

#01-3. PS를 위한 C++

나정휘

<https://justicehui.github.io/>

목차

iostream

std::string

namespace

std::vector와 template

레퍼런스

range-based for문

함수 오버로딩

bits/stdc++.h

iostream

주의 사항

학부 1학년을 타겟으로 제작한 자료

- 의도적으로 설명을 생략하고 넘어가는 부분이 있음
- 특히 클래스, 템플릿, 특수화, trait 등 1학년 1학기에 배우지 않는 내용

자세한 내용이 궁금한 사람들은...

- <https://en.cppreference.com/w/cpp/io>
- <https://www.amazon.com/C-Programming-Language-4th/dp/0321563840>

iostream

iostream

- input/output stream - 스트림을 통한 표준 입출력 기능을 제공하는 헤더 파일
- 스트림: 입출력 작업을 추상화한 것
 - 키보드를 통한 입력, 모니터를 통한 출력, 파일에 읽고 쓰는 작업 등은 모두 스트림을 통해서 진행
 - 우리가 코드를 작성할 때는 입출력 장치의 종류와 무관하게 스트림만 사용하면 됨
 - 파일 입출력은 fstream 헤더 파일에 있음
- 우리는 std::cin과 std::cout을 사용함
 - cin = character input = char input
 - char(-128~127, 아스키코드)로 표현되는 문자들의 입력
 - wcin = wide character input
 - wchar_t로 표현되는 문자들의 입력
 - wchar_t: 더 넓은 범위의 문자(ex. 한글, 한자 등)를 표현하기 위한 자료형

```
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>


namespace std {
    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;

    extern wistream wcin;
    extern wostream wcout;
    extern wostream wcerr;
    extern wostream wclog;
}
```

iostream

std::cin, std::cout

- scanf, printf와 다르게 format string(ex. %d %f) 필요 없음
- >> 연산: 추출(extraction) 연산
 - 입력 스트림에서 값을 추출하는 것
- << 연산: 삽입(insertion) 연산
 - 출력 스트림에 값을 삽입하는 것



```
// BOJ 10953. A+B - 6  
// 각 줄에 A와 B가 주어진다.  
// A와 B는 콤마(,)로 구분되어 있다.
```

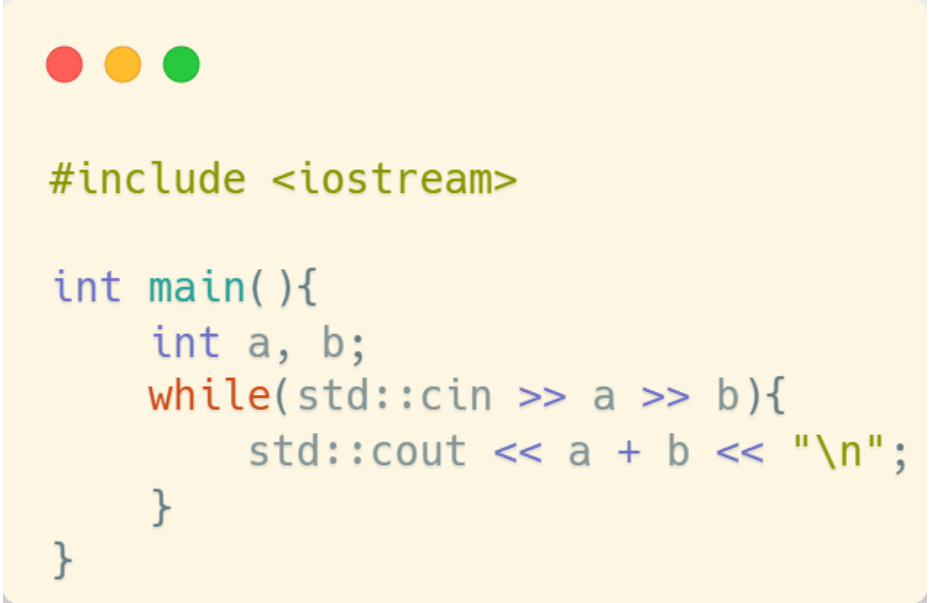
```
#include <iostream>
```

```
int main(){  
    int T;  
    std::cin >> T;  
    while(T--){  
        int a, b; char c;  
        std::cin >> a >> c >> b;  
        std::cout << a + b << "\n";  
    }  
}
```

