



# 인트아이 C++ 심화 스터디

---

THREAD, MUTEX, SEMAPHORE

# 3회차 퀴즈 답안

```
struct OddFirst {
    bool operator()(int a, int b) const {
        if (a % 2 == 0) {
            if (b % 2 == 0) {
                return a < b; // a짝 b짝
            }
            return true; // a짝 b홀
        }
        if (b % 2 == 0) {
            return false; // a홀 b짝
        }
        return a > b; // a홀 b홀
    }
};

std::priority_queue<int, std::vector<int>, OddFirst> pq(A, A + 10);
```

## C++ named requirements: *Compare*

**Compare** is a set of requirements expected by some of the standard library facilities from the user-provided function object types.

The return value of the function call operation applied to an object of a type satisfying *Compare*, when *contextually converted* to `bool`, yields `true` if the first argument of the call appears before the second in the *strict weak ordering relation* induced by this type, and `false` otherwise.

As with any *BinaryPredicate*, evaluation of that expression is not allowed to call non-const functions through the dereferenced iterators and, syntactically, the function call operation must accept `const` object arguments, with the same behavior regardless of whether the arguments are `const` or non-`const` (since C++20).

### Requirements

The type `T` satisfies *Compare* if

- The type `T` satisfies *BinaryPredicate*, and

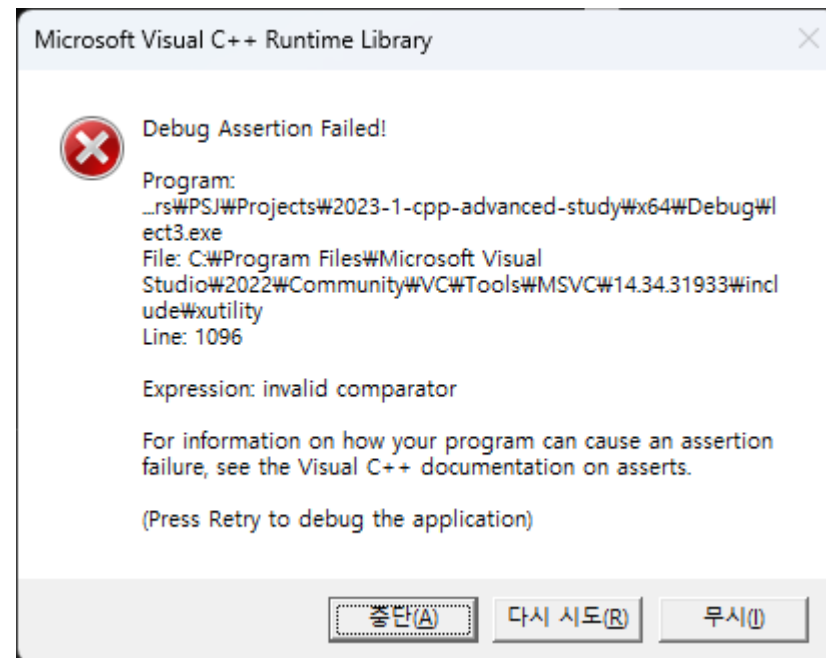
Given

- `comp`, an object of type `T`
- `equiv(a, b)`, an expression equivalent to `!comp(a, b) && !comp(b, a)`

The following expressions must be valid and have their specified effects

Expression	Return type	Requirements
<code>comp(a, b)</code>	implicitly convertible to <code>bool</code>	Establishes <i>strict weak ordering</i> relation with the following properties <ul style="list-style-type: none"><li>• For all <code>a</code>, <code>comp(a, a) == false</code></li><li>• If <code>comp(a, b) == true</code> then <code>comp(b, a) == false</code></li><li>• if <code>comp(a, b) == true</code> and <code>comp(b, c) == true</code> then <code>comp(a, c) == true</code></li></ul>

[https://en.cppreference.com/w/cpp/named\\_req/Compare](https://en.cppreference.com/w/cpp/named_req/Compare)



# Thread [스레드]

---

프로세스 = 메모리 안에서 실행 중인 프로그램

스레드 = 프로세스 안에서 코드를 동작시키는 단위

하나의 프로그램이 하나의 스레드만 사용하면 싱글 스레드, 여러 개의 스레드를 사용하면 멀티스레드

스레드는 전역 변수를 이용해 서로 값을 공유할 수 있음 (프로세스와의 차이점)

