사될 때는, 기존 객체를 기반으로 새로운 객체가 생성됨
 - 객체가 복사되는 경우
 1. 객체를 **pass by value** 방식으로 함수의 매개변수로 전달할 때
 2. 함수에서 **value**의 형태로 결과를 반환할 때
 3. 기존 객체를 기반으로 새로운 객체를 생성할 때
- } 함수 강의에서 "변수가 복사되어 전달된다" 와 동일한 내용
- 이와 같은 상황이 발생할 때, 객체가 "어떻게" 복사될 지 정의해 주어야 함
 - 왜? 내가 만든 클래스이기 때문에, 잘못 복사했다가는 어떤 일이 발생할지 모름
 - 하지만 일단 복사가 아예 안되지는 않도록, 컴파일러에서 자동으로 복사 생성자를 만들어 줌

● 왜 중요한가?

- 복사 과정에서 문제가 발생할 수 있음(포인터가 존재하는 경우)
- 복사 비용에 대한 고려(클래스는 많은 데이터를 포함할 수 있음)

Copy Constructor

- 객체가 복사되는 경우

1. Pass by value로 객체 전달 예시

```
1 void DisplayPlayer(Player p)
2 {
3     p.Print(); //p는 main()의 hero 지역 객체의 사본
4 } //p 소멸자 호출 시점
5
6 int main()
7 {
8     Player hero {0, 0, 1};
9     DisplayPlayer(hero);
10 }
```

Copy Constructor

- 객체가 복사되는 경우
 1. Value의 형태로 결과값 반환

```
1 Player CreateSuperPlayer()
2 {
3     Player superPlayer{1, 1, 1};
4     return superPlayer; //superPlayer객체가 복사되어 main()의 player 지역 객체로 전달
5 } //superPlayer 소멸자 호출 시점
6
7 Player player;
8 player = CreateSuperPlayer();
```


Copy Constructor

- 객체가 복사되는 경우

- 3. 기존 객체를 기반으로 새로운 객체를 생성

```
1 Player hero {1, 1, 1};  
2  
3 Player anotherHero = hero; //hero를 복사해서 anotherHero를 만듦  
4 //또는 Player anotherHero {hero};
```

- 이와 같이 복사가 일어날 때, 정확히 어떻게 복사본을 만들어야 하는지를 클래스를 정의한 여러분이 알려주어야 할 필요가 있음
 - 기본 데이터 타입(int, float 등)은 이미 정해진 방식이 있음

- 자동 생성되는 복사 생성자

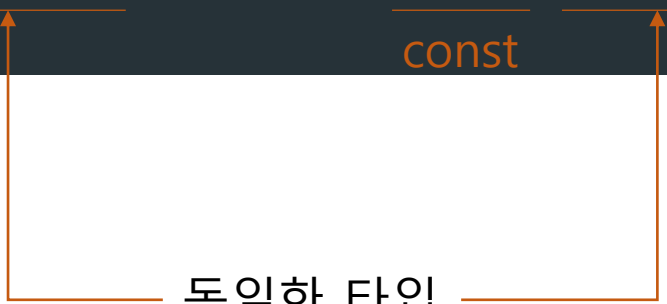
- 사용자가 복사 생성자를 구현하지 않으면, 기본 복사 생성자가 자동으로 만들어져 사용됨
- 멤버 변수들의 값을 복사하여 대입하는 방식!
- **포인터 타입의 멤버 변수가 존재할 때는 주의!**
 - 기본 복사 생성자는 포인터 타입의 변수 또한 복사하여 대입됨
 - 즉, 포인터가 가리키는 데이터의 복사가 아닌 포인터 주소값의 복사!
- **얕은 복사(shallow copy) vs 깊은 복사(deep copy)**

Declaring Copy Constructor

- 복사 생성자의 선언
 - 동일한 타입의 const 참조자가 인자인 생성자

```
1 Type::Type(const Type &source);  
2  
3 Player::Player(const Player &source); //Player 클래스의 복사 생성자  
4 Account::Account(const Account &source); //Account 클래스의 복사 생성자
```

const 참조자



동일한 타입

Inline 이라면

```
1 Player(const Player &source); //Player 클래스의 복사 생성자  
2 Account(const Account &source); //Account 클래스의 복사 생성자
```

Implementing Copy Constructor

- 얇은 복사 생성자의 구현

- 자동 생성되는 복사 생성자는 “얇은 복사” 수행. 그 과정은 아래와 동일함
- 생성자 초기화 리스트를 사용한 방법

```
1 Player(const Player &other)
2     :x{ other.x }, y{ other.y }, speed{other.speed}
3 {
4 }
```

