

# 천하제일 C++ 최적화 퀴즈대회

**주준량**

Email : junryoungju@gmail.com

Github : <https://github.com/artblnd>

1. 자개소개
2. 발표를 듣기 전에...
3. 문제 시간
4. 정리
5. 질문 시간

목차

1. 문제 제시
2. 정답 맞추기
3. 성능 비교
4. 자세한 설명
5. 정리

여기서 문제 시간의 경우에는....

- **Contributions**

- Compiler Infrastructure / **LLVM**

- Scalar Optimization.

- Instruction Combine.

- Compiler Infrastructure / **CoreCLR**(.NET Core Runtime)

- Loop Optimization.

- Bound Check Optimization.

자기소개

# 퍼포먼스 측정 방법

컴파일러 : CL (Visual C++ Compiler)

측정방식 : STL chrono 사용

발표를 듣기 전에

## 사용한 아키텍처

CPU : Ryzen 1600x (6 core, 12 threads)

RAM : 16GB (8 x 2)

OS : Windows 10 (x64, 18362)

발표를 듣기 전에

예시와 코드 설명 방식!

발표를 듣기 전에

```
int Func()  
{  
    // 샘플 코드를 실행해주는 코드!  
    FuncImpl();  
}
```

난이도 X / 실행 코드



```
void FuncImpl()  
{  
    // 차이가 나는 코드!  
}
```

난이도 X / 샘플 1

# 발표의 방향성

문제는 샘플 1과 2로 주어집니다!

어떤 샘플이 더 빠른지, 왜 그런지 설명해주세요!

발표를 듣기 전에

```
int main()
{
    std::vector<int> integers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int total = 0;
    for (unsigned int i = 0; i < 10; ++i) {
        total += AddMul(integers, i);
    }
    return total;
}
```

01번 문제 (난이도 0) / 실행 코드

```
int AddMul(vector<int> values, int idx)
{
    int total = 0;

    for (unsigned int i = 0; i < values.size(); ++i)
    {
        total = values[i] + (idx * 10);
    }

    return total;
}
```

01번 문제 (난이도 0) / 샘플 1

```
int AddMul(vector<int>& values, int idx)
{
    int total = 0;

    for (unsigned int i = 0; i < values.size(); ++i)
    {
        total = values[i] + (idx * 10);
    }

    return total;
}
```

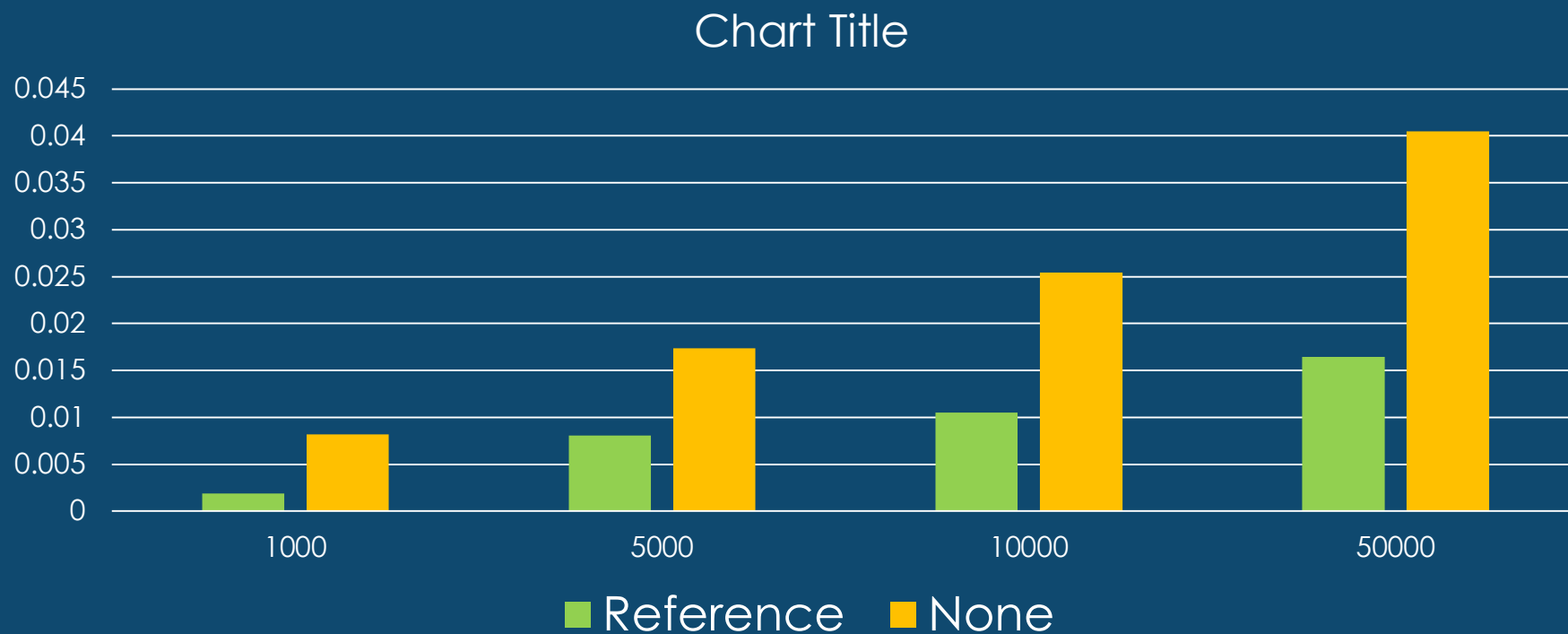
01번 문제 (난이도 0) / 샘플 2

```
int AddMul(vector<int>& values, int idx)
{
    int total = 0;
    for (unsigned int i = 0; i < values.size(); ++i)
    {
        total = values[i] + (idx * 10);
    }

    return total;
}
```

& operator가 존재!

01번 문제 (난이도 0) / 차이점



01 번 문제 (난이도 0) / 성능 차이

그전에 &은 무엇을 의미할까? C++ 레퍼런스에서는...

## Reference initialization

Binds a reference to an object

### Syntax

```
T & ref = object ;  
T & ref = { arg1, arg2, ... } ;  
T & ref ( object ) ;  
T & ref { arg1, arg2, ... } ;
```

(1)

01번 문제 (난이도 0) / 어떻게 작동할까?





## 네이버에 의하면....

[U] (정보를 얻기 위해) 찾아봄, 참고, 참조

Keep the list of numbers near the phone for easy **reference**.  

쉽게 찾아볼 수 있도록 전화번호 목록을 전화기 옆에 두도록 하라.

I wrote down the name of the hotel **for future reference**.  

나는 다음에 참고하기 위해 그 호텔 이름을 적어 놓았다.

The library contains many popular **works of reference**.  

그 도서관에는 인기 있는 참고 문헌들이 많다.

## 01번 문제 (난이도 0) / 그럼 레퍼런스란?

```
lea    r14,[rsp+0x8]  
mov     rdi,r14  
call   AddMul(std::vector<int>&, int)  
add     ebx,eax
```

01번 문제 (난이도 0) / 실제로 까보자!

```
lea    r14,[rsp+0x8]  
mov     rdi,r14  
call   AddMul(std::vector<int>&, int)  
add     ebx,eax
```


vector 오브젝트의 위치(reference)를 전달!

01번 문제 (난이도 0) / 실제로 까보자!

```
lea    r15,[rsp+0x8]
mov     rsi,r15
call    std::vector<int>::vector(std::vector<int> const&)
mov     rdi,r14
call    AddMul(std::vector<int>, int)
add     ebx,eax
```

01번 문제 (난이도 0) / 실제로 까보자!

```
lea    r15,[rsp+0x8]
mov     rsi,r15
call    std::vector<int>::vector(std::vector<int> const&)
mov     rdi,r14
call    AddMul(std::vector<int>, int)
add     ebx,eax
```



constructor를 호출해vector 오브젝트를 복사

01번 문제 (난이도 0) / 실제로 까보자

& 기호를 사용하면 객체의 위치를 전달하여  
필요없는 객체의 복사  
그리고 객체 안의 원소들의 복사를 줄이고  
그것으로 인해 성능이 향상된다!