일반적인 C++ 코드는 하나의 스레드만 이용 → CPU 코어가 여러 개여도 하나 밖에 사용을 못함, 나머지 코어는 놀고 있음

프로그래머 직접 여러 개의 스레드를 사용할 수 있게 코드를 바꿔줘야 함

# Thread

---

```
#include <iostream>
#include <thread>

// Function Pointer
void f1() {
    for (int i = 0; i < 5; i += 1) {
        std::cout << "f1();\n";
    }
}

// Functor
struct f2 {
    void operator()(int n) {
        for (int i = 0; i < 5; i += 1) {
            std::cout << "f2(" << n << ");\n";
        }
    }
};

// Lambda
auto f3 = [](int n, int m) {
    for (int i = 0; i < 5; i += 1) {
        std::cout << "f3(" << n << ", " << m << ");\n";
    }
};

int main() {
    std::thread t1(f1);
    std::thread t2(f2(), 10);
    std::thread t3(f3, 20, 30);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

std::thread를 이용해 새로운 스레드 생성

스레드의 생성자에는 실행할 함수를 넣어 줌

2번째 인자부터는 함수의 인자를 전달

스레드를 생성했으면 join해야 함

join은 생성한 스레드가 끝날 때까지 기다리는 역할

<https://en.cppreference.com/w/cpp/thread/thread>

# Thread

```
#include <iostream>
#include <thread>

// Function Pointer
void f1() {
    for (int i = 0; i < 5; i += 1) {
        std::cout << "f1();\n";
    }
}

// Functor
struct f2 {
    void operator()(int n) {
        for (int i = 0; i < 5; i += 1) {
            std::cout << "f2(" << n << ");\n";
        }
    }
};

// Lambda
auto f3 = [](int n, int m) {
    for (int i = 0; i < 5; i += 1) {
        std::cout << "f3(" << n << ", " << m << ");\n";
    }
};

int main() {
    std::thread t1(f1);
    std::thread t2(f2(), 10);
    std::thread t3(f3, 20, 30);
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

```
f3(f1());
f1();
20, f2(30);
10);
f1();
f1();
f3(f1());
20, 30);
f3(20, f2(10);
30);
f2(f3(10);
f2(10);
20, f2(30);
10);
f3(20, 30);
```

```
f1();
f1();
f2(f1());
10);
f2(f1());
10);
f1();
f1();
f2(f3(20, 10);
30);
f2(f3(10);
20, 30);
f3(f2(20, 10);
30);
f3(20, 30);
f3(20, 30);
```

```
f1();
f1();
f3(f2(10);
f2(f1());
10);
f1();
f1();
20, f2(30);
10);
f3(f2(20, 10);
30);
f2(10);
f3(20, 30);
f3(20, 30);
f3(20, 30);
```

매번 실행 결과가 달라짐

3개의 스레드가 경쟁하며 cout을 사용하면서 출력 결과가 엉망이 됨

# Mutex [뮤텍스]

---

상호 배제 (mutual exclusion)의 약자

여러 개의 스레드가 공유자원(cout 등)을 경쟁적으로 사용할 때, 특정 스레드만 공유자원을 독점해서 사용할 수 있도록 하는 기능

```
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex; 전역 변수로 mutex 선언

// Function Pointer
void f1() {
    for (int i = 0; i < 5; i += 1) {
        mutex.lock(); t1 스레드만 사용할 수 있게 mutex를 잠금, 나머지 스레드는 unlock 될 때까지 기다림
        std::cout << "f1();\n";
        mutex.unlock(); cout 사용이 끝나면 unlock 호출
    }
}
```

