기초 C++ 프로그래밍 #2 보충자료

컴퓨터공학설계및실험 I

서강대학교
SOGANG UNIVERSITY

Table of Contents

Subscript Operator for Const Object (1)

```cpp
#include <iostream>
#include <algorithm>
#include <ctime>
#include <cassert>

class RandomArray {
private:
    size_t size;
    int* data;
public:
    RandomArray(size_t _size) : size(_size), data(new int[_size]) {
        std::srand(std::time(nullptr));
        this->update();
    }
    ~RandomArray() {
        delete[] data;
    }
    void update() {
        // fill in the array with the funciton rand()
        std::generate(data, data + size, std::rand);
    }
    int& operator[](const size_t index) const {
        assert(index < size && "RandomArray: index error");
        return data[index];
    }
};

int main() {
    const RandomArray ra(328);
    // ra.update();
    std::cout << ra[0] << std::endl;
    return 0;
}
```

- **const:**
  When it modifies a data declaration, the **const** keyword specifies that the object or variable isn't modifiable.

- **const** int x vs.
  const RandomArray ra(3)
  - ra.update()?
  - 함수 안에서 member variable이 바뀌지 않는 것이 보장되어야 함

- const member funciton
  - const { .. }, 함수 내부에서 member variable을 바꾸면 compile error
  - const object는 constructor 제외하고는 오직 const member function만 호출 가능하도록 제한

```
const.cpp:32:5: error: 'this' argument to member function 'update' has type 'const RandomArray', but function is not marked const
    ra.update();
    ^~
const.cpp:18:10: note: 'update' declared here
    void update() {
         ^
```

Subscript Operator for Const Object (2)

```
int main() {
    const RandomArray ra(328);
    std::cout << ra[0] << std::endl;
    ra[0] = 5;
    std::cout << ra[0] << std::endl;
    return 0;
}
```

```
g++ const.cpp
./a.out
32357914
5

                ???
```

```
int* data를 public으로 바꾼 후,

ra.data = nullptr; 추가

g++ const.cpp

const.cpp:33:13: error: cannot assign to va
riable 'ra' with const-qualified type 'cons
t RandomArray'
    ra.data = nullptr;
    ~~~~~~~ ^
const.cpp:32:23: note: variable 'ra' declar
ed const here
    const RandomArray ra(328);
    ~~~~~~~~~~~~~~~~~~~~^~~~~~~
1 error generated.
```

Subscript Operator for Const Object (3)

```
    // ...
    const int& operator[](const size_t index) const {
        assert(index < size && "RandomArray: index error");
        return data[index];
    }
    int& operator[](const size_t index) {
        assert(index < size && "RandomArray: index error");
        return data[index];
    }
};

int main() {
    const RandomArray ra(328);
    RandomArray ra2(7);
    ra[0] = 3;
    ra2[2] = 4;
    return 0;
}
```

```
const.cpp:34:11: error: cannot assign to return value
because function 'operator[]' returns a const value
    ra[0] = 3;
    ~~~~~ ^
const.cpp:22:11: note: function 'operator[]' which returns
const-qualified type 'const int &' declared here
    const int& operator[](const size_t index) const {
          ^~~~
1 error generated.
```

Template Programming: Motivation (1)

```
#include <iostream>

using namespace std;

float f(float* arr) {
    float result = 0;
    // more than 300 lines of code
    return result;
}

double f(double* arr) {
    double result = 0;
    // more than 300 lines of code
    return result;
}
```

- library를 만드는 programmer 입장에선, 여러 개의 type을 지원하기 위해서 동일한 code를 반복해서 작성해야 함
  - 확장성이 떨어지고,
  - 유지보수도 어려움:
    한 코드에 문제가 생긴 것을 뒤늦게 발견하면 나머지 함수들을 모두 수정해야 함

Template Programming: Motivation (2)

```
#define MAX(a, b) ((a) < (b) ? (a) : (b))

int g() {
    static int cnt = 1;
    cout << "g: " << cnt++ << endl;
    return 500;
}

int h() {
    static int cnt = 1;
    cout << "h: " << cnt++ << endl;
    return 404;
}


int main() {
    MAX(g(), h());
    return 0;
}
```

- Macro를 길게 써서 type-free function을 만들 수도 있지만,
- Macro는 pre-processor에 의해 단순 code substitution만 일어남
  - 실행 파일이 커지고,
  - 여러 줄 작성하기 불편함
  - Macro 정의 부분이 아닌 expanded code에서 error message
  - 불필요한 evaluation:

