Day-43

Functions
Function Pointers and Function Objects

Function Pointers

- 표준 C는 함수 포인터 개념을 통해 일반적인 프로그래밍 기능과 결합성을 달성함
- 함수는 다른 함수에 대한 포인터로 전달될 수 있으며 “간접 호출”로 동작

```
#include <iostream>
#include<cstdlib>
using namespace std;

int descending(const void* a, const void* b) {
    return *((const int*)a) < *((const int*)b);
}

int main()
{
    int arr[] = { 7, 2, 5, 1 };
    qsort(arr, 4, sizeof(int), descending);
    for (int i = 0; i < 4; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}

// 출력
// 7 5 2 1
```

```
#include <iostream>
#include<cstdlib>
```

```
using namespace std;

int eval(int a, int b, int (*f)(int, int)) {
    return f(a, b);
}

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


int main()
{
    cout << eval(4, 3, add) << endl;
    cout << eval(4, 3, sub) << endl;
    return 0;
}

// 출력
// 7
// 1
```

- 문제점
  - 안정성
    - 일반 케이스(ex : qsort)의 인자 타입에 대한 확인이 없음
  - 성능
    - 모든 연산은 원래 함수에 대한 간접 호출이 필요함
    - 함수 inlining이 불가능

Function Object (or Functor)

- function object 또는 functor은 매개변수로 취급할 수 있는 호출 가능한 객체
- C++은 다른 함수에 “procedure(프로시저)”를 전달하는 보다 효율적이고 편리한 방법을 제공함

```
#include <iostream>
#include<algorithm>
using namespace std;

struct Descending {
    bool operator()(int a, int b) {
        return a > b;
    }
};

int main()
{
    int arr[] = { 7, 2, 5, 1 };
    sort(arr, arr + 4, Descending{});
    for (auto i : arr) {
        cout << i << " ";
    }
    return 0;
}
// 출력
// 7 5 2 1
```

- 장점
  - 안정성
    - 인자 타입 검사가 항상 가능
    - 템플릿을 포함할 수 있음
  - 성능
    - 컴파일러는 대상 함수의 코드에 연산자()를 삽입한 다음 루틴을 컴파일
    - 연산자 인라이닝은 표준 동작
- C++11은 람다 표현식이라는 간단한 함수 객체를 제공하여 개념을 단순화