# Mutex

[https://en.cppreference.com/w/cpp/thread/counting\\_semaphore](https://en.cppreference.com/w/cpp/thread/counting_semaphore)

```
// Functor
struct f2 {
    void operator()(int n) {
        for (int i = 0; i < 5; i += 1) {
            std::lock_guard<std::mutex> lock(mutex);
            std::cout << "f2(" << n << ");\n";
        }
    }
};

// Lambda
auto f3 = [](int n, int m) {
    for (int i = 0; i < 5; i += 1) {
        std::lock_guard<std::mutex> lock(mutex);
        std::cout << "f3(" << n << ", " << m << ");\n";
    }
};
```

```
f1();
f1();
f1();
f1();
f1();
f2(10);
f2(10);
f2(10);
f2(10);
f2(10);
f3(20, 30);
f3(20, 30);
f3(20, 30);
f3(20, 30);
f3(20, 30);
```



# 4개의 스레드로 0~100 합 계산

---

```
std::mutex mutex;
int N;

void sum(int a, int b) {
    for (int n = a; n < b; n += 1) {
        std::lock_guard<std::mutex> lock(mutex);
        N += n;
    }
}

int main() {
    std::thread t1(sum, 0, 25);
    std::thread t2(sum, 25, 50);
    std::thread t3(sum, 50, 75);
    std::thread t4(sum, 75, 101);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    std::cout << N;
    return 0;
}
```

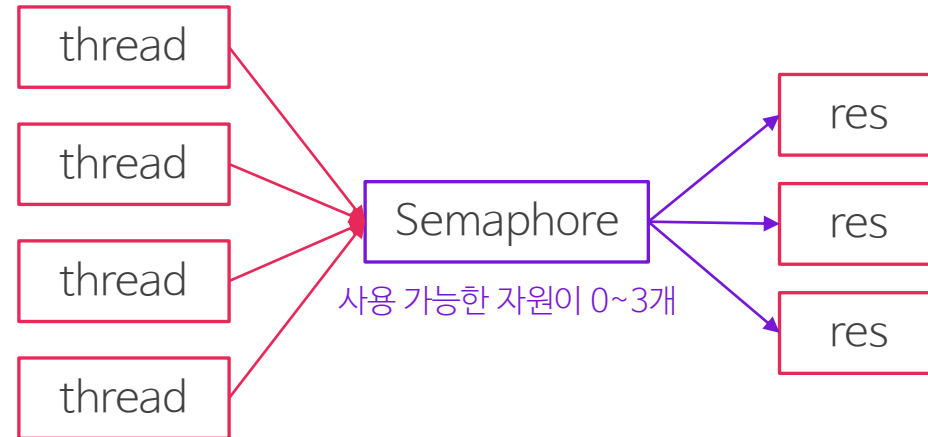
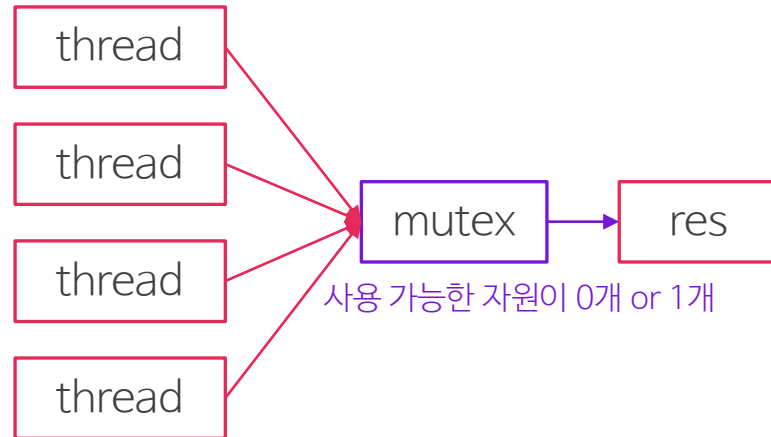
5050

# Semaphore [세마포어]

---

Mutex는 가질 수 있는 상태가 2가지 (locked, unlocked)

Semaphore는 가질 수 있는 상태가 N가지인 Mutex (바꿔 말하면 Mutex는 Binary Semaphore)



# Semaphore

```
#include <iostream>
#include <semaphore>
#include <thread>
```

```
std::counting_semaphore<1> mutex(1); <N>에 사용 가능한 최대 자원 개수, (n)에 현재 사용 가능한 자원 개수
int N;
```

```
void sum(int a, int b) {
    for (int n = a; n < b; n += 1) {
        mutex.acquire(); lock 대신 acquire
        N += n;
        mutex.release(); unlock 대신 release
    }
}
```

```
int main() {
    std::thread t1(sum, 0, 25);
    std::thread t2(sum, 25, 50);
    std::thread t3(sum, 50, 75);
    std::thread t4(sum, 75, 101);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    std::cout << N;
    return 0;
}
```

