


# 람다 드라이버란? (풀 메탈 패닉! 시리즈)

C++에서 람다 표현식의 필요성은 그 효율성과 유연성에서 기인합니다. 람다 표현식은 특히 STL(Standard Template Library)에서 알고리즘을 작성할 때 편리합니다. 정렬, 검색, 변환 등의 작업을 수행하는 함수를 람다 표현식을 사용해 간단하게 작성할 수 있기 때문입니다.

```
// [캡처리스트](파라미터 리스트)->반환타입 {함수 본문};
```

```
auto add = [](int a, int b) {return a + b; };
```

```
auto add = [](int a, int b) {return a + b; };
```

 (지역 변수) `class` `lambda` `[](int a, int b)->int`

✧✧ Copilot으로 설명

re 크기: 1바이트

맞춤: 1바이트

클래스 라고?

## ✓ C++에서 람다는 "익명 클래스"이다

컴파일러는 람다식을 만나면 익명 클래스 타입을 정의하고, 객체를 생성합니다.

cpp

📄 복사 ✎ 편집

```
auto add = [](int a, int b) { return a + b; };
```

이 코드는 내부적으로 아래와 거의 동일한 형태로 컴파일됩니다 (간략화 예):

cpp

📄 복사 ✎ 편집

```
struct __Lambda_add {  
    int operator()(int a, int b) const {  
        return a + b;  
    }  
};
```

```
__Lambda_add add; // 람다 객체 생성
```

즉, 람다는 클래스 타입이며, `operator()` 를 오버로딩한 함수 객체입니다.

좋은 질문입니다.

C++에서 말하는 **익명 클래스 (anonymous class)**는 문법적으로 사용자가 이름을 붙이지 않은 클래스입니다.

람다는 그 대표적인 예입니다.

## ✅ 익명 클래스란?

**정의:** 클래스 타입이지만 사용자가 직접 이름을 붙이지 않은 클래스

C++에서 람다식을 사용하면, 컴파일러가 자동으로 이름 없는 클래스 타입을 생성합니다.

cpp

📄 복사 ✎ 편집

```
auto f = [](int x) { return x + 1; };
```

위 코드는 내부적으로 컴파일러가 대략 다음과 같은 클래스를 생성합니다.

cpp

📄 복사 ✎ 편집

```
// 이름이 없음 (컴파일러가 내부적으로만 관리)
class /*익명*/ {
public:
    int operator()(int x) const {
        return x + 1;
    }
} f;
```

- 이 클래스에는 이름이 없기 때문에, 코드 상에서는 `auto` 나 `decltype()` 을 통해서만 참조 가능합니다.
- 타입은 존재하지만 이름이 없으므로 명시적 선언/사용이 불가능합니다.

```

int main(void) {

    // 람다는 익명클래스 이다.

    int add(int a, int b)
    {
        return a + b;
    }

    auto add = [](int a, int b) {return a + b; };

    auto c1 = add(1, 2);
    auto c2 = add(1, 2);

    return 0;
}

```

```

참조 0개
public class Example
{
    참조 0개
    public static void Main()
    {
        int Add(int a, int b) { return a + b; }
        int c1 = Add(1, 2);
        int c2 = Add(2, 3);

        System.Func<int, int, int> cb = (a, b)
        cb(1, 2);
        cb(2, 3);
    }
}

```

문제 원인: C++에서 함수 선언은 함수 내부에 불가능

- C++ 표준 문법상 함수 안에 또 다른 함수를 선언하는 것은 허용되지 않습니다.
- 즉, 함수 내부에 함수 정의(중첩 함수)는 불가능합니다.

### 3. 로컬 클래스 + 객체 생성 (함수 내부에 클래스 정의 가능)

cpp

 복사  편집

```
int main() {  
    class LocalAdd {  
    public:  
        int operator()(int a, int b) const { return a + b; }  
    };  
  
    LocalAdd add;  
    int result = add(1, 2);  
}
```

## 요약

### 이유

함수 내부에 함수 선언 불가

람다 또는 로컬 클래스 사용

### 설명

C++ 표준에서는 허용하지 않는 문법

함수 내부 함수처럼 쓸 수 있는 대안

// 람다는 익명클래스 이다.

```
auto add = [](int a, int b) {return a + b; };
```

```
auto c1 = add(1, 2);
```

```
auto c2 = add(1, 2);
```

```
struct inner_class
```

```
{
```

```
    int add(int a, int b)
```

```
    {
```

```
        return a + b;
```

```
    }
```

```
};
```

```
inner_class obj;
```

```
auto c3 = obj.add(2, 3);
```

## 1. 람다는 클래스다

- C++에서 람다는 내부적으로 `operator()` 를 가진 클래스(함수 객체) 타입입니다.

## 2. 람다는 익명 클래스다


- 람다는 이름이 없는(익명) 클래스 타입으로, 컴파일러가 자동 생성합니다.