- 대입을 사용한 방법

```
1 Player(const Player &other)
2 {
3     x = other.x;
4     y = other.y;
5     speed = other.speed;
6 }
```

**자동 생성된 복사 생성자 실제 명령문*

```
1 Player(const Player& other)
2 {
3     memcpy(this, &other, sizeof(other));
4 }
```

- 얕은 복사의 문제점

클래스 정의

```
1 class Shallow
2 {
3 private:
4     int *data; //Pointer
5 public:
6     Shallow(int d); //Constructor
7     Shallow(const Shallow &source); //Copy Constructor
8     ~Shallow();
9 };
```

생성자

```
1 Shallow::Shallow(int d)
2 {
3     data = new int;
4     *data = d;
5 }
```

소멸자

```
1 Shallow::~~Shallow()
2 {
3     delete data;
4     cout << "free storage" << endl;
5 }
6
```

Shallow vs Deep Copy

- 얕은 복사의 문제점

- 포인터가 가리키는 데이터가 아닌 포인터 주소값 자체가 복사됨!

```
1 Shallow::Shallow(const Shallow &source)
2   :data{source.data}
3 {
4   cout << "Copy constructor, shallow" << endl;
5 }
```

(얕은) 복사 생성자

Source 객체의 data에 저장된
"주소값"이 현재 객체의 data에
복사됨

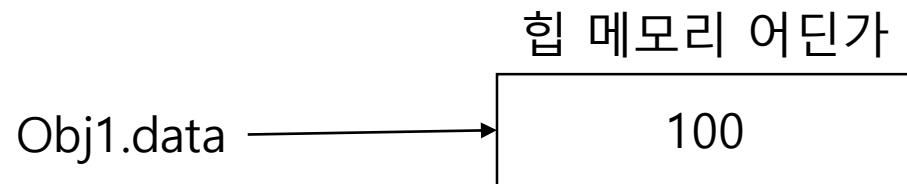
Shallow vs Deep Copy

- 얕은 복사의 문제점

- 포인터가 가리키는 데이터가 아닌 포인터 주소값의 복사!

```
1 int main()
2 {
3     Shallow obj1{ 100 };
4     DisplayShallow(obj1); //copy of obj1 is release data
5     return 0;
6 } //obj1 try to release data again!
```

사용 예시



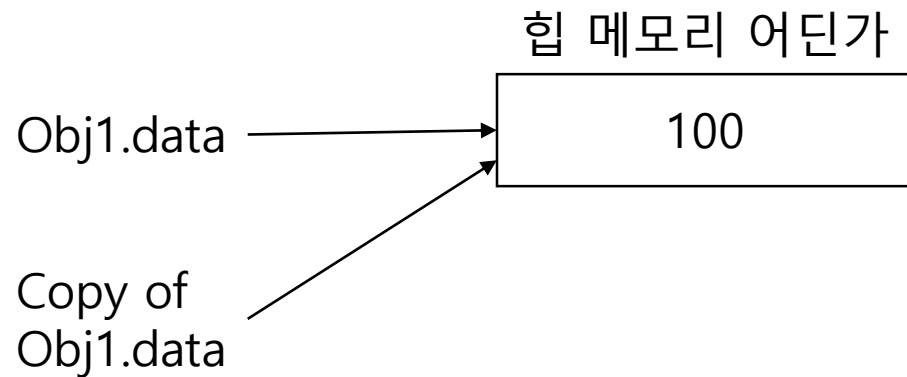
Shallow vs Deep Copy

- 얕은 복사의 문제점

- 포인터가 가리키는 데이터가 아닌 포인터 주소값의 복사!

```
1 int main()
2 {
3     Shallow obj1{ 100 };
4     DisplayShallow(obj1); //copy of obj1 is release data
5     return 0;
6 } //obj1 try to release data again!
```

사용 예시



(DisplayShallow 함수 내부에서 복사 생성된 객체의 data 멤버변수)