iostream

BOJ 10951. A+B - 4

- EOF를 처리해야 하는 문제
- cin의 반환값은 bool(true 또는 false)타입으로 캐스팅할 수 있음
 - https://en.cppreference.com/w/cpp/io/basic_ios/operator_bool
 - EOF를 만나면 false를 반환함



```
#include <iostream>

int main(){
    int a, b;
    while(std::cin >> a >> b){
        std::cout << a + b << "\n";
    }
}
```

iostream

소수점 출력

- BOJ 1008. A/B
- iomanip 헤더에 있는 std::setprecision 사용
 - std::setprecision(n): 소수점 아래 n자리 수까지 사용하도록 지정
 - std::fixed: 정확히 setprecision에서 지정한 자리 만큼 출력하도록 설정
 - std::fixed 안 쓰면 n자리보다 적게 출력할 수도 있음



```
#include <iostream>
#include <iomanip>

int main(){
    int a, b;
    std::cin >> a >> b;
    std::cout << std::fixed << std::setprecision(10) << 1.0 * a / b;
}
```


질문?

iostream

버퍼

- 매번 실제로 콘솔창에 출력하는 것은 시간이 오래 걸림
 - 따라서 프로그램은 임시 공간(버퍼)에 출력할 내용을 저장한 다음
 - 버퍼가 적당히 많이 차면 버퍼에 저장된 데이터를 출력 스트림을 통해 출력하고 버퍼를 비움(flush)
- 입력도 마찬가지
 - 키보드 누를 때마다 보내면 느리기 때문에 버퍼에 적당히 모아서 보냄
 - 버퍼가 아직 비워지지 않았다면 지우고 수정할 수 있다는 장점도 있음
 - 보통 엔터 누르면 입력 버퍼를 비움
- cout 버퍼를 비우는 방법
 - `std::cout << std::flush;`
 - 버퍼를 비움
 - `std::cout << std::endl;`
 - 개행 문자를 출력하고 버퍼를 비움

iostream

BOJ 15552. 빠른 A+B

```
1 #include <iostream>
2
3 int main(){
4     int T;
5     std::cin >> T;
6     while(T--){
7         int a, b;
8         std::cin >> a >> b;
9         std::cout << a + b << std::endl;
10    }
11 }
```

소스 코드 공개 ☐ 공개 ☒ 비공개 ☐ 맞았을 때만 공개

수정

제출 번호	아이디	문제	문제 제목	결과
62453500	jhnan917	15552	빠른 A+B	시간 초과

iostream

BOJ 15552. 빠른 A+B

```
1 #include <iostream>
2
3 int main(){
4     std::ios_base::sync_with_stdio(false);
5     std::cin.tie(nullptr);
6     int T;
7     std::cin >> T;
8     while(T--){
9         int a, b;
10        std::cin >> a >> b;
11        std::cout << a + b << "\n";
12    }
13 }
```

소스 코드 공개 ☐ 공개 ☒ 비공개 ☐ 맞았을 때만 공개 [수정](#)

제출 번호	아이디	문제	문제 제목	결과
62453505	jhnan917	15552	빠른 A+B	맞았습니다!!

