



# C++ 프로그래밍

김 형 기

[hk.kim@jbnu.ac.kr](mailto: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

# 객체지향 프로그래밍 개요

# (Summary) Object Oriented Programming

---

- 절차적 프로그래밍이란?
- 절차적 프로그래밍의 단점
- 객체지향 프로그래밍의 개념과 장점
- 객체지향 프로그래밍의 단점

# Procedural Programming

---

- 절차적 프로그래밍

- 프로그램이 수행하는 일련의 작업을 기준으로 하는 프로그래밍 패러다임
- 작업의 구현=함수 / 함수의 집합=프로그램
- 데이터와 작업이 분리되어 있는 개념
- 데이터는 작업의 실행을 위해 매개변수로 전달될 뿐
- 이해가 쉬운 방식

# Disadvantages of Procedural Programming

---

- 절차적 프로그래밍의 단점
  - 함수가 데이터의 구조를 정확히 알아야만 함
    - 데이터가 변하면, 함수의 수정이 필요
    - Tightly coupled
  - 프로그램의 규모가 커지면,
    - 이해하기 어렵고
    - 유지/보수하기 어렵고
    - 확장하기 어렵고
    - 디버깅하기 어렵고
    - 코드를 재사용하기 어렵고
    - 오동작할 확률이 커짐

# Object Oriented Programming

---

- 객체지향 프로그래밍

- 절차적 프로그래밍의 단점을 극복하기 위해 제안된 프로그래밍 패러다임 중의 하나
- C++, C#, Java 등에서는 이러한 방식을 손쉽게 구현할 수 있는 언어의 문법을 제공
  - 함수형 프로그래밍 등 새로운 패러다임을 적용할 수 있도록 언어는 계속 확장될 수 있음

- 클래스와 객체를 기반으로 함

- 데이터와 작업을 하나로 묶어서 표현

# Object Oriented Programming

---

## ● 객체지향 프로그래밍의 특징

### 1. 캡슐화

➤ 클래스는 (데이터) + (데이터를 기반으로 기능)을 모두 포함

### 2. 정보 은닉(추상화)

➤ 사용자는 내부 구현에 대해 알 필요도 없고, 알아서도 안됨

- 잘못된 사용 및 수정을 방지

➤ 사용자는 외부로 노출된 인터페이스만 활용 가능

➤ 테스트, 디버깅, 유지보수, 확장이 용이해짐

### 3. 상속

### 4. 다형성



# Disadvantages of Object Oriented Programming

---

- 절차적 프로그램의 상위 호환이 아니다!
  - 잘 설계된 절차적 프로그램 > 잘못 설계된 객체지향 프로그램
  - 모든 문제에 어울리는 설계 방안이 아님
  - 모든 대상이 클래스로 치환되는 것은 아님
- 객체지향 프로그램은 어렵다
  - 직관적이지 않을수도...
- 문제를 잘 분석하여 좋은 설계를 만들어야 함 (어렵다)
- 성능에서 손해를 보거나, 지나치게 복잡한 코드가 작성될 수 있음

# (Summary) Object Oriented Programming

---

- 절차적 프로그래밍
  - 데이터와 작업이 분리
- 절차적 프로그래밍의 한계
  - 작업이 데이터에 의존적(ex, 데이터가 변하면 함수를 수정해야 함)
- 객체지향 프로그래밍의 개념과 장점
  - 데이터 + 기능 캡슐화
  - 정보 은닉
  - 상속, 다형성...
- 객체지향 프로그래밍의 한계
  - 완벽한 상위호환은 없음
  - 성능 손실 가능성, 디자인에 더 많은 수고 필요

# 클래스와 객체

# (Summary) Class and Object

---

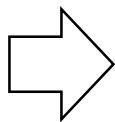
- 클래스란?
- 객체란?
- 클래스의 선언과 객체의 생성
- 접근 제한자
- 멤버 변수와 멤버 함수
- 명세와 구현의 분리
- 구조체 vs 클래스