## 3. C++에서는 로컬 클래스 선언을 허용한다

- 함수 내부에 클래스를 선언하는 것을 '로컬 클래스'라 하며, C++ 표준에서 허용됩니다.

요약하면,

cpp

 복사  편집

```
int main() {  
    class LocalClass { /* ... */ }; // 함수 내부 클래스 허용  
  
    auto lambda = [](int x) { return x + 1; }; // 익명 클래스 객체 생성  
}
```

위 모두 표준적인 C++ 문법입니다.



// [] 는 람다표현식이 자신을 둘러싸는 코드의 범위에서 변수를 캡처할 수 있게 해주는 도구이다.  
// -> 을 이용해서 반환 타입을 지정할 수 있다

```
auto add2 = [](int a, int b) -> float {return a + b; };
```

 (지역 변수) float add2(int a, int b)

[] 는 람다표현식이 자신을 둘러싸는 코드의 범위에서 변수를 캡처할 수 있게 해주는 도구이다.

-> 을 이용해서 반환 타입을 지정할 수 있다

💡 Copilot으로 설명

```
return 0;
```

```
auto add3 = [](float a, double b) { return a + b; };
```

```
auto add4 = [](int a, double b) { return a + b; };
```

// 에러남, string에는 int와 + 가가능한 오퍼레이터가 없으니깐.

```
// auto add10 = [](string a, double b) { return a + b; };
```

## ✓ 캡처 구문 기본 구조

cpp

📄 복사

✎ 편집

```
[capture_list](parameters) { body }
```

- `[capture_list]`: 어떤 외부 변수를 캡처할지 지정
- `parameters`: 일반 함수처럼 인자 목록
- `{ body }`: 실행 내용

## ✓ 캡처 방식 종류 요약

문법	의미	외부 변수 수정 가능	설명
<code>[=]</code>	외부 변수 값 복사 캡처	✗ 불가능 (읽기 전용)	전체 복사
<code>[&amp;]</code>	외부 변수 레퍼런스(참조) 캡처	✓ 가능	전체 참조
<code>[a]</code>	변수 <code>a</code> 만 값 복사	✗	명시적 복사
<code>[&amp;a]</code>	변수 <code>a</code> 만 참조 캡처	✓	명시적 참조
<code>[=, &amp;b]</code>	기본은 값 복사, <code>b</code> 만 참조	✓ (단 <code>b</code> 만)	혼합
<code>[&amp;, a]</code>	기본은 참조, <code>a</code> 만 복사	✗ ( <code>a</code> 는 불변)	혼합
<code>[this]</code>	멤버 함수 내에서 <code>this</code> 포인터 복사	클래스 멤버 접근 가능	객체 람다

람다 표현식의 구조는 다음과 같습니다:

```
[capture](parameters)->return-type{body}.
```

첫 번째 요소인 `capture`는 람다 표현식이 상위 범위의 변수를 어떻게 접근하는지를 결정합니다. 여기에는 여러 가지 옵션이 있습니다:

- `[]`: 캡처하지 않음. 즉, 람다 함수 내에서 상위 범위의 변수를 사용할 수 없습니다.
- `[=]`: 모든 변수를 값으로 캡처합니다. 즉, 상위 범위의 변수의 복사본을 만들어 사용합니다.
- `[&]`: 모든 변수를 참조로 캡처합니다. 이 경우 람다 함수 내에서 상위 범위의 변수를 수정할 수 있습니다.
- `[x, &y]`: `x`는 값으로, `y`는 참조로 캡처합니다.

✓  
// 이걸 안됨.

// auto add6 = [c1, c2]() { c1 = 10; c2 = 20; };

auto add7 = [&c1, &c2]() { c1 = 10; c2 = 20; };  
add7();

- 이걸 됨, &로 이값은 수정해도 됩니다. 라고 해줬으니깐

// 이걸 안됨.

// = c1을 const로 가져온다는 얘기라서,

auto add8 = [=, &c2]() { c1 = 10; c2 = 20; };

```
// &로 하겠다는 얘기는, 나는 여기에 선언된, 변수를 참조로 가져가서,  
// 수정할거이라는 내용임  
auto add9 = [&]() {  
    c1 = 100;  
    c2 = 200;  
    return c1 + c2; };  
  
auto add9value = add9();
```

```
// 이건 안됨  
// =로 하겠다는 얘기는, 외부 변수를 값 복사하는 것이고,  
// 복사된 변수는 const 취급을 당함  
auto add10 = [=]() {  
    c1 = 100;  
    c2 = 200;  
    return c1 + c2; };
```

```
// 이거는 됨  
// =로 가져간거니깐, 이집에 있는 변수를 이쁘게 읽을게요. 같은 얘기  
auto add11 = [=]() { return c1 + c2; };  
auto add11value = add11();
```

## ✓ 올바른 정리

`&`는 참조 캡처입니다.

- 외부 변수를 복사하지 않고, 원본을 직접 참조합니다.
- 따라서 읽기/쓰기 모두 가능합니다.
- `const`도 아닙니다 (사용자가 `const`로 선언한 게 아니라면).