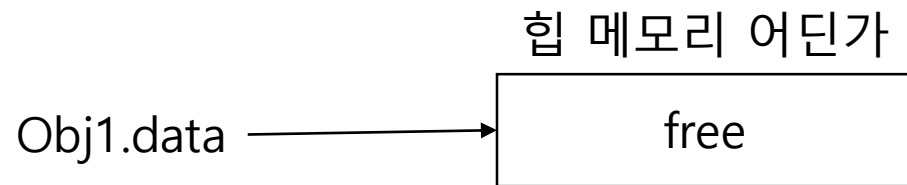
Shallow vs Deep Copy

- 얕은 복사의 문제점

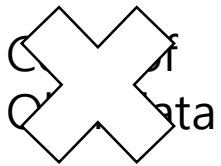
- 포인터가 가리키는 데이터가 아닌 포인터 주소값의 복사!

```
1 int main()
2 {
3     Shallow obj1{ 100 };
4     DisplayShallow(obj1); //copy of obj1 is release data
5     return 0;
6 } //obj1 try to release data again!
```

사용 예시



(DisplayShallow 함수가 끝나면서 소멸자 자동호출)



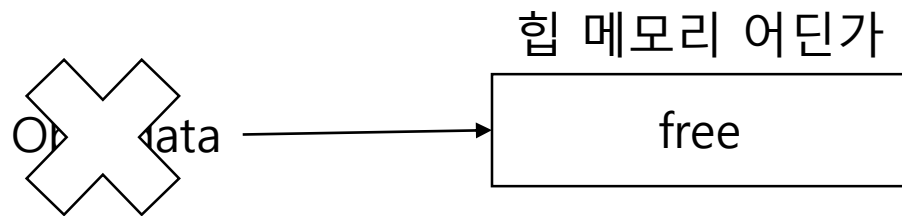
Shallow vs Deep Copy

- 얕은 복사의 문제점

- 포인터가 가리키는 데이터가 아닌 포인터 주소값의 복사!

```
1 int main()
2 {
3     Shallow obj1{ 100 };
4     DisplayShallow(obj1); //copy of obj1 is release data
5     return 0;
6 } //obj1 try to release data again!
```

사용 예시



(main 함수가 끝나면서 obj1의 소멸자 호출)
(해제된 공간을 다시 해제하려 시도! → 에러)

Shallow vs Deep Copy

- 깊은 복사

```
1 class Deep
2 {
3 private:
4     int* data; //Pointer
5 public:
6     Deep(int d); //Constructor
7     Deep(const Deep &source); //Copy Constructor
8     ~Deep();
9 };
```

클래스 정의

```
1 Deep::Deep(int d)
2 {
3     data = new int;
4     *data = d;
5 }
```

생성자

```
1 Deep::~~Deep()
2 {
3     delete data;
4     cout << "free storage" << endl;
5 }
```

소멸자

Shallow vs Deep Copy

● 깊은 복사

- 주소값을 복사하는 것이 아니라, 데이터를 복사하여 복사 생성하는 방식
- 즉, 복사 생성자가 **새로운 힙 공간을 할당한 뒤 동일한 데이터 복사**
 - P.14와의 차이점을 확인할 것

```
1 Deep::Deep(const Deep &source)
2 {
3     data = new int; //allocate new storage
4     *data = *source.data; //dereferencing
5     cout << "Copy constructor, deep" << endl;
6 }
```

복사 생성된 객체의 데이터를 저장하기
위해 새로운 힙 메모리 할당

깊은 복사 생성자
(기본 버전)

깊은 복사 생성자
(생성자 위임 사용 버전)

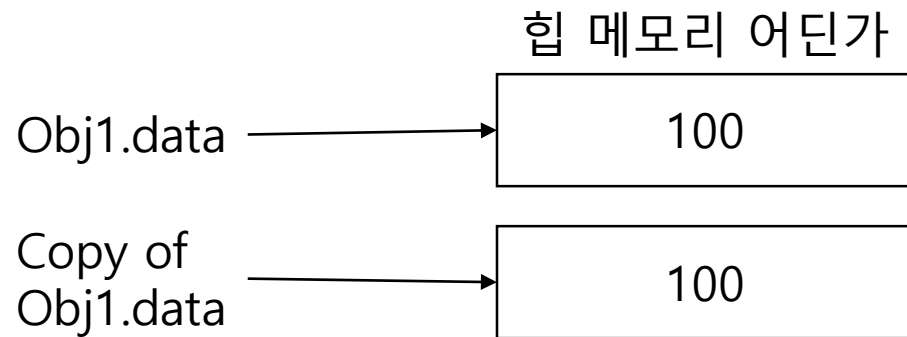
```
1 Deep::Deep(const Deep &source)
2     : Deep{*source.data}
3 {
4     cout << "Copy constructor, deep" << endl;
5 }
```

Shallow vs Deep Copy

- 깊은 복사의 경우...
 - 데이터도 복사하므로 이중 해제의 문제 해결

```
1 int main()
2 {
3     Deep obj1{ 100 };
4     DisplayDeep(obj1); //copy of obj1 is release data
5     return 0;
6 } //obj1 try to release data again!
```

사용 예시



(DisplayDeep 함수 내부에서 복사 생성할 때, 새로운 힙 공간 할당!)