# Classes and Objects

## ● Example

- 플레이어와 여러 명의 적이 존재하는 게임을 만든다면...

```
1 int main()
2 {
3     int playerPositionX, playerPositionY;
4     int playerSpeed;
5
6     int enemyPositionX, enemyPositionY;
7     int enemySpeed;
8
9     ...
10
11     Move(playerPositionX, playerPositionY)
12     Move(enemyPositionX, enemyPositionY)
13
14
15 }
```



```
1 class Player
2 {
3 public:
4     int x, y;
5     int speed;
6
7     void Move(int dx, int dy)
8     {
9         x += dx * speed;
10        y += dy * speed;
11    }
12 };
13
14 int main()
15 {
16     Player player1;
17     player1.x = 10; player1.y = 10; player1.speed = 2;
18     player1.Move(2, 3);
19
20     return 0;
21 }
```

(절차지향) 기존에 배운 바로는 플레이어 관련 변수, 적 관련 변수들을 만들고 이러한 변수들을 함수를 사용해 변경하는 방식으로 프로그램 작성

(객체지향) 플레이어와 관련된 데이터와 동작을 하나로 묶어 정의하고(클래스) 그로부터 변수(객체)를 만들어 사용

# Classes and Objects

- 클래스

- 객체(object)가 생성되기 위한 틀
  - 객체가 가져야 할 데이터와 기능을 정의
- 사용자 정의 "자료형"(user-defined data-type)

- **멤버 변수를 가짐(데이터)**

- 속성(property, attribute), 필드(field), 클래스 변수(class variable) ≈ 멤버 변수와 비슷한 용어

- **멤버 함수를 가짐(함수, 동작)**

- Method ≈ 멤버 함수의 또다른 용어

- 데이터와 함수를 은닉 가능
- 인터페이스를 공개 가능

# Classes and Objects

---

## ● 객체

- 클래스로부터 생성된 실체

- (메모리에 올라간 객체는 인스턴스로 구분하여 명명하는 경우도 있음)