반면, `=`는 값 복사 캡처입니다.

- 외부 변수의 복사본을 생성하여 람다 내부로 가져옵니다.
- 그리고 람다의 `operator()`는 기본적으로 `const` 멤버 함수입니다.
- 그로 인해, 복사한 값도 람다 내부에서 `const`처럼 읽기 전용이 됨 → 수정 불가

```
// 혼합도 됨  
// 기본적으로 = 로 처리하는데,  
// add11value는 값을 수정좀 하겠수다.  
auto add12 = [=, &add11value]() {  
    add11value = 300;  
    return c1 + add11value; };  
auto add12value = add12();
```

// 이걸 안됨

// 캡처리스트가 비어있음

// 람다 내부에서 외부 변수를 안 보겠다고 했음

// 그래서 컴파일러가 c1도, add11value 도 모름

```
auto add13 = []() {  
    add11value = 300;  
    return c1 + add11value; };
```



```
// 람다를 파라미터로 받아 실행하고,  
// 그 결과를 조합하는 함수형 스타일의 코드  
auto add_combine= [](auto someFunc) {  
    int sum = someFunc(1, 2) + someFunc(2, 3);  
    return sum;  
};
```

```
// 이런식으로 람다함수를 매개변수로 전달해서 사용도 가능함  
auto combineValue = add_combine(add);
```

// 람다를 파라미터로 받아 실행하고,  
// 그 결과를 조합하는 함수형 스타일의 코드

```
auto add_combine= [](auto someFunc) {  
    int sum = someFunc(1, 2) + someFunc(4, 5);  
    return sum;  
};
```

// 이런식으로 람다함수를 매개변수로 전달해서 사용도 가능함  
auto combineValue = add\_combine(add);

// 좀더 복잡하게 짜면 이런것도 가능함  
// '동적 함수' 조합 구현  
auto mul = [](int a, int b) {  
 return a \* b; };  
auto combineValue2 = add\_combine(mul);

```
auto add14 = [c1, add11value]()  
{  
    c1 = 10;  
    return c1 + add11value;  
};
```

```
// mutable 먹이면 또 가능함.  
// 그렇다고 c1 값이 또 10으로 된건 아님.
```

```
auto add15 = [c1, add11value]()
```

```
    mutable
```

```
    {
```

```
        c1 = 10;
```

```
        return c1 + add11value;
```

```
    };
```

```
auto add15value = add15();
```

```
// 팩토리얼  
// #include <functional>  
std::function<int(int)> factorial = [&factorial](int i)  
{  
    return i > 1 ? i * factorial(i - 1) : 1;  
};
```

```
// 이것이 궁극의 람다인듯  
auto factorialValue = factorial(5);
```

```

for (auto k : order)
{
    auto v_list = map[k.first];

    std::sort(v_list.begin(), v_list.end(),
        []( std::pair<int, int> a, std::pair<int, int> b) {
            if (a.second == b.second)
                return a.first < b.first;
            return a.second > b.second;
        });

    int min_v = 2;
    min_v = min(min_v, (int)v_list.size());

    for (int i = 0; i < min_v; i++)
    {
        answer.push_back(v_list[i].first);
    }
}

```

코딩 테스트 기준으로  
요정도인데...

[System.Serializable]

참조 11개

✓ public class UserEquip : IKeyParsable

{

public int equip\_index; // 고유키(서버관리용)

public int id; // Equip ID, 장비 마스터 데이터 인덱스,

public int target\_id; // 해당 아이템을 장착한 영웅/함선 ID, "" 이면 장착하지 않은 상태

public int level = 1; // 장비 레벨

public int upgrade = 0; // 강화 단계

public int fopt00; // 고정 옵션1

public int fopt01; // 고정 옵션2

public int vopt00; // 가변 옵션1

public int vopt01; // 가변 옵션2

public int vopt02; // 가변 옵션3

public int lock\_state = 0; // 0:unlock, 1:lock

public int reroll\_opt = -1; // 재굴림 옵션 vopt00, vopt01

public int reroll\_count; // 재굴림 횟수

public EQUIP\_TYPE equip\_type;

public int data\_grade;

public int max\_reroll\_count;

# 유저 장비 데이터

```
var equip_list = UserDataManager.Instance.GetUserEquipPartList(popupArgs.reward_type);  
var data_list = equip_list.Where(v => v.CompareEquipType(find_type) && v.CompareDataGrade(find_grade))  
    .OrderBy(v => v.IsLock() || v.IsEquiped())  
    .ThenBy(v => v.level)  
    .ThenBy(v => v.id)  
    .Select(v => v.equip_index)  
    .ToList();  
listView.InitListView(popupArgs.reward_type, data_list);  
}
```

실무 데이터 레벨까지 가면  
정렬된 데이터를 가져오려고 요기까지 갈거라.  
**람다는 중요함**



**FULL METAL  
PANIC!**

HG 1-50 L—JCP-43 Ver. IV  
LAEWATEN Ver. IV

# END