#### **절차지향 프로그래밍의 문제점**

절차지향 프로그래밍에서는 데이터를 함수에 전달하여 처리를 합니다. 이는 프로그램이 커지면 코드의 재사용성이 낮아지고, 유지보수가 어려워지는 문제점이 있습니다.

#### **객체지향 프로그래밍의 도입**

객체지향 프로그래밍에서는 데이터와 해당 데이터를 처리하는 함수를 하나의 객체로 묶습니다. 이를 통해 데이터와 함수가 밀접하게 연관되어 코드의 이해와 유지보수가 쉬워집니다. Animal 객체를 예로 들어보겠습니다:

#include <iostream>  
  
class Animal {  
 private:  
 int food;  
 int weight;  
  
 public:  
 void set\_animal(int \_food, int \_weight) {  
 food = \_food;  
 weight = \_weight;  
 }  
 void increase\_food(int inc) {  
 food += inc;  
 weight += (inc / 3);  
 }  
 void view\_stat() {  
 std::cout << "이 동물의 food : " << food << std::endl;  
 std::cout << "이 동물의 weight : " << weight << std::endl;  
 }  
};

### **객체지향 프로그래밍의 장점**

1. **캡슐화(Encapsulation)**:
2. **재사용성(Reusability)**:
3. **상속(Inheritance)**:
4. **다형성(Polymorphism)**:

### **클래스의 구조**

클래스는 다음과 같은 구조를 가집니다:

cpp

class ClassName {  
 private:  
 // 멤버 변수  
 public:  
 // 멤버 함수  
};

* **멤버 변수**: 클래스 내부에 선언된 변수들로, 객체의 상태를 나타냅니다.
* **멤버 함수**: 클래스 내부에 선언된 함수들로, 객체의 동작을 정의합니다.
* **접근 지정자(Access Specifiers)**: private, public 키워드를 사용하여 멤버의 접근 권한을 설정합니다.

**C++에서는 함수 오버로딩(function overloading)**

같은 이름을 가진 여러 함수를 정의할 수 있습니다. 이 함수들은 매개변수의 타입이나 개수가 다르면 서로 다른 함수로 인식됩니다.

**함수 선택 과정**

함수를 호출할 때 C++은 가장 적합한 함수를 선택합니다. 이 과정은 다음과 같은 단계로 이루어집니다:

**정확히 일치하는 함수 찾기**

* + 전달된 인자의 타입과 정확히 일치하는 함수가 있는지 찾습니다.

**표준 형변환(Standard Conversion)**

* + 정수 타입 간의 변환: char, unsigned char, short는 int로 변환됩니다.
  + unsigned short는 int 또는 unsigned int로 변환됩니다 (크기에 따라 다름).
  + float는 double로 변환됩니다.
  + 열거형(enum)은 int로 변환됩니다.

**포괄적 형변환(Promotions)**

* + 숫자는 다른 숫자 타입으로 변환될 수 있습니다 (예: float -> int).
  + 열거형(enum)은 double로 변환될 수 있습니다.
  + 0은 포인터나 숫자로 변환될 수 있습니다.
  + 포인터는 void 포인터로 변환될 수 있습니다.

**사용자 정의 타입 변환(User-defined Conversions)**

* + 클래스 타입의 객체는 사용자 정의 타입 변환을 통해 변환될 수 있습니다 (변환 생성자 또는 변환 연산자).

### **모호성(Ambiguity)**

함수 선택 과정에서 동일한 우선순위를 가지는 두 개 이상의 함수가 일치하면 컴파일러는 오류를 발생시킵니다. 이를 "모호한 호출(Ambiguous call)"이라고 합니다.

### **생성자 (Constructor)**

객체를 생성 후 초기화를 도와주는 장치입니다.

오버로딩이 가능하다.

### **디폴트 생성자 (Default constructor)**

생성자를 정의하지 않았을 때 호출되는 생성자. 아무 인자도 받지 않습니다.

### **소멸자 (Destructor)**

객체가 생성될 때 호출되는 생성자처럼 소멸할 때 호출되는 함수이다.

멤버들의 메모리를 해제시켜 누수를 막는용으로 사용하는 경우가 많습니다.

**복사 생성자 (copy constructor)**

다른 객체의 내용을 복사해서 새로운 객체를 생성한다.

T(const T& a);

Photon\_Cannon pc2(pc1);

Photon\_Cannon pc3 = pc2;

**디폴트 복사 생성자(Default copy constructor)**

얕은 복사만 수행해준다는 한계가 있습니다.

**깊은 복사(deep copy)**

메모리를 새로 할당해서 내용을 복사

**얕은 복사(shallow copy)**

단순히 대입

### **생성자의 초기화 리스트(initializer list)**

Marine::Marine() : hp(50), coord\_x(0), coord\_y(0), damage(5), is\_dead(false) {}  
  
Marine::Marine(int x, int y)  
 : coord\_x(x), coord\_y(y), hp(50), damage(5), is\_dead(false) {}

**초기화 리스트를 사용하는 이유**

생성과 초기화를 동시에 함.

사용하지 않는 경우 생성 후 대입을 수행

전자가 더 효율적 +

상수나 레퍼런스들은 생성과 동시에 초기화되어야함. 따라서 초기화 리스트를 사용해 초기화 해야함

**static 함수**

객체를 생성하지 않아도 사용가능

**레퍼런스를 리턴하는 멤버함수**

this는 객체 자신을 가리키는 포인터

return this\*은 객체 자신을 리턴하는 것

만약 리턴타입이 레퍼런스 타입이 아닌경우

임시객체를 생성해 복사된 것이 반환되는 반면

레퍼런스 타입인경우 그 자체가 반환됨

m1.be\_attacked(10).be\_attacked(20); 이 가능

### **const 함수**

int attack() const;

변수들의 값을 바꾸지 않고 읽기 만 하는, 마치 상수 같은 멤버 함수를 상수 함수 로써 선언

**암시적 변환**

객체를 인자로 받는 함수가 있을 때

다른 타입의 객체를 인자로 입력하면

입력한 인자로 받았어야할 객체로 변환 가능한지 확인하고

변환 가능하면 변환해서 넣어준다.

### **explicit**

원하지 않는 암시적 변환을 할 수 없도록 컴파일러에게 명시

DoSomethingWithString("abc")

DoSomethingWithString(MyString("abc"))

다음처럼 암시적변환이 일어날 수 있음.

### 그러나 **explicit** 선언시 제한가능.

explicit MyString(char\* str);

### **Mutable**

mutable 멤버변수는 const 멤버함수에서도 변경이 가능하다.

mutable이 필요한 이유>

const는 객체 내부를 변경하지않는다는 의미이지만

실제로는 읽기를 위한 함수여도 내부에서 변경이 일어나고 사용자관점에서 명시하기위해 const를 붙여야할 때도 있다

이럴 때 예외적으로 변경할수있게 해주는 것이 mutable이다.