- 객체는 개별적으로 관리되며, 원하는 만큼 생성 가능함

- **객체를 통해** 클래스에 정의된 멤버함수 호출, 멤버변수 접근 가능

- 개념적인 예

- "철수"는 학생이라는 클래스의 객체

- "영희"도 학생이라는 클래스의 객체

- 철수와 영희는 각각의 학과, 학번, 키, 몸무게, 나이 등의 멤버 변수(데이터)를 가지고 있음

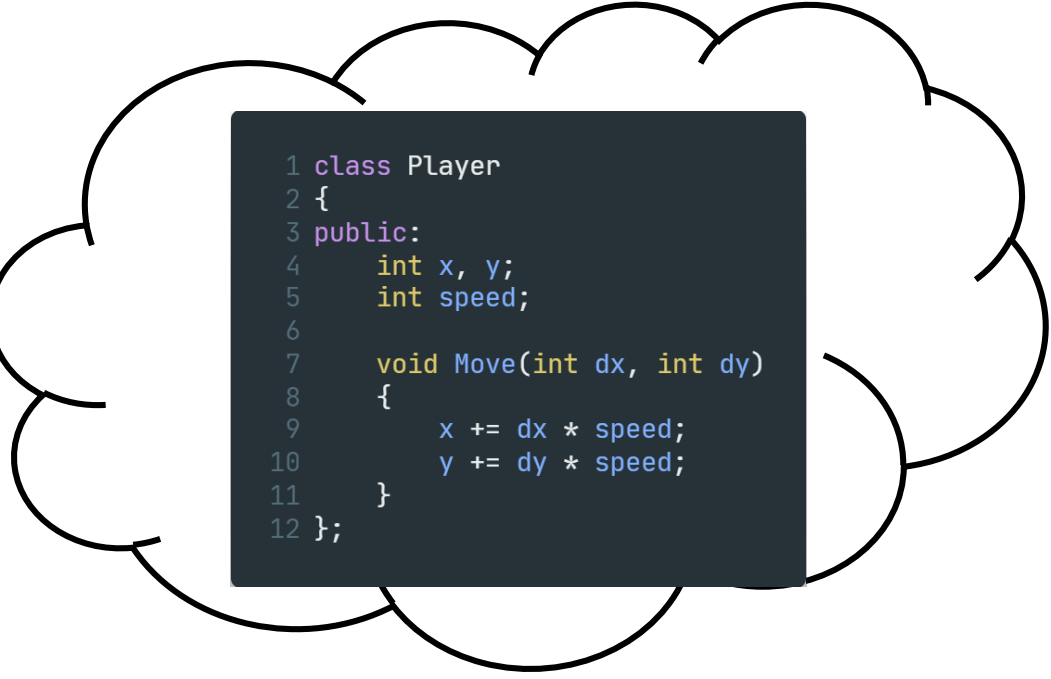
- 클래스와 객체

- 클래스 이름은 자료형(ex, int) 처럼 사용됨
- 클래스를 정의하는 것 = 새로운 데이터 타입을 만드는 것!
  - 우리가 사용할 실체는 "객체"로 만들어 메모리에 올려야 함

```
1 // 변수 타입, 띄고 변수 이름
2 int highScore;
3 int lowScore;
4
5 // 클래스 타입, 띄고 객체 이름
6 Player Kim;
7 Player Lee;
```



- 클래스와 객체



```
1 class Player
2 {
3 public:
4     int x, y;
5     int speed;
6
7     void Move(int dx, int dy)
8     {
9         x += dx * speed;
10        y += dy * speed;
11    }
12};
```

2) Player라는 클래스는 이러한 데이터들과 이러한 기능을 수행해야 해 라는 개념을 정의하는 부분일 뿐

1) 실제 프로그램이 동작할 때 메모리에 올라가는 객체들을 만들고 그 안에 포함된 멤버 변수와 멤버 함수를 사용하는 부분  
객체가 있어야만 멤버 변수와 멤버함수가 메모리상에 "존재" 하게 됨

```
1 int main()
2 {
3     Player player1;
4     player1.x = 10; player1.y = 10; player1.speed = 2;
5     player1.Move(2, 3);
6
7     Player player2;
8     player2.x = 30; player2.y = 50; player2.speed = 3;
9     player2.Move(2, 3);
10
11
12     return 0;
13 }
```

# Class Definition

- 클래스의 정의

- 기본 모양

```
1 class ClassName
2 {
3     //declarations;
4 };
```

- 예시

```
1 class Player
2 {
3     //member variable
4     std::string name;
5     int health;
6     int xp;
7
8     //member function
9     void Talk(std::string text);
10    bool IsDead();
11 };
```

} 멤버 변수, 데이터

} 멤버 함수, 행위/기능

# Create Object

- 객체의 생성

- 변수와 동일하게 스택 또는 힙 메모리에 선택적으로 생성 가능

```
1 Player khk;  
2 Player hero;  
3  
4 Player *enemy = new Player();  
5 delete enemy;
```

# Account Example

- 계좌 클래스 예시

```
1 //클래스 정의
2 class Account
3 {
4     std::string name; //예금주 이름
5     double balance; //계좌 잔액
6
7     bool Withdraw(double amount); //출금
8     bool Deposit(double amount); //입금
9 };
10
11 int main()
12 {
13     // 객체의 생성(스택)
14     Account kimAccount;
15     Account leeAccount;
16
17     // 객체의 생성(힙)
18     Account *parkAccount = new Account();
19     delete parkAccount; // 객체 해제(소멸)
20 }
```

# Accessing Class Members

---

- 클래스 멤버의 접근

- 멤버 = 멤버 변수 + 멤버 함수
- 멤버 변수/함수에 접근하기 위해서는 객체가 필요
  - (Static 멤버의 경우는 예외)
- 어떠한 멤버 변수/함수는 "클래스 외부"에서 접근이 불가능하게 만들 수 있음(정보 은닉)

- 값 형식의 객체인 경우 멤버 접근 방법
  - "." 연산자 사용 (멤버 접근 연산자)

```
1 Player player1;  
2  
3 player1.name; //player1이 가진 name 멤버 변수에 접근  
4 player1.Move(2, 3); // player1이 가진 Move() 멤버 함수에 접근
```

*"Player가 가진"이 아니라, "player1이 가진" 이라고 표현한 것에 유의!*

- 객체의 포인터인 경우
  - 역참조 후 점 연산자 사용 (사용 빈도 적음)

```
1 Player *player1 = new Player();  
2  
3 (*player1).name; // kimAccount 포인터가 가리키는 곳에 있는 객체의 name에 접근  
4 (*player1).Move(1,1); // kimAccount 포인터가 가리키는 곳에 있는 객체의 Move()에 접근
```