iostream

BOJ 15552. 빠른 A+B

- `sync_with_stdio(false);`
 - C++은 C의 기능을 대부분 지원함
 - 따라서 기본적으로 `iostream`과 `stdio.h`가 연동되어 있음
 - `sync_with_stdio(false);` 는 `stdio.h`와의 연동을 끊음
 - 속도가 빨라지는 대신 더 이상 `scanf/printf`를 사용하면 안 됨
- `cin.tie(nullptr);`
 - 기본적으로 `cin`은 `cout`과 `tie`되어 있음
 - `cin`이 호출되면 `cout`의 버퍼를 비움
 - `cin.tie(nullptr)`로 연결을 풀어주면 `flush` 안 해서 빨라짐
- `std::endl` → 개행 문자
 - `endl`은 개행 문자를 출력한 다음 버퍼를 비움
 - 버퍼를 비우는 것은 느리므로 개행 문자만 출력하도록 수정



```
#include <iostream>

int main(){
    int T;
    std::cin >> T;
    while(T--){
        int a, b;
        std::cin >> a >> b;
        std::cout << a + b << std::endl;
    }
}
```



```
#include <iostream>

int main(){
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int T;
    std::cin >> T;
    while(T--){
        int a, b;
        std::cin >> a >> b;
        std::cout << a + b << "\n";
    }
}
```

iostream

주의 사항

- sync 끊은 다음에 cin/scanf, cout/printf 섞어서 쓰면 안 됨
- tie 풀면 마지막에 모든 출력이 다 나올 수 있으니 당황하지 말기

```
8 while(T--){
9     int a, b;
10    if(T % 2 == 0){
11        std::cin >> a >> b;
12        std::cout << a + b << "\n";
13    }
14    else{
15        scanf("%d %d", &a, &b);
16        printf("%d\n", a + b);
17    }
18 }
19 }
```

소스 코드 공개 ☐ 공개 ☒ 비공개 ☐ 맞았을 때만 공개 [수정](#)

제출 번호	아이디	문제	문제 제목	결과
62453514	jhn917	15552	빠른 A+B	틀렸습니다



```
#include <iostream>
```

```
int main(){
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int T;
    std::cin >> T;
    while(T--){
        int a, b;
        std::cin >> a >> b;
        std::cout << a + b << "\n";
    }
}
```

```
3
1 2
3 4
5 6
3
7
11
```

질문?

std::string

std::string

std::string

- C언어의 문자열(문자들의 배열)은 매우 불편함
 - strlen, strcmp, strcpy, strcat, strtok, strstr, ...
- C++에는 std::string이라는 클래스가 있음
 - #include <string>
 - C-style string보다 훨씬 편함

```
1234
123456
0
1
4
1 1 4 4
1234
0 1
```

```
#include <iostream>
#include <string>

int main(){
    std::string a = "12", b = "34";
    std::cout << a + b << "\n";    // 1234
    a += b;
    b = "56";
    std::cout << a + b << "\n";    // 123456
    std::cout << (a == b) << "\n"; // 0
    b = "1234";
    std::cout << (a == b) << "\n"; // 1
    std::cout << a.size() << "\n"; // 4

    std::cout << a[0] << " ";
    std::cout << a.front() << " ";
    std::cout << a[a.size()-1] << " ";
    std::cout << a.back() << "\n";
    for(int i=0; i<a.length(); i++) std::cout << a[i];
    std::cout << "\n";

    a.clear();
    std::cout << a.size() << " " << a.empty() << "\n";
}
```

std::string

BOJ 11718. 그대로 출력하기

- 입력받은 내용을 그대로 출력하는 문제
- cin은 공백 단위로 끊어서 입력받음
- 띄어쓰기가 연속해서 여러 번 주어지면?
- 한 줄을 통째로 입력받는 방법
 - std::getline
 - 반환값은 cin과 동일

```
#include <iostream>
#include <string>

int main(){
    std::string s;
    while(std::getline(std::cin, s)){
        std::cout << s << "\n";
    }
}
```

질문?

