C++

> 공식자료의 목차에서 모르는 개념만 정리하였다.

▪ Language

▫ Class

C++에서 클래스(class)는 객체 지향 프로그래밍의 핵심 요소로, 사용자 정의 타입을 만들 때 사용된다. 구조체(`struct`)와 비슷하지만, 접근 제어와 상속 등 추가적인 기능이 있다.

🔷 클래스의 기본 구조

클래스는 데이터 멤버(변수)와 멤버 함수(메서드)로 구성된다. 기본적인 선언 방식은 다음과 같다:

```
class 클래스이름 {
public:
    // 생성자
    클래스이름();
```

```
    // 멤버 함수
    반환타입 함수이름();

private:
    // 데이터 멤버
    타입 변수이름;
};
```

- `public`: 외부에서 접근 가능한 멤버들을 선언.
- `private`: 클래스 내부에서만 접근 가능한 멤버들을 선언.

🔷 생성자와 소멸자

- **생성자(Constructor)**: 객체가 생성될 때 자동으로 호출되는 함수로, 멤버 변수를 초기화하는 데 사용된다.

```
클래스이름::클래스이름() {
    // 초기화 코드
}
```

- **소멸자(Destructor)**: 객체가 소멸될 때 자동으로 호출되는 함수로, 동적 할당된 메모리를 해제하는 데 사용된다.

```
클래스이름::~클래스이름() {
    // 정리 코드
}
```

🔷 멤버 함수와 데이터 멤버

- **멤버 함수**: 클래스 내에 정의된 함수로, 객체의 동작을 정의한다.

```
cpp
복사편집
반환타입 클래스이름::함수이름() {
    // 함수 구현
}
```

- **데이터 멤버**: 클래스 내에 정의된 변수로, 객체의 상태나 속성을 저장한다.

🔷 접근 제어 지시자

- `public`: 모든 곳에서 접근 가능.
- `private`: 해당 클래스의 멤버 함수와 `friend`로 선언된 함수에서만 접근 가능.
- `protected`: 해당 클래스와 이를 상속받은 클래스에서도 접근 가능.

🔷 상속(Inheritance)

C++에서는 한 클래스가 다른 클래스를 상속받아 기존의 기능을 재사용하거나 확장할 수 있다.

```
class 부모클래스 {
    // ...
};

class 자식클래스 : public 부모클래스 {
    // ...
};
```

- `public` **상속**: 부모 클래스의 `public` 멤버는 자식 클래스에서 `public`으로, `protected` 멤버는 `protected`로 상속된다.
- `protected` **상속**: 부모 클래스의 `public`과 `protected` 멤버가 모두 자식 클래스에서 `protected`로 상속된다.
- `private` **상속**: 부모 클래스의 `public`과 `protected` 멤버가 모두 자식 클래스에서 `private`으로 상속된다.

🔷 가상 함수와 추상 클래스

- **Virtual Function**: 부모 클래스에서 선언하고 자식 클래스에서 재정의할 수 있는 함수로, 다형성을 구현하는 데 사용된다.

  ```
  class 부모클래스 {
  public:
      virtual void 함수이름();
  };
  ```

- **추상 클래스(Abstract Class)**: 하나 이상의 virtual 함수를 포함하는 클래스. 직접 객체를 생성할 수 없으며, 파생 클래스에서 해당 virtual 함수를 구현해야 한다.

```
class 추상클래스 {
public:
    virtual void 함수이름() = 0; // 순수 가상 함수
};
```

🔷 중첩 클래스(Nested Class)

클래스 내부에 다른 클래스를 정의할 수 있으며, 이를 중첩 클래스라고 한다. 중첩 클래스는 외부 클래스의 멤버에 접근할 수 있다.

```
cpp
복사편집
class 외부클래스 {
public:
    class 내부클래스 {
        // ...
    };
};
```

> 자프실에서 간단한 프로그램 만들면서 써봐서 개념의 이해는 어렵지 않았다. cpp에서 어떻게 사용하는지 자세히 정리해서 나중에 보기 편할 듯.

▫ Constant Expressions

컴파일 타임에 값이 결정되는 expression. 즉, 컴파일러가 실행 전에 값을 확정지을 수 있어야 함.

사용하는 이유

- `std::array` 같은 템플릿 기반 구조는 **크기 같은 값이 컴파일 타임에 정해져야 한다.**
- `switch-case`의 case 값, 열거형 값 등도 마찬가지

```
int n = 5;
std::array<int, n> arr; //오류: n은 런타임 값

constexpr int cn = 5;
std::array<int, cn> arr2; //OK: cn은 컴파일 타임 값
```

🔷 const vs constexpr

- `const`는 변하지 않는 값이지만, 꼭 컴파일 타임에 정해질 필요는 없다,
- `constexpr`은 **무조건 컴파일 타임**에 값이 정해져야 한다.

```
const int a = some_runtime_function(); // 가능
constexpr int b = 5 + 3; // 컴파일 타임에 계산됨
```

→ 상수 표현식이 필요한 곳엔 `constexpr`을 써야 안전하다.

🔷 constexpr 함수

C++11부터 함수도 `constexpr` 가능. 리턴값이 상수 표현식으로 쓰이게 만들 수 있다.

```
constexpr int square(int x) {
    return x * x;
}

constexpr int result = square(4);
```

조건은:

- 리턴 타입과 모든 인자가 상수 표현식 가능해야한다
- 함수 내용이 조건문, 루프 없이 계산만 있어야한다

사용하면 좋은 경우

- 배열 크기
- 열거형 초기화
- 템플릿 인자
- switch-case 값

- 기타 컴파일 타임 계산 필요한 곳

> 결론 : 조건만 잘 지키면 유용할 듯. 하지만 실제로 const를 사용하는 것과 유의미한 차이를 느낄 수 있을지는 모르겠다.

▪ Library

▫ <cmath>

기능	함수
절댓값	`std::abs`
제곱/제곱근	`std::pow` `std::sqrt`
삼각함수	`std::sin` `std::cos` 등
로그/지수	`std::log` `std::exp`
반올림	`std::floor` `std::round`
나머지	`std::fmod`
거리	`std::hypot` (root(x^2 + y^2))

> 자주 쓰는 함수를 간단히 정리해봤다. 거리함수 유용할 듯!

> 앞으로도 문서 자주 보며 시간날때마다 정리해야겠다.