01번 문제 (난이도 0)

```
int main()
{
    vector<string> names = { "Shereen Philip",
        "Bella-Rose Traynor", "Soren Harper",
        "Crystal Robles", "Tommie Cummings",
        "Reagan Clay", "Eleasha Harris", "Alexandria Cooper",
        "Warwick Oneill", "Lizzie Black" };

    string AllName = GetAllName(names);
}
```

02번 문제(난이도 1) / 실행 코드

```
string GetAllNameImpl(vector<string> names)
{
    string allName = "";
    for (string s : names)
        allName += s;
    return allName;
}

string GetAllName(vector<string>& names)
{
    return GetAllNameImpl(names);
}
```

02번 문제 (난이도 1) / 샘플 1



```
string GetAllNameImpl(vector<string> names)
{
    string allName = "";
    for (string s : names)
        allName += s;
    return allName;
}


string GetAllName(vector<string>& names)
{
    return GetAllNameImpl(std::move(names));
}
```

02번 문제 (난이도 1) / 샘플 2

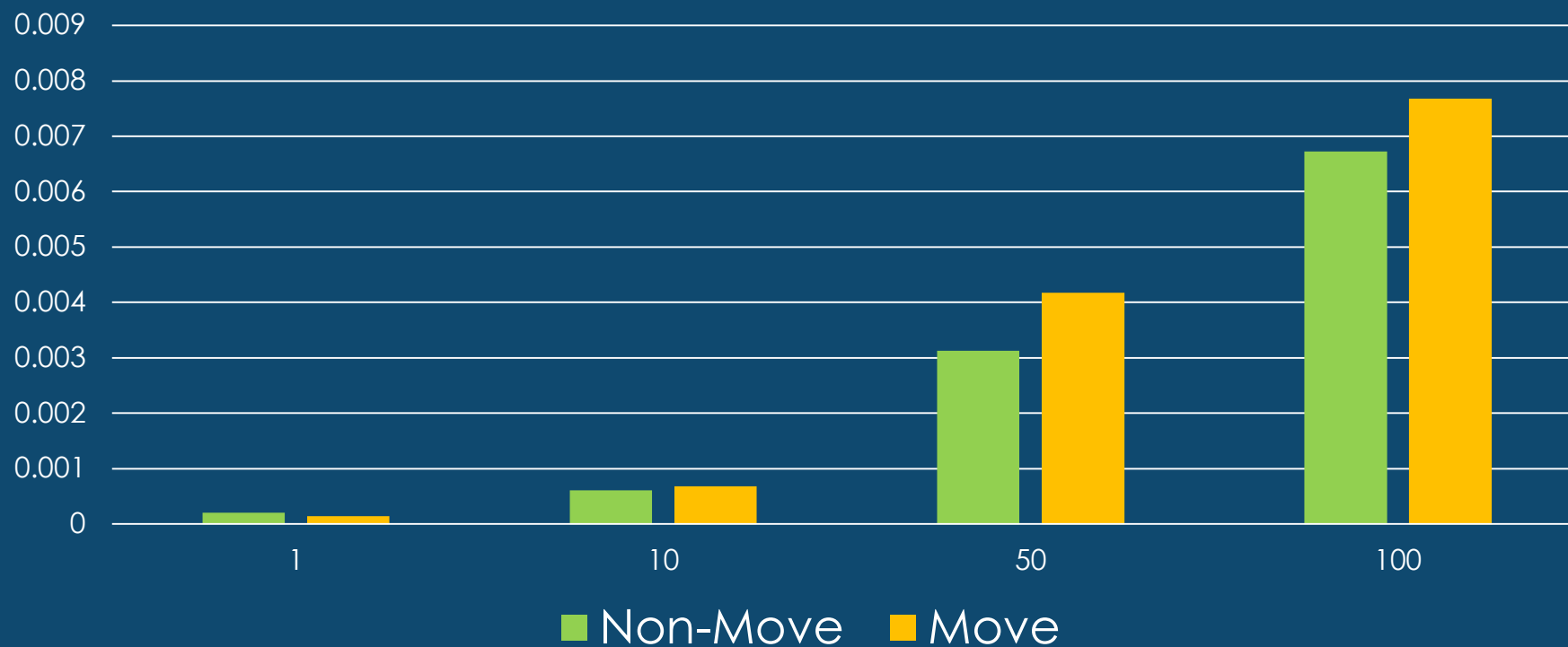
```
string GetAllNameImpl(vector<string> names)
{
    string allName = "";
    for (string s : names)
        allName += s;
    return allName;
}

string GetAllName(vector<string>& names)
{
    return GetAllNameImpl(std::move(names));
}
```

std::move를 사용



02번 문제 (난이도 1) / 차이점



02번 문제 (난이도 1) / 성능 차이

```
// FUNCTION TEMPLATE move
template <class _Ty>
constexpr
remove_reference_t<_Ty>&&
move(_Ty& _Arg) noexcept { // forward _Arg as movable
    return static_cast<remove_reference_t<_Ty>&&>(_Arg);
}
```

\_Ty이 vector<string>이라고 가정하고 다시 보자!

02번 문제 (난이도 1) / MOVE가 뭘까?

```
// FUNCTION TEMPLATE move
constexpr
vector<string>&&
move(vector<string>& _Arg) noexcept {
    // forward _Arg as movable
    return (vector<string>&&)(_Arg);
}
```

02번 문제 (난이도 1) / MOVE가 뭘까?

```
// FUNCTION TEMPLATE move
constexpr      Args를 vector<string>&&으로 캐스팅하고 있다!
vector<string>&&
move(vector<string>& _Arg) noexcept {
    // forward Arg as movable
    return (vector<string>&&)(_Arg);
}
```

02번 문제 (난이도 1) / MOVE가 뭘까?

```
vector(const vector& x)
    : __base(__alloc_traits::select_on_container_copy_construction(
        x.__alloc()))
{
    size_type n = x.size();
    if (n > 0)
    {
        __vallocate(n);
        __construct_at_end(x.__begin_, x.__end_, n);
    }
}
```

02번 문제 (난이도 1) / VECTOR를 보자!

```
vector(const vector& x)
: __base(__alloc_traits::select_on_container_copy_construction(
    x.__alloc()))
{
    vector& x의 크기만큼 할당하고 element를 복사한다!
    size_type n = x.size();
    if (n > 0)
    {
        __vallocate(n);
        __construct_at_end(x.__begin_, x.__end_, n);
    }
}
```

02번 문제 (난이도 1) / VECTOR를 보자!



const vector& x

this

pData[0]

pData[1]

...

pData[n]

Construct Copy

pData[0]

pData[1]

...

pData[n]

02번 문제 (난이도 1) / VECTOR를 보자!

```
vector(vector&& x)
: __base(_VSTD::move(x.__alloc()))
{
    this->__begin_      = x.__begin_;
    this->__end_         = x.__end_;
    this->__end_cap()    = x.__end_cap();

    x.__begin_ = x.__end_ = x.__end_cap() = nullptr;
}
```

02번 문제 (난이도 1) / VECTOR를 보자!

```
vector(vector&& x)
: __base(_VSTD::move(x.__alloc()))
{
    this->__begin_ = x.__begin_;
    this->__end_ = x.__end_;
    this->__end_cap() = x.__end_cap();

    x.__begin_ = x.__end_ = x.__end_cap() = nullptr;
}
```

vector&&로 캐스팅 했기에 이constructor가 작동된다!

02번 문제 (난이도 1) / VECTOR를 보자!

```
vector(vector&& x)
: __base(_VSTD::move(x.__alloc()))
{
    this->__begin_      = x.__begin_;
    this->__end_         = x.__end_;
    this->__end_cap()    = x.__end_cap();
    x.__begin_ = x.__end_ = x.__end_cap() = nullptr;
}
```

포인터를 그대로 가져오고 x를 초기화한다!

02번 문제 (난이도 1) / VECTOR를 보자!

const vector&& x

this

p a

Move

pData

02번 문제 (난이도 1) / VECTOR를 보자!

객체를 더 이상 현재 코드에서 사용하지 않고  
다른 곳으로 전달한다면  
`std::move`를 사용해서 객체가 복사되는 것을 방지하자!