- 화살표 연산자 (member of pointer 연산자) 사용

➤ 위와 완전히 동일한 의미. 그냥 축약 표현일 뿐

```
1 Player *player1 = new Player();  
2  
3 player1→name;  
4 player1→Move(1,1);
```

# Class Member Access Modifier (visibility)

---

- 정보 은닉을 위해 멤버 접근을 제한할 수 있음
- 클래스 멤버 접근 제한자
  - public
    - 객체에서 접근 가능
  - private
    - 클래스의 멤버들만 접근 가능
  - protected
    - 상속된 클래스의 객체에서만 접근 가능 (*상속 강의에서 설명 예정*)



# Class Member Access Modifier

- 클래스 멤버 접근 제한자

```
1 class Player
2 {
3 public:
4     void Talk(std::string text);
5     bool IsDead();
6 private:
7     std::string name;
8     int health;
9     int xp;
10 };
```

} *Public, 객체를 통해서도 접근 가능*

} *Private, 클래스 내부에서만 접근 가능*

- 클래스 멤버 접근 제한자의 적용

- private 멤버에 접근하기 위해서는 public 멤버 함수가 필요
- 다 public으로 하면 안되나요?
  - 멤버에 직접 접근하는 것이 오류를 초래할 수 있음
  - 예를 들어, 게임에서 플레이어의 체력이 100이 상한선인데, 실수로 1000을 할당해버린다면?

```
1 Player kim;  
2 kim.name = "KHK"; //Compiler ERROR!  
3 kim.health = 1000; //Compiler ERROR!  
4 kim.Talk("Ready to battle!"); //OK  
5  
6 Player *enemy = new Player();  
7 enemy->xp = 100; //Compiler ERROR!  
8 enemy->Talk("I will kill you!"); //OK  
9  
10 delete enemy;
```

# Account Example

- 계좌 클래스 예제

```
1 class Account
2 {
3 public:
4     bool Withdraw(double amount); // 출금
5     bool Deposit(double amount); // 입금
6 private:
7     std::string name; // 예금주 이름
8     double balance; // 계좌 잔액
9 };
```

# Account Example

- 계좌 객체 예제

- Kim의 계좌 잔액(balance)에 문제가 있다면? 입금(deposit) 과정만 살펴보면 된다!
  - 잔액에 직접 접근이 불가능하고, public 입금 멤버함수를 통해서만 접근이 가능하기 때문
- 테스트 및 디버깅이 쉬워짐, 오류의 가능성이 줄어듦, 방어적 프로그래밍

```
1 Account kimAccount;  
2 kimAccount.balance = 1000.00; //Compiler ERROR!  
3 kimAccount.Deposit(1000.00); //OK  
4 kimAccount.name = "Kim's Account"; //Compiler ERROR!  
5  
6 Account *leeAccount = new Account();  
7 leeAccount->balance = 1000.00; //Compiler ERROR!  
8 leeAccount->Withdraw(1000.00); //OK  
9  
10 delete leeAccount;
```

# Implementing Member Methods

---

- 멤버 함수의 구현

- 기존 일반 함수의 구현과 유사
- **멤버 변수에 접근이 가능**하기 때문에 인자로 전달할 데이터가 적어짐
- 클래스 선언 내에 구현 가능
  - Inline 구현
- 클래스 선언 외부에서도 구현 가능
  - `ClassName::MethodName`
- 명세(specification)와 구현의 분리
  - 클래스의 선언은 .h 파일에 작성
  - 클래스의 구현은 .cpp 파일에 작성

# Implementing Member Methods

- 클래스 선언 내에 멤버 함수 구현

```
1 class Account
2 {
3     public:
4         void SetBalance(double bal)
5         {
6             balance = bal;
7         }
8         double GetBalance()
9         {
10             return balance;
11         }
12     private:
13         double balance;
14 };
```

# Implementing Member Methods

- 클래스 선언 외부에 구현

```
1 class Account
2 {
3 public:
4     void SetBalance(double bal);
5     double GetBalance();
6 private:
7     double balance;
8 };
9
10 void Account::SetBalance(double bal)
11 {
12     balance = bal;
13 }
14 double Account::GetBalance()
15 {
16     return balance;
17 }
```

클래스 내에서는 멤버 함수의 선언만 해 두고,

외부에서 따로 본문을 구현하는 방식.