Copy Constructor

- 복사 생성자 잘 사용하는 법

- 포인터 타입의 멤버 변수가 존재할 때는 (깊은) 복사 생성자 직접 구현
 - 새로운 heap 공간을 할당하여 값을 복사해 두어야 한다는 것을 명심
- *STL / smart pointer 사용*
 - *STL은 내부적으로 안전한 복사 생성자가 구현되어 있음*
 - *Smart pointer는 복사를 허용하지 않거나(unique) 참조되는 동안은 해제하지 않음(shared)*

(Summary) Copy Constructor

- 복사 생성자 개요

- 매개변수 전달, value 반환, 기존 객체로 새 객체 생성시 복사 생성자 호출

- 기본 복사 생성자

- 컴파일러가 만들어주는 기본 복사 생성자는 멤버 변수를 그대로 대입(얕은 복사)할 뿐
- 얕은 복사는 이중 해제 오류 발생

- 깊은 복사

- 새로운 힙공간을 할당하여 복사 생성

this, const, static, friend

this, const, static, friend

- this
- const
- static
- friend

this Pointer

- “this” 포인터

- this는 C++ 키워드
- 멤버 함수를 호출한 객체의 주소값
- 사용 예

```
1 void Player::SetPosition(int x, int y)
2 {
3     x = x; ← ?? 명확하지 않음
4     y = y;
5 }
6
7 void Player::SetPosition(int x, int y)
8 {
9     this→x = x;
10    this→y = y; ← 좌변은 멤버 변수, 우변은 인자로 명확
11 }
```

this Pointer

- “this” 포인터

- This는 C++ 키워드
- 멤버 함수를 호출한 객체의 주소값
- 확인해보기

```
1 class Player
2 {
3     private:
4         int x, y;
5         int speed;
6     public:
7         Player(int x,int y,int speed)
8             : x{ x }, y{ y }, speed{ speed }
9         {
10             cout << this << endl;
11         }
12 };
13
14 int main()
15 {
16     Player* p = new Player{ 1,1,1 };
17     cout << p << endl;
18 }
```

this Pointer

- “this” 포인터

- This는 C++ 키워드
- 멤버 함수를 호출한 객체의 주소값
- 사용 예

```
1 void Player::ComparePosition(const Player& other)
2 {
3     if (this == &other)
4         cout << "Same Position" << endl;
5     ...
6 }
7 player1.ComparePosition(player1);
```

- (연산자 오버로딩 강의에서 자주 사용됨)

const Object

- const 객체

- Const-correctness
- Const 객체의 멤버 변수의 값은 변경 불가능

p가 const 객체이기 때문에, 멤버 변수의 값을 바꾸는 SetPosition() 멤버 함수는 컴파일러가 호출하지 못하도록 오류를 발생시킴

```
1 class Player
2 {
3 private:
4     int x, y;
5     int speed;
6 public:
7     Player(int x,int y,int speed)
8         : x{ x }, y{ y }, speed{ speed }
9     {
10         cout << this << endl;
11     }
12     void SetPosition(int x, int y)
13     {
14         this->x = x;
15         this->y = y;
16     }
17
18 };
19
20 int main()
21 {
22     const Player p{ 1,1,1 };
23     p.SetPosition(2, 2); //ERROR!
24 }
```

const Object

- const 객체

- Const-correctness
- Const 객체의 멤버 변수의 값은 변경 불가능
- 값을 변경하지 않을 때에는?

PrintPosition() 함수는 멤버 변수인 x와 y의 값을 바꾸지 않으니 괜찮은걸까?

```
1 class Player
2 {
3 private:
4     int x, y;
5     int speed;
6 public:
7     Player(int x,int y,int speed)
8         : x{ x }, y{ y }, speed{ speed }
9     {
10         cout << this << endl;
11     }
12     void SetPosition(int x, int y)
13     {
14         this->x = x;
15         this->y = y;
16     }
17     void PrintPosition()
18     {
19         cout << x << "," << y << endl;
20     }
21 };
22
23 int main()
24 {
25     const Player p{ 1,1,1 };
26     p.PrintPosition(); // ??
27 }
```