02번 문제 (난이도 1) / 결론

```
int main()
{
    list<string> strs = {
        "All", "Of", "Us", "Are",
        "C++", "Programmer", "!"};

    string mergedString = MergeString(strs);
}
```

03번 문제 (난이도 1) / 실행 코드

```
string MergeString(list<string>& strs)
{
    string merged = "";

    for (auto it = strs.begin(); it != strs.end(); ++it)
    {
        merged += *it;
    }
    return merged;
}
```

03번 문제 (난이도 1) / 샘플 1



```
string MergeString(list<string>& strs)
{
    string merged = "";

    for (auto it = strs.begin(); it != strs.end(); it++)
    {
        merged += *it;
    }
    return merged;
}
```

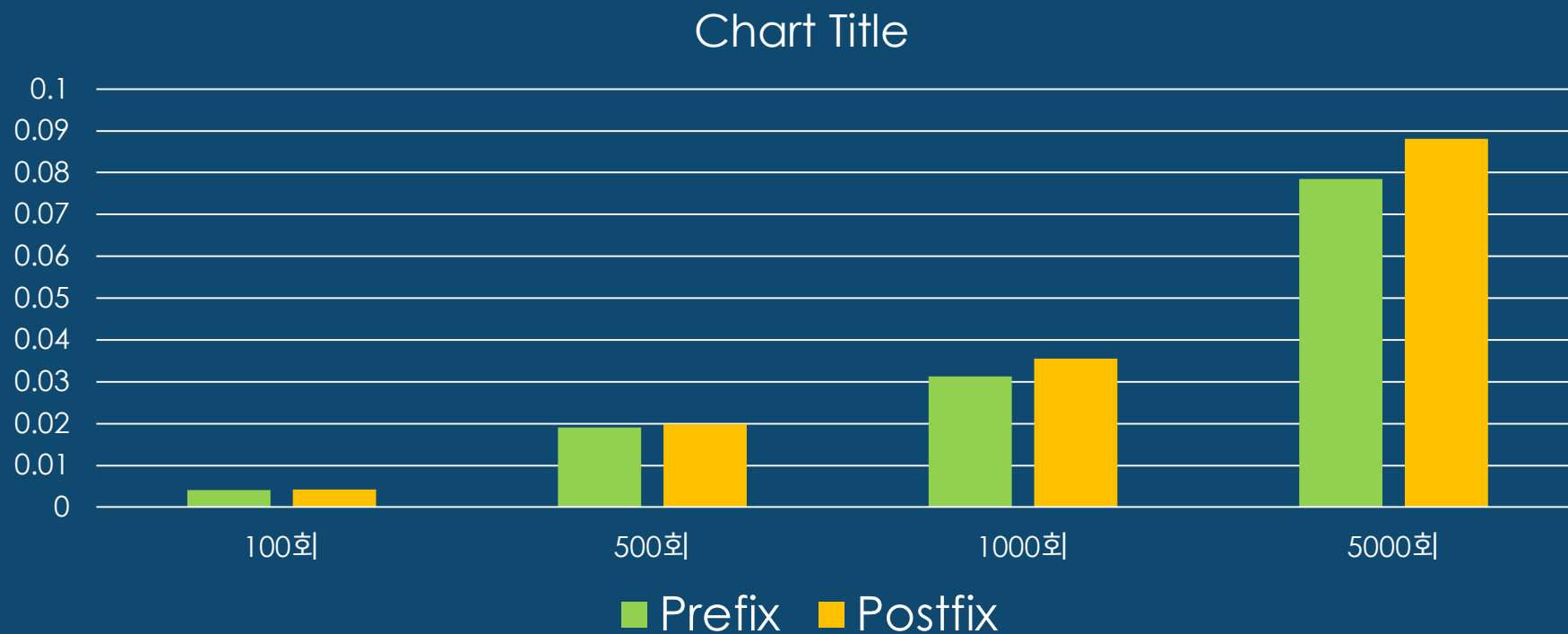
03번 문제 (난이도 1) / 샘플 2

```
string MergeString(list<string>& strs)
{
    string merged = "";

    for (auto it = strs.begin(); it != strs.end(); it++)
    {
        merged += *it;
    }
    return merged;
}
```

++ operator의 위치가 다르다!

03번 문제 (난이도 1) / 샘플 2



03번 문제 (난이도 2) / 성능 차이

```
_NODISCARD iterator begin() noexcept
{ // return iterator for beginning of mutable sequence
    return iterator(_Myhead()->_Next, _STD addressof(_Get_data()));
}

_NODISCARD iterator end() noexcept
{ // return iterator for end of mutable sequence
    return iterator(_Myhead(), _STD addressof(_Get_data()));
}
```

03번 문제 (난이도 2) / LIST를 까 보자!

iterator를 만든다!

```
_NODISCARD iterator begin() noexcept  
{ // return iterator for beginning of mutable sequence  
    return iterator(_Myhead()->_Next, _STD addressof(_Get_data()));  
}  
  
_NODISCARD iterator end() noexcept  
{ // return iterator for end of mutable sequence  
    return iterator(_Myhead(), _STD addressof(_Get_data()));  
}
```

그럼 iterator는 어떻게 되어있을까?

03번 문제 (난이도 2) / LIST를 까 보자!

++X와 X++의 차이점이 뭘까요?

++X



1을 더하고 X를 준다!

X++



X를 주고 1을 더한다!

03번 문제 (난이도2) / 그 전에!

```
_List_iterator& operator++() {  
    _Mybase::operator++();  
    return *this;  
}  
  
_List_iterator operator++(int) {  
    _List_iterator _Tmp = *this;  
    _Mybase::operator++();  
    return _Tmp;  
}
```

03번 문제 (난이도 2) / ITERATOR를 까 보자!

```
_List_iterator& operator++() {  
    _Mybase::operator++();  
    return *this;  
}
```

```
_List_iterator operator++(int) {  
    _List_iterator _Tmp = *this;  
    _Mybase::operator++();  
    return _Tmp;  
}
```

Int가 더 붙어있다?

03번 문제 (난이도 2) / ITERATOR를 까 보자!



## C++ Reference에서는....

When the postfix increment and decrement appear in an expression, the corresponding user-defined function (operator++ or operator--) is called with an integer argument 0. Typically, it is implemented as T operator++(int), where the argument is ignored. The postfix increment and decrement operator is usually implemented in terms of the prefix version

03번 문제 (난이도 2) / ITERATOR를 까 보자

## C++ Reference에서는....

When the **postfix increment and decrement** appear in an expression, the corresponding user-defined function (`operator++` or `operator--`) is **called with an integer argument 0**. Typically, it is implemented as `T operator++(int)`, where the argument is ignored. The postfix increment and decrement operator is usually implemented in terms of the prefix version

03번 문제 (난이도 2) / ITERATOR를 까 보자

```
_List_iterator& operator++() {  
    _Mybase::operator++();  
    return *this;  
}
```

Prefix Increment (++x)

```
_List_iterator operator++(int) {  
    _List_iterator _Tmp = *this;  
    _Mybase::operator++();  
    return _Tmp;  
}
```

Postfix Increment (x++)

03번 문제 (난이도 2) / ITERATOR를 까 보자!

```
_List iterator& operator++() {
```

```
    _Mybase::operator++();  
    return *this;
```

← 자기 자신에 1을 더하고 리턴

```
}
```

리턴하기 전에 자기 자신을 복사하고 리턴

```
_List iterator operator++(int) {
```

```
    _List_iterator _Tmp = *this;  
    _Mybase::operator++();  
    return _Tmp;
```

```
}
```

03번 문제 (난이도 2) / ITERATOR를 까 보자!

```
_List_iterator& operator++() {  
    _Mybase::operator++();  
    return *this;  
}  
  
_List_iterator operator++(int) {  
    _List_iterator _Tmp = *this;  
    _Mybase::operator++();  
    return _Tmp;  
}
```

원치 않게 Iterator가 한번 더 생성됨!

03번 문제 (난이도 2) / ITERATOR를 까 보자!