이때 이 함수가 Account 클래스에 속한 멤버 함수하는 것은  
Account:: 을 통해 알려주어야 함

- 명세와 구현의 분리

- 헤더 파일(Account.h)

- Include guard를 통해 전처리기에서 중복적인 헤더 파일의 포함을 방지

```
1 #ifndef _ACCOUNT_H_
2 #define _ACCOUNT_H_
3
4 class Account
5 {
6 public:
7     void SetBalance(double bal);
8     double GetBalance();
9 private:
10    double balance;
11 };
12
13 #endif
```

*Include guard*



- 명세와 구현의 분리

- 헤더 파일(Account.h)

- Include guard를 통해 전처리기에서 중복적인 헤더 파일의 포함을 방지

```
1 #pragma once
2
3 class Account
4 {
5 public:
6     void SetBalance(double bal);
7     double GetBalance();
8 private:
9     double balance;
10 };
```

← Include guard

# Implementing Member Methods

- 명세와 구현의 파일 분리

- 일반적으로 마주치는 대부분의 라이브러리는 클래스 명세(정의)와 구현이 별도의 파일로 분리되어 있음
- 구현 파일(Account.cpp)

```
1 #include "Account.h"
2
3 void Account::SetBalance(double bal)
4 {
5     balance = bal;
6 }
7 double Account::GetBalance()
8 {
9     return balance;
10 }
```

# Implementing Member Methods

- 명세와 구현의 분리
  - 메인 파일(main.cpp)

```
1 #include <iostream>
2 #include "Account.h"
3
4 int main()
5 {
6     Account kimAccount;
7     kimAccount.SetBalance(1000.00);
8     double bal = kimAccount.GetBalance();
9
10    std::cout << bal << std::endl; //1000
11    return 0;
12 }
```

*Account.h에 Account 클래스의  
정의를 있으므로 사용 가능*

*Account.cpp에 Account::SetBalance() 함수의  
본문이 있으므로 링커가 연결 가능*

# Struct vs Class

## ● 구조체와 클래스

- C++에서는 구조체와 클래스 모두 사용 가능
- 문법적으로는, 기본 접근 권한의 차이 외에는 차이점이 존재하지 않음
  - 클래스 : 명시되어 있지 않으면 private이 기본값
  - 구조체 : 명시되어 있지 않으면 public이 기본값

```
1 class Person
2 {
3     std::string name;
4     std::string GetName();
5 };
6
7 Person p;
8 p.name = "Kim"; //ERROR!
9 cout << p.GetName(); //ERROR!
```

클래스

```
1 struct Person
2 {
3     std::string name;
4     std::string GetName();
5 };
6
7 Person p;
8 p.name = "Kim"; //OK!
9 cout << p.GetName(); //OK!
```

구조체

# Struct vs Class

---

- 구조체와 클래스 사용 가이드라인

- 구조체

- Public 접근이 필요한 데이터로 사용
    - 멤버 함수를 구조체 안에 설정하지 않는 것을 권고

- 클래스

- Private 멤버 변수와 멤버 함수
    - 멤버 함수를 통해서 멤버 변수에 접근하도록 get/set 구현

# (Summary) Class and Object

---

- 클래스란? (사용자 정의 자료형, 객체의 틀, 데이터 + 함수)
- 객체란? (클래스를 기반으로 만들어진 객체)
- 클래스의 선언과 객체의 생성
- 접근 제한자 (private, public, protected)
- 멤버 변수와 멤버 함수 (멤버 함수는 멤버 변수에 바로 접근 가능)
- 명세와 구현의 분리 (.h파일에 명세, .cpp 파일에 구현 소스)
- 구조체 vs 클래스 (동일한 문법, 기본 접근 권한이 다르다)

# 생성자와 소멸자

# (Summary) Constructor and Destructor

---

- 생성자
- 소멸자
- 기본 생성자



- 생성자

- 특수한 멤버 함수
- 객체가 생성될 때 자동으로 호출됨
- 초기화 목적으로 유용하게 사용됨
- 클래스와 동일한 이름을 갖는 멤버 함수
- 반환형은 존재하지 않음
- 오버로딩 가능