● const 객체

- Const-correctness
- PrintPosition()이 값을 변경하지 않는다는 것을 명시적으로 알려주어야 함
 - 그래서 const로 선언된 p 객체에서 값이 바뀌지 않는 것이 보장된 함수를 선별하여 호출이 가능함
- 값이 바뀌지 않는 것을 보장하는 법
 - 멤버 함수 선언 뒤에 const를 붙인다.

```
1 class Player
2 {
3 private:
4     int x, y;
5     int speed;
6 public:
7     Player(int x,int y,int speed)
8         : x{ x }, y{ y }, speed{ speed }
9     {
10         cout << this << endl;
11     }
12     void SetPosition(int x, int y)
13     {
14         this->x = x;
15         this->y = y;
16     }
17     void PrintPosition() const
18     {
19         //x = 10; //값을 바꾸려고 시도하면 ERROR!
20         cout << x << "," << y << endl;
21     }
22 };
23
24 int main()
25 {
26     const Player p{ 1,1,1 };
27     p.PrintPosition(); //OKAY!
28 }
```

Static Class Members

- static 클래스 멤버 변수
 - 객체가 아닌 클래스에 속하는 변수
 - 개별적인 객체의 데이터가 아닌 클래스에 공통 데이터 구현이 필요할 때 사용
- static 클래스 멤버 함수
 - 객체가 아닌 클래스에 속하는 함수
 - 클래스 이름 하에서 바로 호출 가능
 - **Static 클래스 멤버 함수는 static 클래스 멤버 변수에만 접근 가능**

Static Class Members

- static 클래스 멤버

```
1 class Player
2 {
3 private:
4     static int numPlayers;
5     ...
6 Public:
7     static int GetNumPlayers();
8     ...
9 };
```

Player.h

```
1 #include "Player.h"
2
3 int Player::numPlayers = 0;
4
5 ...
6
7 int Player::GetNumPlayers()
8 {
9     return numPlayers;
10 }
11
12 ...
```

Player.cpp

Static Class Members

- static 클래스 멤버

- Num_player를 증가시키는 위치에 주의!

- 생성자 위임 등을 통해 의도하지 않는 플레이어 숫자 증가가 발생하지 않아야 함

```
1 Player::Player(int x, int y, int speed)
2     : x{x}, y{y}, speed{speed}
3 {
4     numPlayers++;
5 }
6
7 Player::~~Player()
8 {
9     numPlayer--;
10 }
```

Player.cpp 파일

Static Class Members

- static 클래스 멤버

main.cpp

```
1 void DisplayActivePlayers()
2 {
3     cout << "Active players: " << Player:: GetNumPlayers() << endl;
4 }
5
6 int main()
7 {
8     DisplayActivePlayers(); // 0
9
10    Player obj1 {1,1,1};
11    DisplayActivePlayers(); // 1
12    ...
13 }
```

Static Class Members

- Static



Static 변수는 stack과 heap이 아닌 별도 메모리 공간에 저장

Friends of a Class

- friend 키워드

- Private 멤버에 대해 접근을 허용할 특정 함수나 클래스를 선언할 때 사용
- 비대칭
 - A가 B의 friend라고 B가 A의 friend는 아님
- 전이되지 않음
 - A가 B의 friend이고 B가 C의 friend라고, A가 C의 friend는 아님

Friends of a Class

- friend 클래스

```
1 class Player
2 {
3     friend void DisplayPlayer(const Player& p);
4 private:
5     int x, y;
6     int speed;
7 public:
8     Player(int x,int y,int speed)
9         : x{ x }, y{ y }, speed{ speed }
10    {
11        cout << this << endl;
12    }
13    void SetPosition(int x, int y)
14    {
15        this->x = x;
16        this->y = y;
17    }
18 };
19
20 void DisplayPlayer(const Player& p)    *Player::DisplayPlayer() 가 아님!!!
21 {
22     cout << p.x << "," << p.y << endl; //friend가 아니면 private에 접근 불가능
23 }
```

Friends of a Class

- friend 클래스

```
1 class Player
2 {
3     friend class OtherClass;
4 private:
5     int x, y;
6     int speed;
7 public:
8     ...
9 };
```

(Summary) this, const, static, friend

- this

- 현재 객체의 주소값을 명시적으로 표현

- const

- Const correctness를 위해 필요한 경우 멤버 함수 뒤에 const 명시 필요

- static

- 객체가 아닌 클래스에 속하는 변수/함수를 위한 키워드
- Ex. 플레이어 숫자의 관리

- friend

- Private에 접근을 허용해주는 키워드
- 꼭 필요할 때만 사용!