namespace

namespace

namespace

- A라는 사람이 foo() 라는 함수를 만들었는데 B라는 사람도 foo() 라는 함수를 만들면?
- 똑같은 이름의 함수가 2개 있으므로 컴파일 오류
- 해결 방법
 - 한 명이 함수 이름을 바꾼다.
 - 각자 namespace를 만든다.



```
#include <iostream>

namespace a{
    void foo(){ std::cout << "namespace a\n"; }
}
namespace b{
    void foo(){ std::cout << "namespace b\n"; }
}

int main(){
    a::foo();
    b::foo();
}
```

namespace

namespace std

- 표준 라이브러리에 있는 모든 함수, 클래스는 std 안에 있음
 - std::cin, std::cout, std::string, ...
 - sync_with_stdio는 namespace std 안에 있는 namespace ios_base 안에 있음

```
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;

    extern wistream wcin;
    extern wostream wcout;
    extern wostream wcerr;
    extern wostream wclog;
}
```

namespace

using 선언문

- using 선언문을 이용해 namespace를 지정하지 않을 수 있음
 - ex) using std::cin; 하면 앞으로 코드에서 발견되지 않는 cin은 std::cin으로 취급
 - 단, using보다 더 안쪽에서 선언된 cin이 있으면 std::cin이 아닌 cin으로 취급



```
#include <iostream>
using std::cin;

int main(){
    int a, b;
    cin >> a >> b;
    std::cout << a + b;
}
```



```
#include <iostream>
using std::cin;

int main(){
    int a, b, cin;
    cin >> a >> b; // 입력받는 거 아님
    std::cout << a + b;
}
```

namespace

using 지시문

- using 지시문을 이용해 해당 namespace에 있는 것을 모두 사용할 수 있음
 - ex) using namespace std; 를 사용하면 앞으로 코드에서 안 보이는 식별자는 std에서 찾을 수 있음
 - 단, 전역에 cin이라는 이름이 선언되어 있으면 이름 충돌해서 컴파일 에러



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

int main(){
    int a, b;
    int string = 1;
    cin >> a >> b; // 코드에 cin 없으므로 std::cin
    cout << a + b; // 코드에 cout 없으므로 std::cout
    cout << string;
    // 이 string은 std::string이 아닌 int string = 1에서 선언된 것
}
```


질문?

std::vector와 template

std::vector와 template

std::vector

- 동적 배열: 들어있는 원소의 개수에 따라 크기가 동적으로 바뀌는 배열
 - C언어의 배열은 정적 배열 → 한 번 선언하면 크기가 바뀌지 않음
- Python의 list처럼 맨 뒤에서 원소 삽입/삭제 가능
- std::string처럼 size, empty, clear, front, back, 인덱스 접근 가능

```

#include <iostream>
#include <vector>
using namespace std;

int main(){
    // 원소의 타입을 지정해야 함
    // vector<int>는 int형 데이터를 저장하는 벡터
    // vector<double>은 double형 데이터를 저장하는 벡터

    vector<int> empty_vector; // 빈 벡터 선언
    vector<int> sized_vector(5); // 크기를 미리 지정할 수 있음, 0으로 초기화
    vector<int> init_vector(5, 1); // 초기값을 지정할 수 있음, 모두 1로 초기화
    vector<int> init_list_vector = {1, 2, 3, 4, 5}; // 이렇게 초기화할 수도 있음

    vector<int> v;
    for(int i=0; i<5; i++) v.push_back(i * 2); // 맨 뒤에 삽입
    for(int i=0; i<5; i++) cout << v[i] << "\n"; // 인덱스 접근 가능
    v.pop_back(); // 맨 뒤 원소 삭제
    cout << v.size() << " " << v.empty() << "\n"; // 원소의 개수, 비어있는지 확인
    cout << v.front() << " " << v.back() << "\n"; // 맨 앞/뒤 원소

    v.clear(); // 모든 원소 삭제
    cout << v.size() << " " << v.empty() << "\n";