가능하다면 prefix increment(++x)를 사용하여  
iterator가 한번 더 복사되는 것을 방지하자!

03번 문제 (난이도 2) / 결론

```
int main()
{
    auto result = Add
    (
        {1, 4, 5, 2, 3, 5, 1, 6, 9, 4, 3, 4, 2},
        {6, 7, 2, 6, 4, 2, 9, 9, 5, 3, 1, 5, 1}
    );
}
```

04번 문제 (난이도 3) / 실행 코드

```
auto Add(vector<unsigned int> a, vector<unsigned int> b)
{
    vector<unsigned int> result;

    auto itA = a.begin(), itB = b.begin();
    while (itA != a.end() && itB != b.end())
    {
        result.push_back(*itA + *itB);
        ++itA; ++itB;
    }
    return result;
}
```

04번 문제 (난이도 3) / 샘플 1



```
auto Add(list<unsigned int> a, list<unsigned int> b)
{
    list<unsigned int> result;

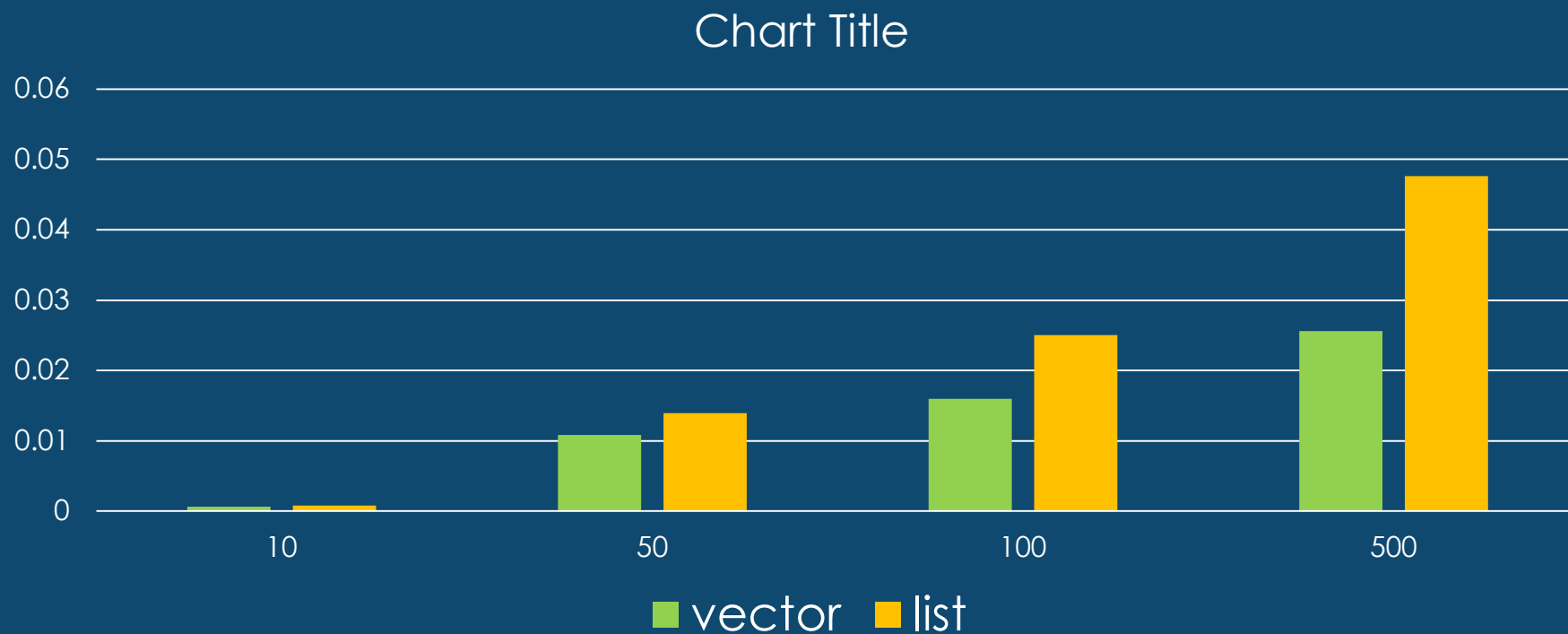
    auto itA = a.begin(), itB = b.begin();
    while (itA != a.end() && itB != b.end())
    {
        result.push_back(*itA + *itB);
        ++itA; ++itB;
    }
    return result;
}
```

04번 문제 (난이도 3) / 샘플 2

```
auto Add(vector<unsigned int> a, vector<unsigned int> b)
{
    vector<unsigned int> result;

    auto itA = a.begin(), itB = b.begin();
    while (itA != a.end() && itB != b.end())
    {
        result.push_back(*itA + *itB);
        ++itA; ++itB;
    }
    return result;
}
```

04번 문제 (난이도 3) / 샘플 1



04번 문제 (난이도 3) / 성능 차이

## C++ Reference에서 std::list란....

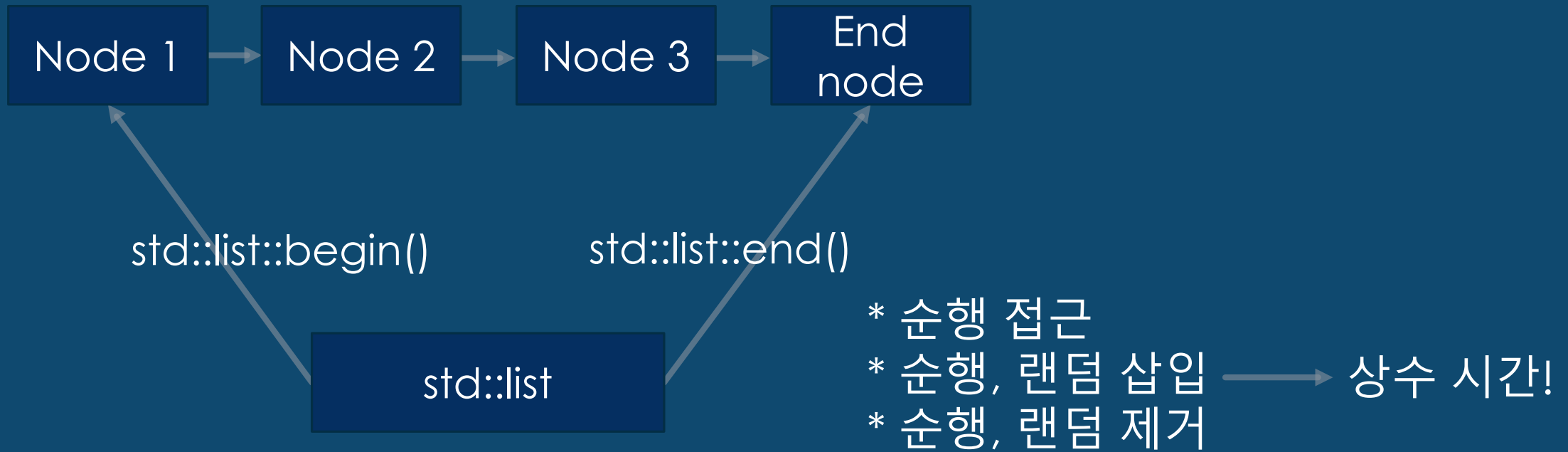
std::list is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list. Compared to std::forward\_list this container provides bidirectional iteration capability while being less space efficient.

04번 문제 (난이도 3) / 특징을 보자

## C++ Reference에서 std::list란....

std::list is a container that supports **constant time insertion and removal** of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list. Compared to std::forward\_list this container provides bidirectional iteration capability while being less space efficient.

04번 문제 (난이도 3) / 특징을 보자



04번 문제 (난이도 3) / 특징을 보자

# C++ Reference에서 std::vector란....

std::vector is a sequence container that encapsulates dynamic size arrays.

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

## 04번 문제 (난이도 3) / 특징을 보자

# C++ Reference에서 std::vector란....

std::vector is a sequence container that encapsulates dynamic size arrays.

The **elements are stored contiguously**, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

## 04번 문제 (난이도 3) / 특징을 보자



Element	Element	Element	Element
1	2	3	4

`std::vector::begin()`  
`std::vector::data()`

`std::list::end()`

`std::vector`

\* 순행, 랜덤 접근  
\* 순행 삽입, 제거 → 상수 시간!