5050

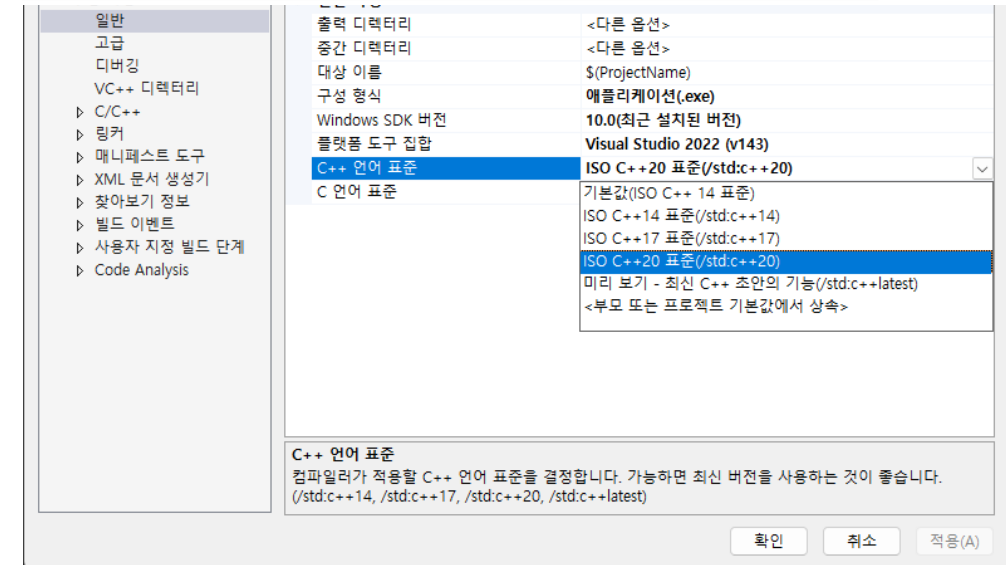
```
std::counting_semaphore, std::binary_semaphore
```

Defined in header <semaphore>

```
template<std::ptrdiff_t LeastMaxValue = /* implementation-defined */>
class counting_semaphore; (1) (since C++20)
```

```
using binary_semaphore = std::counting_semaphore<1>; (2) (since C++20)
```

C++20 이상에서만 사용 가능



# Quiz

여러 개의 Thread가 동시에  
push/pop 할 수 있는 Queue를  
구현하시오.

## quiz.h

```
#pragma once
#include <mutex>
#include <queue>

template<typename T>
class MutexQueue {
public:
    MutexQueue() = default;

    explicit MutexQueue(std::queue<T> queue)
        : queue(queue) {}

    int size() const {
        return queue.size();
    }

    T pop() {
        // todo
    }

    void push(T value) {
        std::lock_guard<std::mutex> lock(mutex);
        queue.push(value);
    }

private:
    std::mutex mutex;
    std::queue<T> queue;
};
```

## main.cpp

```
#include <iostream>
#include <thread>
#include "quiz.h"

MutexQueue<int> mq;

void push_numbers() {
    for (int n = 0; n < 100; n += 1) {
        mq.push(n);
    }
}

void pop_numbers() {
    for (int n = 0; n < 50; n += 1) {
        mq.pop();
    }
}

int main() {
    std::thread t1(push_numbers);
    std::thread t2(push_numbers);
    t1.join();
    t2.join();
    std::cout << "size: " << mq.size() << '\n'; // 200

    std::thread t3(pop_numbers);
    std::thread t4(pop_numbers);
    std::thread t5(pop_numbers);
    t3.join();
    t4.join();
    t5.join();
    std::cout << "size: " << mq.size() << '\n'; // 50
    return 0;
}
```

```
size: 200
size: 50
```

# Hint

---



```
T pop() {  
    // todo  
    const T value = queue.front();  
    queue.pop();  
    return value;  
}
```