# Constructors

- 생성자

```
1 class Player
2 {
3 public:
4     Player();
5     Player(std::string name);
6     Player(std::string name, int health, int xp);
7 private:
8     std::string name;
9     int health;
10    int xp;
11 };
```

} 생성자들  
반환형이 없고, 함수 이름이  
클래스 이름과 동일함

# Constructors

- 생성자

```
1 class Account
2 {
3 public:
4     Account();
5     Account(std::string name, double balance);
6     Account(std::string name);
7     Account(double balance);
8 private:
9     std::string name;
10    double balance;
11 };
```

} 생성자들  
반환형이 없고, 함수 이름이  
클래스 이름과 동일함

### ● 소멸자

- 특수한 멤버 함수
- 객체가 소멸할 때 자동으로 호출됨
- 메모리 및 기타 리소스(파일 close 등)해제 목적으로 유용하게 사용됨
- 클래스와 동일한 이름 앞에 "~"을 갖는 멤버 함수
- 반환형 및 파라미터는 존재하지 않음
- 오버로딩 불가능!

# Destructor

- 소멸자

```
1 class Player
2 {
3 public:
4     Player();
5     Player(std::string name);
6     Player(std::string name, int health, int xp);
7     ~Player();
8 private:
9     std::string name;
10    int health;
11    int xp;
12 };
```

# Destructor

- 소멸자

- 객체 또는 객체의 포인터가 소멸되는 시점에 자동으로 호출

```
1 {  
2     Player slayer;  
3     Player kim {"Kim", 100, 4};  
4     Player hero {"Hero"};  
5     Player enemy {"Enemy"};  
6 } //4 destructor called  
7  
8 Player *enemy = new Player{"Enemy2", 1000, 0};  
9 delete enemy; //destructor called
```

# Default Constructor

- 기본 생성자

- 인자가 없는 생성자

- **클래스의 생성자를 직접 구현하지 않으면, 컴파일러가 기본적으로 만들어 줌!**

- 초반부에, 생성자가 없는 상태에서도 객체를 만들 수 있었음

- 컴파일러가 기본 생성자를 알아서 만들어서 사용했기 때문

- 객체를 인자 없이 생성하면 호출됨

```
1 class Player
2 {
3 };
```

이렇게만 코드를 작성해도,

```
1 class Player
2 {
3     Player()
4     {
5     }
6 };
```

컴파일하면 자동으로 생성자가 하나 생김!

(진짜로 눈에 보이게 코드가 생성되는 것이 아니라,  
내부적으로 생성됨)

```
1 Player kim;
2 Player *enemy = new Player;
```

따라서 위와 같은 명령문을 생성자 만들지 않고도  
사용 가능했음

# Declaring a Class

- 인자가 없는 클래스 생성자도 구현 해 주는 것이 좋음
  - 쓰레기 값은 항상 방지하는 것이 안전

```
1 class Account
2 {
3 public:
4     Account()
5     {
6         name = "None";
7         balance = 0.0;
8     }
9     bool Withdraw(double amount);
10    bool Deposit(double amount);
11 private:
12    std::string name;
13    double balance;
14 };
```




# Declaring a Class

- 인자가 있는 생성자만 구현한 경우
  - 기본 생성자가 자동 생성되지 않음

```
1 class Account
2 {
3 public:
4     Account(std::string nameVal, double bal)
5     {
6         name = nameVal;
7         balance = bal;
8     }
9     bool Withdraw(double amount);
10    bool Deposit(double amount);
11 private:
12    std::string name;
13    double balance;
14 };
```

인자가 있는 생성자 "만" 정의되어 있으므로,  
인자가 없이 객체를 만들 수 없음  
기본 생성자(인자가 없는 생성자)가 자동  
생성되지 않기 때문



```
1 Account kimAccount; //ERROR
2 Account leeAccount = new Account; //ERROR
3 delete leeAccount;
4
5 Account parkAccount {"Park", 1000.0}; //OK
```