04번 문제 (난이도 3) / 특징을 보자

둘 다 순행 접근과 삽입이 상수 시간인데?

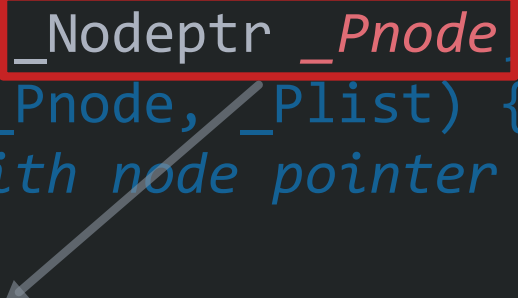
04번 문제 (난이도 3) / 특징을 보자

```
_List_iterator(_Nodeptr _Pnode, const _Mylist* _Plist)  
    : _Mybase(_Pnode, _Plist) {  
    // construct with node pointer _Pnode  
}
```

04번 문제 (난이도 3) / LIST를 들여다보자!

```
_List_iterator(_Nodeptr Pnode, const _Mylist* _Plist)
    : _Mybase(_Pnode, _Plist) {
// construct with node pointer _Pnode
}

struct _List_node { // list node
    using value_type = _Value_type;
    using _Nodeptr    = _Rebind_pointer_t<_Voidptr, _List_node>;
    _Nodeptr _Next; // successor node, or first element if head
    _Nodeptr _Prev; // predecessor node, or last element if head
    _Value_type _Myval; // the stored value, unused if head
    ... }
```



04번 문제 (난이도 3) / LIST를 들여다보자!

```

_List_iterator(_Nodeptr _Pnode, const _Mylist* _Plist)
    : _Mybase(_Pnode, _Plist) {
// construct with node pointer _Pnode
}

struct _List_node { // List node
    using value_type = _Value_type;
    using _Nodeptr = _Rebind_pointer_t<_Voidptr, _List_node>;
    _Nodeptr _Next; // successor node, or first element if head
    _Nodeptr _Prev; // predecessor node, or last element if head
    _Value_type _Myval; // the stored value, unused if head
    ...
}

```

다음 노드의 위치

이전 노드의 위치

현재 노드가 가지고있는 값

04번 문제 (난이도 3) / LIST를 들여다보자!

```
_List_const_iterator& operator++() {  
    this->_Ptr = this->_Ptr->_Next;  
    return *this;  
}  
  
_NODISCARD reference operator*() const {  
    return _Ptr->_Myval;  
}
```

04번 문제 (난이도 3) / LIST를 들여다보자!

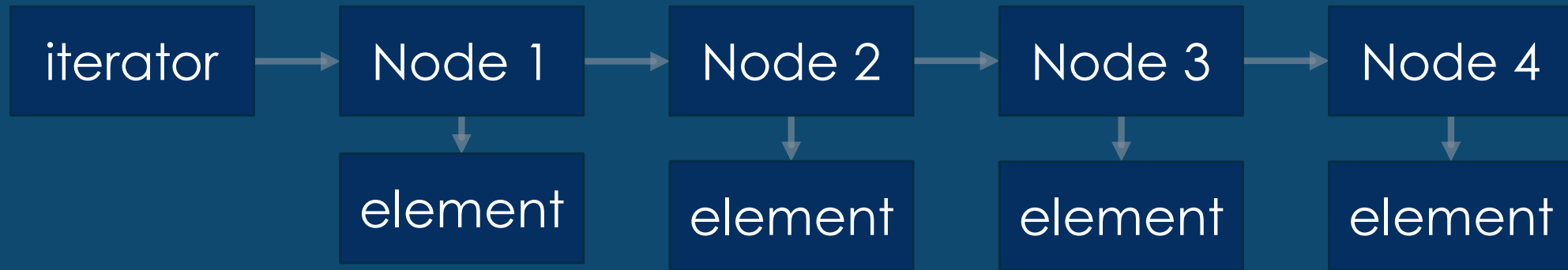
```
_List const iterator& operator++() {  
    this->_Ptr = this->_Ptr->_Next;  
    return *this;  
}  
  
_NODISCARD reference operator*() const {  
    return _Ptr->_Myval;  
}
```

다음 노드로 노드 위치를 바꿈

노드에 있는 값

04번 문제 (난이도 3) / LIST를 들여다보자!

순행 접근이 상수 시간이지만, 실제로는 이렇게 접근!



Iterator에서 추가적인 접근이 필요하다!

04번 문제 (난이도 3) / 그래서?



```
_Vector_iterator(pointer _Parg, const _Container_base* _Pvector)  
    : _Mybase(_Parg, _Pvector)  
{  
}
```


04번 문제 (난이도 3) / VECTOR를 들여다보자!

```
_Vector_iterator(pointer _Parg, const _Container_base* _Pvector)
    : _Mybase(_Parg, _Pvector)
{
}

using _Tptr = typename _Myvec::pointer;


_Vector_const_iterator(
    _Tptr _Parg, const _Container_base* _Pvector)
    : _Ptr(_Parg) { this->_Adopt(_Pvector); }


_Tptr _Ptr; // pointer to element in vector
```



04번 문제 (난이도 3) / VECTOR를 들여다보자!

```
_Vector_iterator(pointer _Parg, const _Container_base* _Pvector)
    : _Mybase(_Parg, _Pvector)
{
}

using _Tptr = typename _Myvec::pointer;
_Vector_const_iterator(  현재 값으로 바로 통하는 포인터
    _Tptr _Parg, const _Container_base* _Pvector)
    : _Ptr(_Parg) { this->_Adopt(_Pvector); }

 _Tptr _Ptr; // pointer to element in vector
```

04번 문제 (난이도 3) / VECTOR를 들여다보자!

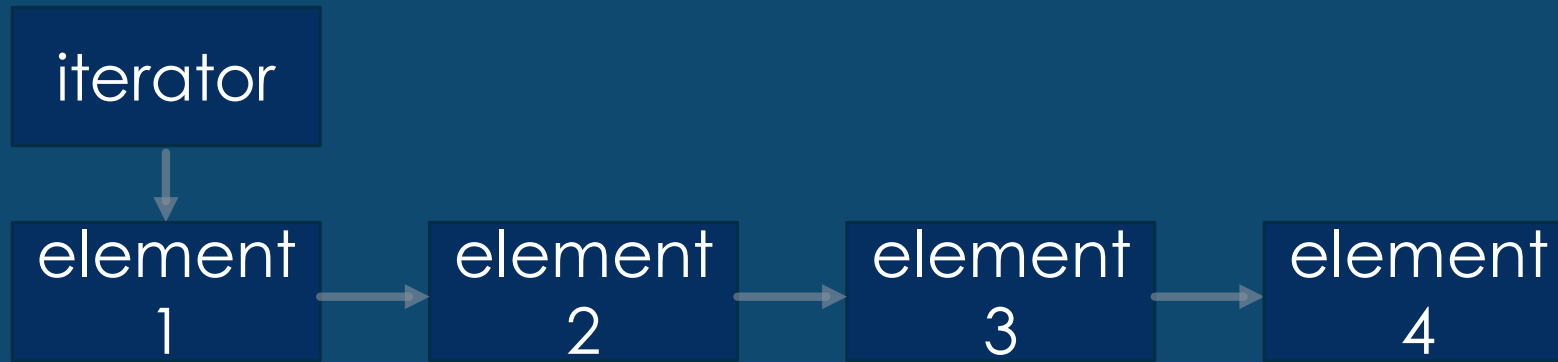
```
_Vector_const_iterator& operator++() {  
    ++_Ptr;  
    return *this;  
}  
  
_NODISCARD reference operator*() const {  
    return *_Ptr;  
}
```

04번 문제 (난이도 3) / VECTOR를 들여다보자!

```
_Vector_const_iterator& operator++() {  
    ++_Ptr; ← 포인터의 위치를 바꿈!  
    return *this;  
}  
  
_NODISCARD reference operator*() const {  
    return *_Ptr; ← 현재 포인터의 위치에 있는 값을 반환!  
}
```

04번 문제 (난이도 3) / VECTOR를 들여다보자!