}
```

std::vector와 template

std::vector

- 맨 뒤가 아닌 다른 곳에서도 삽입/삭제가 가능하지만 별로 안 좋음
 - $v[i]$ 에 새로운 원소를 삽입하면 기존에 $v[i]$, $v[i+1]$, $v[i+2]$, ... 에 있던 원소를 한 칸씩 뒤로 밀어야 함
 - 마찬가지로 $v[i]$ 를 삭제하면 $v[i+1]$, $v[i+2]$, ... 에 있는 원소를 한 칸씩 앞으로 이동시켜야 함
- 맨 앞에서 삽입/삭제를 하고 싶으면 std::deque 사용
 - push_front, pop_front 사용 가능
 - 하지만 push_front, pop_front를 제외하면 대부분 std::vector보다 속도가 느림
 - 맨 뒤 삽입/삭제, 순차 탐색, 인덱스 접근 등
 - 자세한 이유는 나중에!

std::vector와 template

std::vector

- size() 의 반환형은 size_t(주로 unsigned int 또는 unsigned long long)이라는 것에 주의해야 함
 - v가 빈 벡터일 때 v.size() - 1은 아주 큰 값 (size_t에서 가능한 최댓값)
 - 이 코드에서 런타임 오류 발생
 - $i+1 < v.size()$ 를 쓰면 안전함

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

void print_except_last_element(vector<int> v){
    for(int i=0; i<v.size()-1; i++) cout << v[i];
    cout << "\n";
}

int main(){
    vector<int> v = {1, 2, 3};
    print_except_last_element(v); // v = {1, 2, 3}
    v.pop_back();
    print_except_last_element(v); // v = {1, 2}
    v.pop_back();
    print_except_last_element(v); // v = {1}
    v.pop_back();
    print_except_last_element(v); // v = {}
}
```

질문?

std::vector와 template

template

- 코드를 찍어내는 틀
 - 아래 코드에서 add1과 add2는 타입을 제외한 모든 부분이 동일함
 - `T add(T a, T b){ return a + b; }` 라는 틀을 만든 다음 T에 적절한 타입만 대입해서 찍어내도 됨
 - 컴파일러는 `add<int>`에 대한 코드와 `add<float>`에 대한 코드를 각각 생성함
 - `vector<int>`와 `vector<float>`도 저장하는 타입을 제외하면 나머지 부분은 동일함
 - `vector<T>` 라는 틀을 만든 다음 T에 `int`와 `float`를 각각 대입해서 찍어내면 됨

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

int add1(int a, int b){ return a + b; }
float add2(float a, float b){ return a + b; }

template<typename T>
T add(T a, T b){ return a + b; }

int main(){
    cout << add1(1, 2) << "\n";
    cout << add2(1.1, 2.2) << "\n";

    cout << add<int>(1, 2) << "\n";
    cout << add<float>(1.1, 2.2) << "\n";
}
```

```
int add<int>(int, int):
```

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
pop     rbp
ret
```

```
float add<float>(float, float):
```

```
push    rbp
mov     rbp, rsp
movss   DWORD PTR [rbp-4], xmm0
movss   DWORD PTR [rbp-8], xmm1
movss   xmm0, DWORD PTR [rbp-4]
addss   xmm0, DWORD PTR [rbp-8]
pop     rbp
ret
```

std::vector와 template

template

- algorithm 헤더에 있는 std::min와 std::max도 템플릿 함수
 - template<typename T> T min(T a, T b){ return a < b ? a : b; } 와 같은 형태
 - 사실은 조금 다름



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

int main(){
    int a = 1, b = 2;
    // 컴파일러가 알아서 타입 지정함
    cout << min(a, b) << " " << max(a, b) << "\n";
    // 물론 직접 명시할 수도 있음
    cout << min<int>(a, b) << " " << max<int>(a, b) << "\n";

    double c = 1.1, d = 2.2;
    cout << min(c, d) << " " << max(c, d) << "\n";
    // 타입을 int로 명시하면 c와 d가 int로 캐스팅돼서 전달, 처리됨
    cout << min<int>(c, d) << " " << max<int>(c, d) << "\n";
}
```


std::vector와 template

template

- algorithm 헤더에 있는 std::min와 std::max도 템플릿 함수
 - 두 값의 타입이 다르면 컴파일 오류 발생 → $T \min(T a, T b)$ 형태가 아니기 때문
 - `min<int>(a, b)` 처럼 타입을 명시하면 됨



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

int main(){
    int a = 1;
    long long b = 2;
    cout << min<int>(a, b) << " " << max<int>(a, b) << "\n"; // 가능
    cout << min(a, b) << " " << max(a, b) << "\n"; // 불가능
}
```

질문?