# (Summary) Constructor and Destructor

---

- 생성자

- 클래스 이름과 동일한 이름을 갖는 특별한 함수
- 객체가 생성될 때 자동 호출되며 반환 값 없음. 오버로딩 가능

- 소멸자

- 클래스 이름과 동일한 이름 앞이 ~가 붙은 특별한 함수
- 객체가 소멸될 때 자동 호출되며 반환 값 없음. 오버로딩 불가능

- 기본 생성자

- 명시하지 않으면 자동으로 만들어주지만, 쓰레기 값은 없어야 하므로 구현 필요
- 인자가 있는 생성자를 만들면 기본 생성자는 자동 생성되지 않는다

# 생성자 오버로딩

# (Summary) Constructor Overloading

---

- 생성자 오버로딩 개요
- 생성자 초기화 리스트
- 생성자 위임
- 생성자 기본 매개변수

# Overloading Constructor

- 생성자 오버로딩
  - 생성자는 오버로딩 가능 (생성자도 함수이므로)
  - 각각의 생성자는 고유해야 함(매개변수가 달라야 함)
    - 함수 오버로딩과 동일

```
1 class Player
2 {
3 public:
4     Player();
5     Player(std::string nameVal);
6     Player(std::string nameVal, int healthVal, int xpVal);
7 private:
8     std::string name;
9     int health;
10    int xp;
11 };
```

# Overloading Constructor

- 생성자 오버로딩

```
1 Player::Player()
2 {
3     name = "None";
4     health = 0;
5     xp = 0;
6 }
7
8 Player::Player(std::string nameVal)
9 {
10     name = nameVal;
11     health = 0;
12     xp = 0;
13 }
14
15 Player::Player(std::string nameVal, int healthVal, int xpVal)
16 {
17     name = nameVal;
18     health = healthVal;
19     xp = xpVal;
20 }
```

# Overloading Constructor

- 오버로딩된 생성자의 활용


```
1 Player empty; //None, 0, 0
2
3 Player hero {"Hero"}; //Hero, 0, 0
4
5 Player kim {"Kim", 100, 5}; //Kim, 100, 5
6
7 Player *player1 = new Player; //None, 0, 0
8 delete player1;
9
10 Player *player2 = new Player{"Enemy2"}; //Enemy2, 0, 0
11 delete player2;
12
13 Player *player3 = new Player{"Enemy3", 1000, 0}; //Enemy3, 1000, 0
14 delete player3;
```

### ● 생성자 초기화 리스트

- 진짜(?) 초기화
- 이전의 생성자는 생성자 본체(body) 내에서 멤버 변수에 값을 대입
- 생성자 초기화 리스트를 사용할 경우 생성과 동시에 값이 지정됨

이 시점에서 생각해보면, 멤버 변수에  
쓰레기 값이 들어있는 상태

```
1 Player::Player()  
2 {  
3     name = "None";  
4     health = 0;  
5     xp = 0;  
6 }
```



기존 방법

```
1 Player::Player()  
2     : name{"None"}, health{0}, xp{0}  
3 {  
4 }
```

생성자 멤버 초기화 리스트 (권장 방법)



# Delegating Constructors

---

- 생성자 위임

- 다양한 생성자의 오버로딩에 유사한 코드가 반복적으로 사용
- 오류의 가능성이 높아짐
- 생성자 위임을 통해 오류 가능성과 코드 반복을 줄일 수 있음
  - 다른 생성자를 멤버 초기화 리스트 위치에서 호출
  - 생성자 멤버 초기화 리스트를 사용해서만 가능!