비슷해 보이지만 노드를 거치지 않고 접근!



List의 iterator와 달리 추가적인 접근이 필요하지 않다!

04번 문제 (난이도 3) / 그래서?

랜덤 삽입을 정말로 많이 이용하게 아니라면  
list 대신 vector를 애용하자!  
똑같은 상수 시간이라고 해도 다르다!

04번 문제 (난이도 3) / 결론

```
int func(int v)
{
    int result = 0;
    for (unsigned int i = 1; i != v; ++i)
    {
        result = pow2(i);
    }

    return result;
}
```

05번 문제 (난이도 3) / 실행 코드



```
int pow2(int v)
{
    return v * v;
}
```

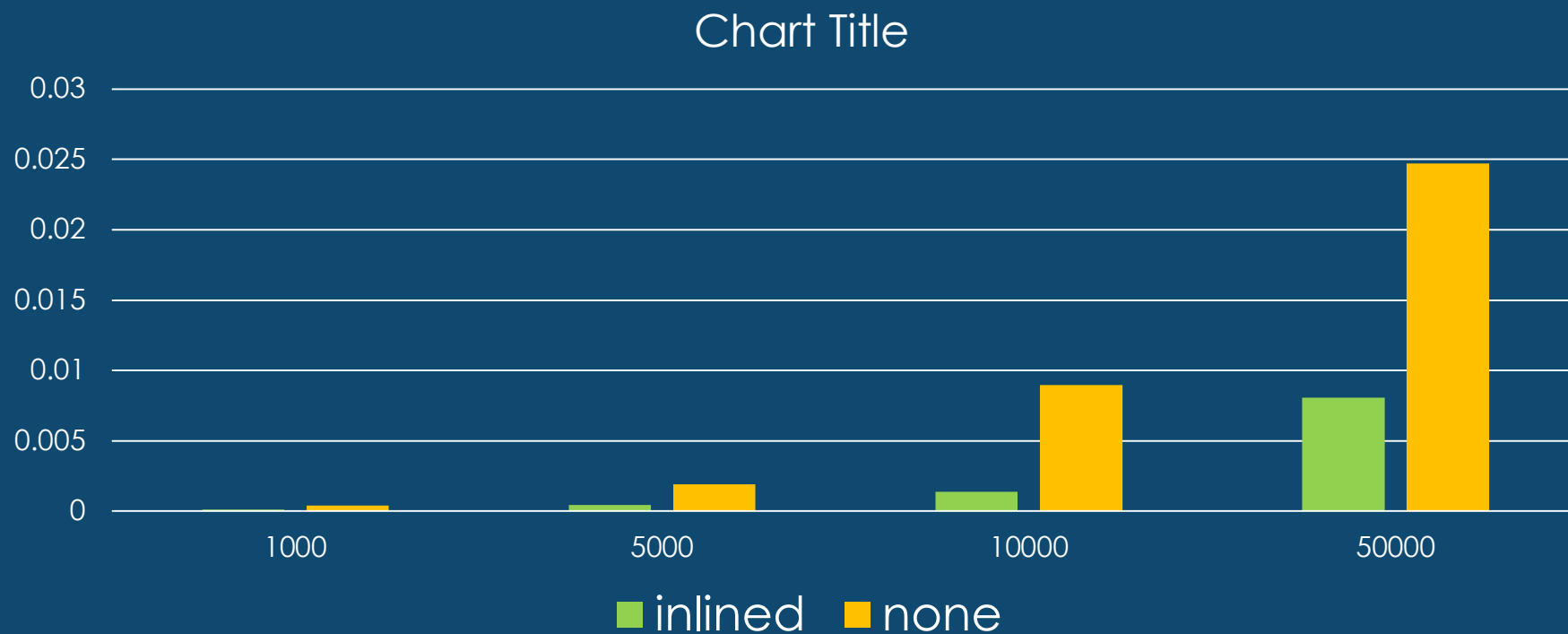
05번 문제 (난이도 3) / 샘플 1

```
inline int pow2_1(int v)
{
    return v * v;
}
```

05번 문제 (난이도 3) / 샘플 2

```
inline int pow2_1(int v)
{
    return v * v;
}
```

05번 문제 (난이도 3) / 차이점



05번 문제 (난이도 3) / 성능 차이

## C++ 레퍼런스에서 inline은....

The original intent of the inline keyword was to serve as an indicator to the optimizer that inline substitution of a function is preferred over function call, that is, instead of executing the function call CPU instruction to transfer control to the function body, a copy of the function body is executed without generating the call. This avoids overhead created by the function call (passing the arguments and retrieving the result) but it may result in a larger executable as the code for the function has to be repeated multiple times.

05번 문제 (난이도 3) / 어떻게 작동할까?

## C++ 레퍼런스에서 inline은....

The original intent of the inline keyword was to serve as an indicator to the optimizer that inline substitution of a function is preferred over function call, that is, instead of executing the function call CPU instruction to transfer control to the function body, **a copy of the function body is executed without generating the call**. This avoids overhead created by the function call (passing the arguments and retrieving the result) but it may result in a larger executable as the code for the function has to be repeated multiple times.

05번 문제 (난이도 3) / 어떻게 작동할까?

```
int func(int v)
{
    int result = 0;
    for (unsigned int i = 1; i != v; ++i)
    {
        result = pow2(i);
    }
    return result;
}
```

05번 문제 (난이도 3) / 어떻게 작동할까?

```
int func(int v)
{
    int result = 0;
    for (unsigned int i = 1; i != v; ++i)
    {
        result = i * i;
    }
    return result;
}
```

05번 문제 (난이도 3) / 어떻게 작동할까?



```
int func(int v)
{
    int result = 0;
    for (unsigned int i = 1; i != v; ++i)
    {
        result = i * i;
    }
    return ret;
}
```

pow2함수가 왔던 자리가  $i * i$  대체된다!

05번 문제 (난이도 3) / 어떻게 작동할까?

하지만 정말로 그것만 그렇게 작동할까?

05번 문제 (난이도 3) / 어떻게 작동할까?

```
$LL4@func:  
    mov ecx, ebx  
    call int pow2(int)  
    add edi, eax  
    inc ebx  
    cmp ebx, esi  
    jb SHORT $LL4@func
```

05번 문제 (난이도 3) / 뜯어보자!

```
$LL4@func:
```

```
mov ecx, ebx
```

```
call int pow2(int)
```

```
add edi, eax
```

```
inc ebx
```

```
cmp ebx, esi
```

```
jb SHORT $LL4@func
```

← 평범하게 pow2를 호출하고있다!

05번 문제 (난이도 3) / 뜯어보자!

```
$LL4@func:  
    movd xmm0, edx  
    lea eax, DWORD PTR [rdx+4]  
    pshufd xmm0, xmm0, 0  
    add edx, 8  
    paddb xmm0, xmm3  
    ...  
    pmulld xmm0, xmm0  
    paddb xmm0, xmm1  
    movdqa xmm1, xmm0  
    cmp edx, ecx  
    jb SHORT $LL4@func
```

05번 문제 (난이도 3) / 뜯어보자!

```
$LL4@func:
```

```
movd xmm0, edx
```

```
lea eax, DWORD PTR [rdx+4]
```

```
pshufd xmm0, xmm0, 0
```

```
add edx, 8
```

```
padd xmm0, xmm3
```

```
...
```

```
pmulld xmm0, xmm0
```

```
padd xmm0, xmm1
```

```
movdqa xmm1, xmm0
```

```
cmp edx, ecx
```

```
jb SHORT $LL4@func
```

???????????



05번 문제 (난이도 3) / 뜯어보자!

```
$LL4@func:
```

```
mov ecx, ebx  
call int pow2(int)
```

```
add edi, eax  
inc ebx  
cmp ebx, esi  
jb SHORT $LL4@func
```

```
int pow2(int) PROC
```

```
imul ecx, ecx  
mov eax, ecx
```

```
int pow2(int) ENDP
```

05번 문제 (난이도 3) / 뜯어보자!

```
$LL4@func:
```

```
mov ecx, ebx
```

```
imul ecx, ecx
```

```
mov eax, ecx
```

```
add edi, eax
```

```
inc ebx
```

```
cmp ebx, esi
```

```
jb SHORT $LL4@func
```

call 자리에 함수 body가!