레퍼런스

레퍼런스

함수 파라미터 전달 방법

- call by value
 - 값을 복사해서 전달
 - 만약 길이가 100인 벡터를 파라미터로 전달하면 100개의 원소를 모두 복사함 → 길이가 길면 오래 걸림
 - 복사해서 만든 새로운 객체이므로 함수 안에서 값을 수정하더라도 원본 데이터에 영향을 주지 않음

```
● ● ●

#include <iostream>
#include <vector>
using namespace std;

class temp{
public:
    temp(){ cout << "construct "; } // 생성될 때 호출
    temp(const temp &obj){ cout << "copy "; } // 복사될 때 호출
};

void f(vector<temp> v){}
void g(int a){ a = 2; }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    vector<temp> v(3);
    f(v);
    int a = 1;
    g(a);
    cout << a;
}
// construct construct construct copy copy copy 1
```

레퍼런스

함수 파라미터 전달 방법

- call by reference
 - 원본 데이터에 접근할 수 있는 방법(보통 주소값)을 구성해서 전달, alias를 생성한다고 생각하면 됨
 - 함수 안에서 값을 수정하면 원본 데이터도 수정됨
 - 값을 복사하는 것이 아니므로 전달하고자 하는 벡터의 크기가 커도 시간이 오래 걸리지 않음

```
● ● ●

#include <iostream>
#include <vector>
using namespace std;

class temp{
public:
    temp(){ cout << "construct "; } // 생성될 때 호출
    temp(const temp &obj){ cout << "copy "; } // 복사될 때 호출
};

void f(vector<temp> &v){} // 타입 뒤에 &를 붙임
void g(int &a){ a = 2; }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    vector<temp> v(3);
    f(v);
    int a = 1;
    g(a);
    cout << a;
}
// construct construct construct 2
```

레퍼런스

함수 파라미터 전달 방법

- 두 변수의 값 교환
 - call by value로 전달하면 원본 데이터에 영향을 주지 않으므로 교환 안 됨
 - call by reference로 전달해야 함

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

void fake_swap(int a, int b){
    cout << "(fake) before swap : " << a << " " << b << "\n"; // 1 2
    int t = a; a = b; b = t;
    cout << "(fake) after swap : " << a << " " << b << "\n"; // 2 1
}

void real_swap(int &a, int &b){
    cout << "(real) before swap : " << a << " " << b << "\n"; // 1 2
    int t = a; a = b; b = t;
    cout << "(real) after swap : " << a << " " << b << "\n"; // 2 1
}

int main(){
    int a = 1, b = 2;
    fake_swap(a, b);
    cout << "(fake) result : " << a << " " << b << "\n"; // 1 2
    real_swap(a, b);
    cout << "(real) result : " << a << " " << b << "\n"; // 2 1
}
```

레퍼런스

레퍼런스 변수

- 별명을 붙인다고 생각하면 됨
 - `int a = 1, &b = a;` 는 a에게 b라는 별명을 붙이는 것
- 레퍼런스 변수는 선언할 때 어떤 것을 참조할지 지정해야 함
 - `int &c;` 는 컴파일 에러



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

int main(){
    int a = 1; int &b = a;
    cout << a << " " << b << "\n"; // 1 1
    a = 2;
    cout << a << " " << b << "\n"; // 2 2
    b = 3;
    cout << a << " " << b << "\n"; // 3 3

    int &c; // 컴파일 에러, 선언할 때 초기화도 해야 함
}
```