# Delegating Constructors

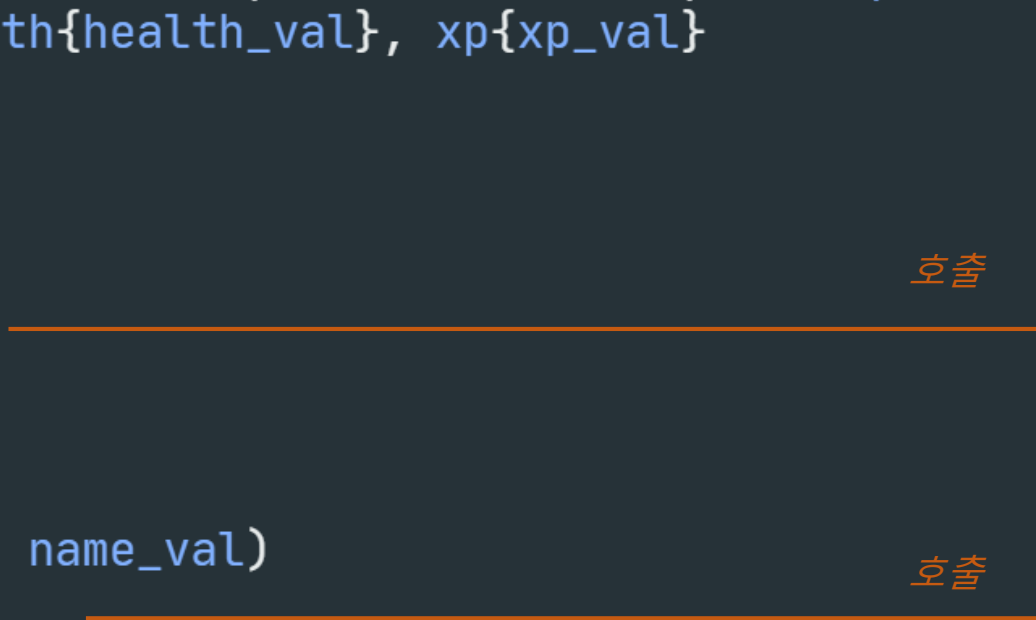
- 생성자 위임을 사용하지 않는 기존 코드

```
1 Player::Player(std::string nameVal, int healthVal, int xpVal)
2     : name{nameVal}, health{healthVal}, xp{xpVal}
3 {
4 }
5
6 Player::Player()
7     : name{"None"}, health{0}, xp{0}
8 {
9 }
10
11 Player::Player(std::string nameVal)
12     : name{nameVal}, health{0}, xp{0}
13 {
14 }
```

# Delegating Constructors

- 생성자 위임을 사용한 코드

```
1 Player::Player(std::string name_val, int health_val, int xp_val)
2     : name{name_val}, health{health_val}, xp{xp_val}
3 {
4 }
5
6 Player::Player()
7     : Player{"None", 0, 0}
8 {
9 }
10
11 Player::Player(std::string name_val)
12     : Player{name_val, 0, 0}
13 {
14 }
```



# Default Constructor Parameters

- 생성자 기본 매개변수
  - 생성자 또한 함수이므로, 기본 매개변수 사용 가능

```
1 class Player
2 {
3 public:
4     Player(std::string nameVal="None", int healthVal = 0, int xpVal = 0);
5 private:
6     std::string name;
7     int health;
8     int xp;
9 };
10
11 Player::Player(std::string nameVal, int healthVal, int xpVal)
12     : name{nameVal}, health{healthVal}, xp{xpVal}
13 {
14 }
15
```

# Default Constructor Parameters

- 생성자 기본 매개변수
  - 생성자 또한 함수이므로, 기본 매개변수 사용 가능

```
1 Player empty;  
2 Player kim{"Kim"}; //Kim, 0, 0  
3 Player hero {"Hero",100}; //Hero, 100, 0  
4 Player enemy{"Enemy",1000,0}; //Enemy, 1000, 0
```

# (Summary) Constructor Overloading

---

- 생성자 오버로딩 개요
  - 매개변수가 다른 고유한 생성자들을 오버로딩
- 생성자 초기화 리스트
  - 본체에서 값을 대입하는 것이 아닌 생성과 동시에 값의 지정
- 생성자 위임
  - 다른 생성자를 초기화 리스트 자리에서 호출
  - 생성자 초기화 리스트를 사용해서만 가능
- 생성자 기본 매개변수
  - 함수처럼 기본 매개변수 사용 가능

- 아래 각각의 경우의 차이점은 무엇이고, 언제 어떤 방법을 사용해야 할까?

```
class A
{
public:
    A()
        :width(500)
        ,height(300){};
    int width;
    int height;
};

class B
{
public:
    B()
    {
        width = 500;
        height = 300;
    };
    int width;
    int height;
};

class C
{
public:
    int width = 500;
    int height = 300;
};
```