[실행 결과]

```
./example
g: 1
h: 1
h: 2
```

Template Programming: function templates (1)

```cpp
#include <iostream>
#include <utility>  // std::pair
#include <vector>

using namespace std;

template<typename T, typename U>
vector<pair<T, U> > Cartesian(
    const vector<T>& vec1,
    const vector<U>& vec2
) {
    vector<pair<T, U> > result;
    result.reserve(vec1.size() * vec2.size());
    for (const auto& element1: vec1) {
        for (const auto& element2: vec2) {
            result.emplace_back(element1, element2);
        }
    }
    return result;
}

int main() {
    vector<int> vec1 = {1, 2};
    vector<char> vec2 = {'a', 'b', 'c'};
    const auto vec3 = Cartesian(vec1, vec2);
    for (const auto& element: vec3) {
        cout << "(" << element.first << ", ";
        cout << element.second << ")" << endl;
    }
}
```

- **Instantiation**:
  template function의 parameter들은 compile time에 type이 결정됨

- typename 대신 class로 써도 동일

from ISO C++ 17.1.2:

- There is no semantic difference between class and typename in a *type-parameter-key*

```
g++ template_fn1.cpp -std=c++17
./a.out
(1, a)
(1, b)
(1, c)
(2, a)
(2, b)
(2, c)
```

Template Programming: function templates (2)

```
#include <iostream>
#include <cmath>    // std::fabs

using namespace std;

const float EPSILON = 1e-3;

template<typename T>
int compare(const T& a, const T& b) {
    if (a < b)          return 1;
    else if (a == b)    return 0;
    else                return -1;
}

template<>
int compare(const float& a, const float& b) {
    if (fabs(a - b) < EPSILON)
        return 0;
    else if (a < b)     return 1;
    else                return -1;
}

int main() {
    cout << compare('a', 'b') << endl;
    cout << compare(1.0f, 1.00001f) << endl;
    return 0;
}
```

- **specialization:**
  일부 type에 대해서 함수가 다른 동작을 할 수 있도록 별도로 정의.

- 예를 들면, standard library의 동적 배열 class인 vector는 bool type에 대해서 원소 하나당 1byte가 아니라 1bit로 줄임

```
g++ template_fn2.cpp -std=c++17
./a.out
1
0
```

Template Programming: class templates (1)

```cpp
#include <iostream>
#include <algorithm>    // std::fill
#include <random>       // std::rand
#include <cassert>

class RandomInteger {
public:
    int value;
    RandomInteger() : value(rand()) {}
};

template<typename T, size_t size>
class Array {
protected:
    T data[size];
public:
    // initilize data using default value/constructor
    Array() = default; // since c++11
    explicit Array(const T& _init) {
        std::fill(data, data + size, _init);
    }
    const T& operator[](const size_t index) const {
        assert(index < size && "Array index error");
        return data[index];
    }
    T& operator[](const size_t index) {
        assert(index < size && "Array index error");
        return data[index];
    }
    void print(const std::string& msg = "Array") const {
        std::cout << msg << ": ";
        for (const auto& x : data) {
            std::cout << x << ' ';
        }
        std::cout << std::endl;
    }
};
```

- Template이 적용된 Array
- size_t: *NTTP*
  Non-Type Template Parameter
  - constant expression만 가능
  - int n = 4;
    Array<int, n>: compile error

Template Programming: class templates (2)

```
template<typename T, size_t size>
class RangeArray : public Array<T, size> {
private:
    int base;
public:
    explicit RangeArray(int _base)
        : Array<T, size>() // call the base constructor
        , base(_base) {}
    RangeArray(int _base, T _init)
        : Array<T, size>(_init)
        , base(_base) {}
    const T& operator[](const size_t index) const {
        return Array<T, size>::operator[](index - base);
    }
    T& operator[](const size_t index) {
        return Array<T, size>::operator[](index - base);
    }
    void print_details() const {
        // print("RangeArray");
        Array<T, size>::print("RangeArray");
        std::cout << "base: " << base << std::endl;
    }
};

int main() {
    Array<int, 4> int_arr(-20);
    int_arr.print();

    const Array<RandomInteger, 3> randint_arr;
    std::cout << randint_arr[0].value << std::endl;

    RangeArray<int, 4> int_rangearr(-10);
    int_rangearr.print_details();

    const RangeArray<double, 3> db_rangearr(-10, 3.14);
    std::cout << db_rangearr[-9] << std::endl;
    return 0;
}
```

- Base Class에 대해 올바른 template parameter와 constructor 호출
- Derived class에서는 Template Base class의 함수를 바로 쓸 수 없음.
  - print("RangeArray") X

→ template instantiation이 일어나기 전이므로 compiler는 함수 이름만 가지고 base class의 함수인지 알 수 없기 때문.