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

int factorial_save[11];
// n!을 계산한 적 있으면 factorial_save[n] = n!
// 그렇지 않으면 factorial_save[n] = 0

int factorial(int n){
    if(n == 1) return 1;
    int &res = factorial_save[n];
    if(res == 0){ // if(factorial_save[n] == 0)
        res = n * factorial(n-1);
        // factorial_save[n] = n * factorial(n-1)
    }
    return res; // return factorial_save[n]
}

int main(){
    for(int i=1; i<=10; i++) cout << factorial(i) << "\n";
}
```

질문?

range-based for문

range-based for문

range-based for문

- for(타입 변수이름 : 데이터리스트)
 - 타입 앞에 다른 게 붙는 경우도 있는데 지금은 생략...
 - for(int i : v) 에서 i는 v의 원소를 차례대로 순회함



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

int main(){
    vector<int> v = {1, 3, 5};
    for(int i : v) cout << i << " "; // 1 3 5
    cout << "\n";

    for(int i : v) i *= 2; // call by value
    for(int i : v) cout << i << " "; // 1 3 5
    cout << "\n";

    for(int &i : v) i *= 2; // call by reference
    for(int i : v) cout << i << " "; // 2 6 10
    cout << "\n";
}
```

함수 오버로딩

함수 오버로딩

함수 오버로딩

- 매개 변수의 구성이 다르면 이름이 같은 함수를 여러 개 만들 수 있음
- 단, 함수의 반환형은 오버로딩에서 고려하지 않음
 - int f(int a); double f(int b); 는 안 된다는 뜻
 - 매개 변수의 구성이 달라야 함



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

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

int main(){
    cout << add(1, 2) << "\n";        // 3
    cout << add(1, 2, 3) << "\n";      // 6
    cout << add(1.2, 3.4) << "\n";    // 4.6
    cout << add("12", "34") << "\n"; // 1234
}
```

함수 오버로딩

함수 오버로딩

- 매개변수가 정확히 일치하는 함수가 없으면 자동 형변환이 발생할 수도 있음
- 자동 형변환을 통해 가능한 함수가 여러 가지이면 컴파일 오류 - f5(1LL) 참고



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

void f1(int a){ cout << "f1(int a)\n"; }
void f1(long long a){ cout << "f1(long long a)\n"; }
void f2(int a){ cout << "f2(int a)\n"; }
void f3(long long a){ cout << "f3(long long a)\n"; }
void f4(double a){ cout << "f4(double a)\n"; }
void f5(int a){ cout << "f5(int a)\n"; }
void f5(double a){ cout << "f5(double a)\n"; }

int main(){
    f1(1); f1(1LL); // int, long long -> int, long long
    f2(1); f2(1LL); // int, long long -> int, int
    f3(1); f3(1LL); // int, long long -> long long, long long
    f4(1); f4(1LL); // int, long long -> double, double
    f5(1); f5(1LL); // int, long long -> int, compile-error
}
```

bits/stdc++.h

bits/stdc++.h

bits/stdc++.h

- GCC에서 제공해 주는 헤더 파일
 - 언어 표준에 있는 헤더 파일은 아님 → 환경에 따라(Visual C++ 등) 사용하지 못할 수도 있음
- 대부분의 표준 라이브러리를 포함하고 있는 헤더 파일
 - https://gcc.gnu.org/onlinedocs/gcc-13.1.0/libstdc++/api/a00839_source.html 참고
- 장점
 - 코드 타이핑 시간을 줄일 수 있음
 - 각 함수/클래스가 어떤 헤더 파일에 있는지 신경쓰지 않아도 됨
- 단점
 - GCC로 컴파일할 때만 사용할 수 있음
 - bits/stdc++.h 안에 들어있는 모든 헤더 파일을 컴파일해야 하므로 컴파일이 오래 걸림
 - <https://stackoverflow.com/questions/31816095/why-should-i-not-include-bits-stdc-h>
- 채점 서버는 대부분 GCC, 컴파일은 길어야 5초 이내이므로 알고리즘 문제 풀 때는 사용해도 상관 없음
 - 일단 난 지금까지 문제 된 적 없었음

질문?