05번 문제 (난이도 3) / 뜯어보자!



더 쉬운 예제를 보자!

05번 문제 (난이도 3) / 어떻게 작동할까?

```
inline int add_5(int v)
{
    return v + 5;
}

int func(int v)
{
    return add_5(5);
}
```

05번 문제 (난이도 3) / 어떻게 작동할까?

```
int func(int) PROC  
    mov ecx, 5  
    jmp int add_5(int)
```



```
int func(int) PROC  
    mov ecx, 5  
    add ecx, 5
```

05번 문제 (난이도 3) / 어떻게 작동할까?

```
int func(int) PROC  
    mov ecx, 5  
    jmp int add_5(int)
```



```
int func(int) PROC  
    mov ecx, 10
```

constant가 합쳐지는 최적화가 이루어짐!

05번 문제 (난이도 3) / 어떻게 작동할까?

Inline 키워드는 단순히 함수가 합쳐지는 것 뿐만이 아니라  
더 많은 최적화를 가능하게 해 준다!

05번 문제 (난이도 3) / 결론

```
int Func()
{
    vector<int> numbers = {10, 1000, 400, 302, 402, 1000, 30000, 400,
                          1000000, 1000000000, 1000, 20004, 500000, 100000, 32450};

    int ret = 0;
    for (int v : numbers)
    {
        ret += isPowOf10(v);
    }
    return ret;
}
```

06번 문제(난이도 4) / 실행 코드

```
bool isPowOf10(int v)
{
    return
        (v == 10) || (v == 100) || (v == 1000) ||
        (v == 10000) || (v == 100000) || (v == 1000000) ||
        (v == 10000000) || (v == 100000000) ||
        (v == 1000000000);
}
```

06번 문제(난이도 4) / 샘플 1

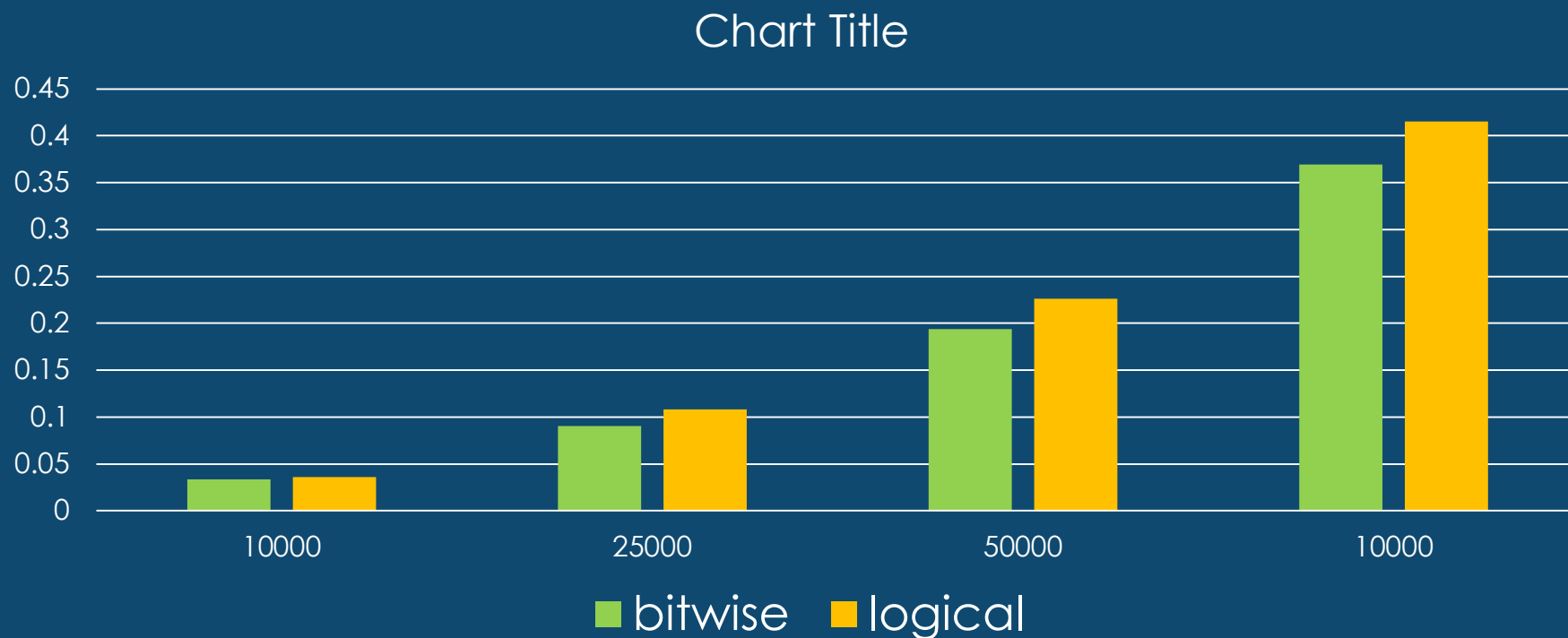
```
bool isPowOf10(int v)
{
    return
        (v == 10) | (v == 100) | (v == 1000) |
        (v == 10000) | (v == 100000) | (v == 1000000) |
        (v == 10000000) | (v == 100000000) |
        (v == 1000000000);
}
```

06번 문제(난이도 4) / 샘플 2



```
bool isPowOf10(int v)
{
    return
        (v == 10) || (v == 100) || (v == 1000) ||
        (v == 10000) || (v == 100000) || (v == 1000000) ||
        (v == 10000000) || (v == 100000000) ||
        (v == 1000000000);
}
```

06번 문제(난이도 4) / 차이



06번 문제 (난이도 4) / 성능 차이

# 둘다 OR 연산자다 하지만....

The bitwise inclusive OR operator (|) compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

## 06번 문제 (난이도 4) / 뭐가 다를까?

둘다 OR 연산자다 하지만....

하나가 1이면 전부 1!

The **bitwise** inclusive OR operator (|) compares each bit of its first operand to the corresponding bit of its second operand. **If either bit is 1, the corresponding result bit is set to 1.** Otherwise, the corresponding result bit is set to 0.

06번 문제 (난이도 4) / 뭐가 다를까?

## 둘다 OR 연산자다 하지만....

The logical OR operator ( `||` ) returns the boolean value `TRUE` if either or both operands is `TRUE` and returns `FALSE` otherwise. The operands are implicitly converted to type `bool` prior to evaluation, and the result is of type `bool`. Logical OR has left-to-right associativity.

06번 문제 (난이도 4) / 뭐가 다를까?

# 둘다 OR 연산자다 하지만....

The **logical** OR operator (`||`) returns the boolean value **TRUE** if **either or both operands is TRUE** and returns **FALSE** otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical OR has left-to-right associativity.

둘 중 하나가 TRUE라면 TRUE!

## 06번 문제 (난이도 4) / 뭐가 다를까?

동시에 일어나는 것과 순서가 있는 것이  
무슨 차이일까?

06번 문제 (난이도 4) / 실제로 본다면...

```
cmp edi,0x1869f
jle 40053c <isPowOf10(int)+0x2c>
cmp edi,0x98967f
jle 400550 <isPowOf10(int)+0x40>
cmp edi,0x989680
je 400572 <isPowOf10(int)+0x62>
cmp edi,0x5f5e100
je 400572 <isPowOf10(int)+0x62>
// 그리고 20줄 정도 더...
```

생각한대로 하나하나 비교한다!

06번 문제 (난이도 4) / 뜯어보자!



```
pcmpeqd xmm1,xmm0
pcmpeqd xmm0,XMMWORD PTR [rip+0xcd]
packssdw xmm0,xmm1
packsswb xmm0,xmm0
pmovmskb eax,xmm0
test al,al
setne al
```

06번 문제 (난이도 4) / 뜯어보자!

왜 이런 코드가 나왔을까?