(아래와 같이 header와 implementation을 분리하고 컴파일 할 수도 있고, specialized instance가 만들어질지 모름)

```
make
g++ -O3 -std=c++11 -c -o
RangeArray.o RangeArray.cpp
g++ -O3 -std=c++11 -c -o main.o main.cpp
g++ -o main main.o RangeArray.o
```

```
g++ template_class1.cpp -std=c++11
./a.out
Array: -20 -20 -20 -20
16807
RangeArray: 0 0 0 0
base: -10
3.14
```

Template Programming: standard library (1)

```cpp
#include <iostream>
#include <map>
#include <string>
#include <utility>
#include <vector>

// Type alias (since C++11), similar to -
// typedef std::pair<double, double> point2D
using Point2D = std::pair<double, double>;

int main() {
    std::vector<Point2D> points;
    std::map<std::string, int> majorcode;

    majorcode["Computer Science"] = 101;
    majorcode["Economics"] = 202; majorcode["Medical"] = 404;

    // Usage example
    std::cout << majorcode["Computer Science"] << std::endl;

    points.push_back(Point2D(1.0, 3.0));
    points.push_back(Point2D(1.3, 2.3));

    // ranged based for loop (since C++11) with
    // structured binding declaration (since C++17)
    for (const auto& [key, value] : majorcode) {
        std::cout << "major: " << key << ", ";
        std::cout << "code: " << value << std::endl;
    }
    for (const auto& [x, y] : points) {
        std::cout << "(x, y): " << x << ", "  << y << std::endl;
    }
    return 0;
}
```

std::map

Defined in header <map>

```cpp
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

C++ standard library는 여러가지 templated algorithm & data structure 제공

[실행 결과]

```
g++ -o examples std_examples.cpp -std=c++17
./examples
101
major: Computer Science, code: 101
major: Economics, code: 202
major: Medical, code: 404
(x, y): 1, 3
(x, y): 1.3, 2.3
```

C++17: cspro1, cspro2, cspro3의 g++에서 지원 확인 (22/03/27)

Template Programming: standard library (2)

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

class Store {
public:
    int star;           // 1 to 5
    float distance;     // km
    bool operator>(const Store& rhs) const {
        if (star > rhs.star)
            return true;
        else if (star == rhs.star)
            return distance < rhs.distance;
        else
            return false;
    }
};

int main() {
    std::vector<Store> storeDB = {{3, 1.2}, {3, 0.2}, {5, 3}};
    std::sort(
        storeDB.begin(),
        storeDB.end(),
        std::greater<>()
    );
    for (const auto& [star, dist] : storeDB) {
        std::cout << "star: " << star << ", ";
        std::cout << "dist: " << dist << std::endl;
    }
    return 0;
}
```

std::greater

Defined in header <functional>

```cpp
template< class T >
struct greater;
```
(until C++14)

```cpp
template< class T = void >
struct greater;
```
(since C++14)

Function object for performing comparisons. Unless specialized, invokes `operator>` on type T.

- Operator Overloading만 해준다면, 원하는 대로 class sort 가능하다.
- std::sort는 비교 함수(greater)에 따라서 동작할 것이고, 비교(>) 결과만 알 수 있다면 type에 무관하게 정렬 수행

Template Programming: Remark

- Following Stepanov, we can define generic programming without mentionin g language features: **Lift algorithms and data structures from concrete ex amples to their most general and abstract form.**

  - Stroustrup, B. (2007, June). Evolving a language in and for the real world: C++ 1991–2006. In *Proceedi ngs of the third ACM SIGPLAN conference on History of programming languages* (pp. 4-1).

Virtual function (1)

```
#include <iostream>

using namespace std;

class Base {
public:
    Base() { print(); }
    virtual void print() {
        cout << "print base" << endl;
    }
};

class Derived : public Base {
public:
    Derived() : Base() {}
    void print() override {
        cout << "print derived" << endl;
    }
};

int main() {
    Base b;
    Derived d;
    d.print();
    return 0;
}
```

- virtual function이 있는 class는 compiler가 내부적으로 function pointer들이 담긴 vtable, vptr을 생성, runtime에 올바른 함수를 호출
  - dynamic dispatch / late binding
  - 직접 호출(early binding)보다 느림
- Instance가 완전히 생성된 다음에 virtual function을 사용해야 함
- override (since C++11):
  base class의 **virtual function**을 override했다는 것을 명시

Virtual function (2)

```
class Vehicle {
protected:
    int fuel = 100;
public:
    virtual std::string get_status() const {
        return "Vehicle ok";
    }
};

class Airplane : public Vehicle {
public:
    std::string get_status();
};

class Boat : public Vehicle {
public:
    std::string get_status();
};

std::string Airplane::get_status() {
    std::string msg;
    msg = (fuel = 0) ? "cannot move" : "Airplane ok";
    return msg;
}

int main() {
    Airplane ap;
    Boat bt;
    Vehicle* ptr;
    if ( 1 /* some condition */ )
        ptr = &ap;
    else
        ptr = &bt;
    std::cout << ptr->get_status();
    return 0;
}
```

- class Airplane의 문제점은?
- get_status 뒤에 override를 붙이고 compile 했다면?

```
override.cpp:8:25: note: hidden overloaded virtual
function 'Vehicle::get_status' declared here:
different qualifiers ('const' vs unqualified)
    virtual std::string get_status() const {
                        ^

1 error generated.
```

References

- https://en.cppreference.com/w/
- https://isocpp.org/
- https://cppcon.org/