4강: algorithm header, lambda

SCSC 장필식

PPT 오류 수정

array initialization

default initialization: 기본값으로 모두 초기화

-> 참고: std::array<int, 3> arr1; 을 하면 값들이 0으로 초기화가 되지 않는다.

```
std::array<int, 3> arr1 = {};
```

initializer list 사용

```
std::array<int, 3> arr2 = {2, 3, 5};
//혹은
std::array<int, 3> arr2 = {{2, 3, 5}};
```

고차원 어레이

-> 참고: 다중 std::array 선언을 할때에는 중괄호를 두번 써주는게 좋다.

```
std::array<std::array<int, 3>, 3> mat = {{
    {{1, 2, 3}},
    {{4, 5, 6}},
    {{7, 8, 9}}
}}
```

예시: **string** 넘겨받기 **(오류)**

"Dongsu"라는 스트링을 복사하지 않기 위해서는 reference로 넘겨주는 게 좋다.

```
void greetPerson(std::string& name) {
    cout << "Hello, " << name << "!" << endl;
}

// 컴파일 안됨! 왜 그럴까?
int main() {
    greetPerson("Dongsu");
}
```

C++: implicit conversion

큰따옴표로 만든 string는 사실 STL string아니라 C string이다!

```
auto name = "Dongsu";
```

```
const char* name = "Dongsu"; // name의 실제 타입
```

C++: implicit conversion

```
std::string name = "Dongsu";
```

는 사실

```
std::string name = std::string("Dongsu");
```

로 자동으로 변환된다.

오류가 생긴 이유

`const char* name -> std::string` 으로는 자동으로 변환해주는 함수가 있지만,

`const char* name -> std::string&` 으로 변환해주는 함수는 없다!

```
void greetPerson(std::string& name) {
    cout << "Hello, " << name << "!" << endl;
}

// 컴파일 안됨! 왜 그럴까?
int main() {
    greetPerson("Dongsu");
}
```

예시: **string** 넘겨받기

하지만 다음과 같은 경우에는?

`const char* name -> const std::string&` 으로 자동으로 변환이 된다!

```
void greetPerson(const std::string& name) {
    cout << "Hello, " << name << "!" << endl;
}

// 이것도 됨! 왜 그럴까?
int main() {
    greetPerson("Dongsu");
}
```

정확한 이유는 몇주 후에...

그럼 이젠 본론으로

algorithm 헤더와 **Lambda**식

참고 자료:

- C++ Primer: Chap. 10
- http://en.cppreference.com/w/cpp/algorithm/find

Example: find

주어진 컨테이너에서 특정 원소를 찾고 싶을 때
첫번째로 찾은 놈에 대한 iterator를 반환한다.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

int main()
{
    int n1 = 3;
    int n2 = 5;
    std::vector<int> v{0, 1, 2, 3, 4};

    auto result1 = std::find(v.begin(), v.end(), n1);
    auto result2 = std::find(v.begin(), v.end(), n2);

    // result1는 3을 가리키는 iterator
    if (result1 != v.end()) {
        std::cout << "3 found inside vector!" << std::endl;
    }
    // result2는 v의 끝을 가리키고 있다
    if (result2 == v.end()) {
        std::cout << "Cannot find 5 inside vector!" << std::endl;
    }
}
```

Example: find_if

근데 특정한 값이 아니라 특정한 조건으로 찾고 싶다면?

예를 들어:

- 해당 vector에 10으로 나누어떨어지는 숫자가 있는지
- 주어진 단어들 중에 길이가 10자 이상을 넘어가는 단어가 있는지

Example: find_if

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

bool isDivisibleBy10(int i) {
    return (i % 10 == 0);
}
int main()
{
    std::vector<int> v{0, 4, 3, 6, 43, 20, 17}

    // 함수에 대한 레퍼런스를 받을 수 있다!
    auto result = std::find(v.begin(), v.end(), isDivisibleBy10);
    if (result != v.end()) {
        std::cout << *result << std::endl;
    }
    else {
        std::cout << "Not found..." << endl;
    }
}
```

흠.. **find_if**를 쓸 때 마다 함수를 정의해야 하나요

그럴 필요 없습니다.
즉석에서 함수를 만들 수 있는 기능이 있기 때문이죠.

Lambda expressions

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

int main()
{
    std::vector<int> v{0, 4, 3, 6, 43, 20, 17}

    // 함수에 대한 레퍼런스를 받을 수 있다!
    auto result = std::find(v.begin(), v.end(),
        [](int i) { return (i % 10 == 0); }
    );
    if (result != v.end()) {
        std::cout << *result << std::endl;
    }
    else {
        std::cout << "Not found..." << endl;
    }
}
```

Lambda expressions

함수를 변수처럼 취급하기!

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

std::cout << lambda(2, 3) << std::endl;
```

Lambda expression: syntax

[**captures**] (**params**) -> **return type** { **body** }

혹은

[**captures**] (**params**) { **body** }

Lambda expression: capture group

대괄호 안에 어떤 변수들을 참조할 것인지를 적을 수 있다.

Capture by value:

```
int n = 10;
auto adder = [n](int a) { return a + n; }
```

Capture by reference:

```
std::string greeting = "Hello, ";
auto greeter = [&greeting](std::string s) { return greeting + s; }
```

Lambda expression: capture group

혹은 자동으로 참조하게 할 수도 있다.

Capture everything by value:

```
int n = 10;
auto adder = [=](int a) { return a + n; }
```

Capture everything by reference:

```
std::string greeting = "Hello, ";
auto greeter = [&](std::string s) { return greeting + s; }
```

Lambda expression: std::function

Lambda 함수 자체는 구체적인 타입을 가지고 있진 않지만...
std::function이라는 타입으로 가지고 있을 수 있다.

```
std::function<void(int, int)> adder =
        [](int a, int b) { return a + b; }
```

다시 **algorithm** 헤더로...

Example: filtering a vector

주어진 vector에서 3으로 나누어떨어지는 놈들을 제외한 vector를 구하라.

Example: filtering a vector (without)

절차형 프로그래밍

```
std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9};

std::vector<int> newVec;
for (int i : vec) {
    if (i % 3 != 0) newVec.push(i);
}
```

Example: filtering a vector (with)

함수형 프로그래밍

```
std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9};

std::vector<int> newVec;
std::copy_if(vec.begin(), vec.end(), std::back_inserter(newVec),
    [](int i) { return i % 3 != 0; });
```

여담: 절차지향적 **vs** 함수지향적 프로그래밍

절차형 프로그래밍:

- 컴퓨터가 해야 하는 행동을 하나씩 순차적으로 명시
- 변수, 제어문, 반복문, 함수 실행 등을 순차적으로 조합해 새로운 기능들을 만듦

함수형 프로그래밍:

- 컴퓨터가 해야 하는 작업을 그대로 정의
- 함수들을 서로 조합해서 새로운 기능들을 만듦

어느게 더 좋고 나쁘다는 없다. 상황을 봐서 더 적절한 방식을 쓰면 된다.

Example: remove vector element by value

주어진 string에서 스페이스를 모두 없애고 싶을 때

```
#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>

int main()
{
    std::string str = "Text with some   spaces";
    str1.erase(std::remove(str1.begin(), str1.end(), ' '),
               str1.end());
    std::cout << str << std::endl;
}
```

```
Textwithsomespaces
```

끝...? 여담

STL 라이브러리: 좀 불편하긴 하다

흠... 다른 언어들처럼 함수형 프로그래밍 쓰기 쉽게 만들 수 없을까?

예

예를 들면... 1^2 + ... + 10^2는?

C++

```
std::vector<int> vec(10);
std::iota(vec.begin(), vec.end(), 1);
std::transform(vec.begin(), vec.end(), vec.begin(),
    [](int i) { return i * i; });
std::accumulate(vec.begin(), vec.end(), 0);
```

하스켈

```
sum $ map (\x -> x*x) [1..10]
```

Java example

심지어 Java에서도 이렇게 편리한데...

```
int total = Stream.iterate(0, i -> i + 1)
                  .map(i -> i * i)
                  .take(10)
                  .reduce(0, Integer::sum);
```

TS Ranges Proposal (2020년 예정!)

Eric Niebler가 설계한 새로운 STL 알고리즘 라이브러리의 초안.

```
int total = accumulate(view::iota(1) |
                       view::transform([](int x){ return x
                       view::take(10), 0);
```

진짜 끝

어캠갑시다