06번 문제 (난이도 4) / 뜯어보자!

# 둘다 OR 연산자다 하지만....

The **logical** OR operator (`||`) returns the boolean value **TRUE** if **either or both operands is TRUE** and returns **FALSE** otherwise. The operands are implicitly converted to type **bool** prior to evaluation, and the result is of type **bool**. Logical OR has left-to-right associativity.

## 06번 문제 (난이도 4) / 뭐가 다를까?

# 둘다 OR 연산자다 하지만....

The **bitwise** inclusive OR operator ( | ) compares each bit of its first operand to the corresponding bit of its second operand. **If either bit is 1**, the corresponding result bit is **set to 1**. Otherwise, the corresponding result bit is set to 0.

## 06번 문제 (난이도 4) / 뭐가 다를까?

```
bool isPowOf10(int v)
{
    return
        (v == 10) || (v == 100) || (v == 1000) ||
        (v == 10000) || (v == 100000) || (v == 1000000) ||
        (v == 10000000) || (v == 100000000) ||
        (v == 1000000000);
}
```

06번 문제 (난이도 4) / 그렇다면?

```
bool isPowOf10(int v)
{
    return
        (v == 10) | (v == 100) | (v == 1000) |
        (v == 10000) | (v == 100000) | (v == 1000000) |
        (v == 10000000) | (v == 100000000) |
        (v == 1000000000);
}
```

06번 문제 (난이도 4) / 샘플 2

많은 양의 값을 동시에 확인하는 코드를 만들 때에는  
Logical 대신 bitwise를 사용하자!

06번 문제 (난이도 4) / 결론

```
int func()
{
    std::vector<int> data = MakeData();
    return data[0] + data[1] + data[2] + data[3];
}
```

07번 문제(난이도 5) / 실행 코드



```
vector<int> MakeData()  
{  
    vector<int> data;  
    data.push_back(1);  
    data.push_back(2);  
    data.push_back(3);  
    data.push_back(4);  
  
    return data;  
}
```

07번 문제(난이도 5) / 샘플 1

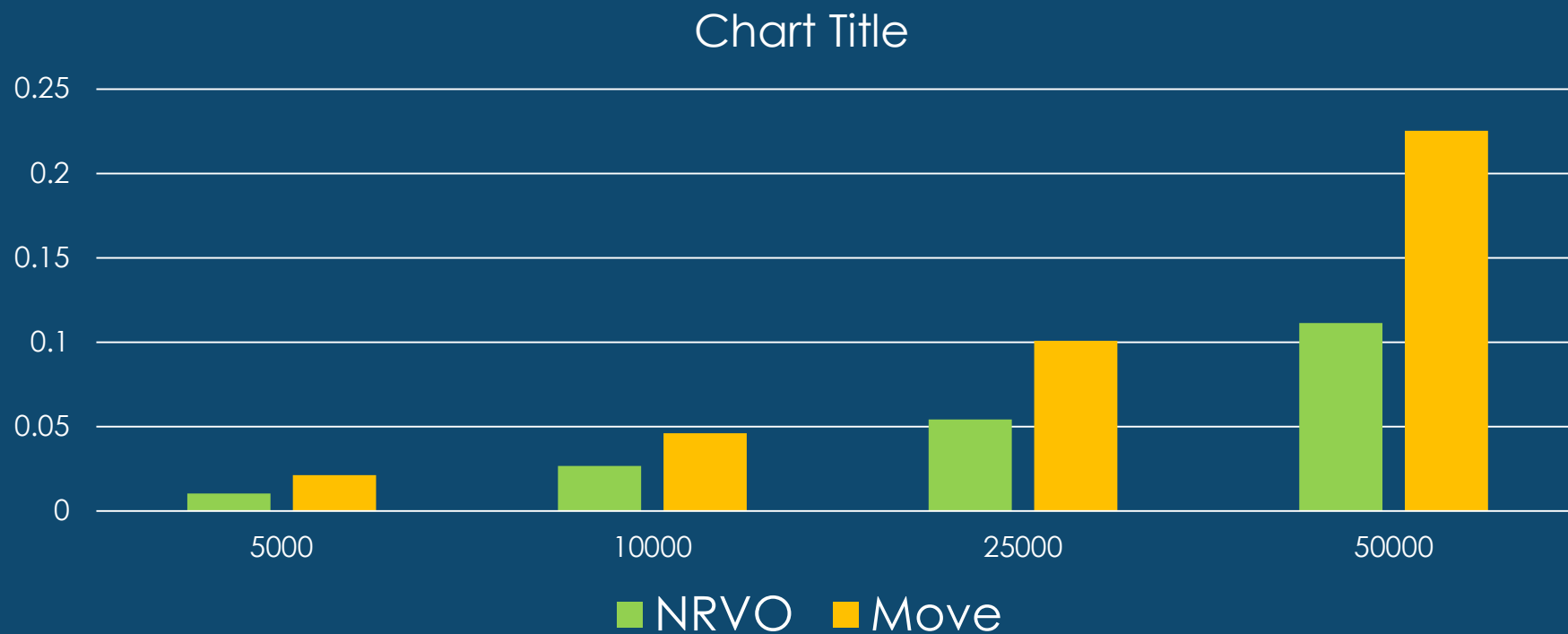
```
vector<int> MakeData()
{
    vector<int> data;
    data.push_back(1);
    data.push_back(2);
    data.push_back(3);
    data.push_back(4);

    return move(data);
}
```

07번 문제(난이도 5) / 샘플 2

```
vector<int> MakeData()  
{  
    vector<int> data;  
    data.push_back(1);  
    data.push_back(2);  
    data.push_back(3);  
    data.push_back(4);  
  
    return move(data);  
}
```

07번 문제 (난이도 5) / 비교



07번 문제 (난이도 5) / 성능 차이

아까전에는 `std::move`가 빠르다면서!!

07번 문제(난이도 5) / 왜 그럴까?

```
std::vector<int> MakeData()  
{  
    std::vector<int> data = std::vector<int>();  
    data.push_back(1);  
    data.push_back(2);  
    data.push_back(3);  
    data.push_back(4);  
    return std::vector<int>(std::move(data));  
}
```

07번 문제(난이도 5) / 뜯어보자!!

```
std::vector<int> MakeData()
{
    std::vector<int> data = std::vector<int>() /*NRVO variable*/;
    data.push_back(1);
    data.push_back(2);
    data.push_back(3);
    data.push_back(4);
    return std::vector<int>(data);
}
```

07번 문제(난이도 5) / 뜯어보자!!

```
std::vector<int> MakeData()
{
    std::vector<int> data = std::vector<int>() /*NRVO variable /;
    data.push_back(1);
    data.push_back(2);
    data.push_back(3);
    data.push_back(4);
    return std::vector<int>(data);
}
```

이상한 설명이 붙어있다?

07번 문제(난이도 5) / 뜯어보자!!



# NRVO는 무엇을 의미할까?

In a return statement, when the operand is the name of a non-volatile object with automatic storage duration, which isn't a function parameter or a catch clause parameter, and which is of the same class type (ignoring cv-qualification) as the function return type. This variant of copy elision is known as NRVO, "named return value optimization".

07번 문제(난이도 5) / 뜯어보자!!

# NRVO는 무엇을 의미할까?

NRVO는 copy elision의 종류다?

In a return statement, when the operand is the name of a non-volatile object with automatic storage duration, which isn't a function parameter or a catch clause parameter, and which is of the same class type (ignoring cv-qualification) as the function return type. This variant of copy elision is known as NRVO, "named return value optimization".

07번 문제(난이도 5) / 뜯어보자!!

# Copy Elision이란?

When copy elision occurs, the implementation treats the source and target of the omitted copy/move (since C++ 11) operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization.

복사나 이동을 생략한다?

07번 문제(난이도 5) / 뜯어보자!!

```
struct RVOclass
{
    RVOclass(){ cout << "Default Called!\n";}
    RVOclass(const RVOclass& rhs){ cout << "Copy Called!\n";}
    RVOclass(const RVOclass&& rhs){ cout << "Move Called!\n";}
    ~RVOclass(){ cout << "Destructor Called!\n";}
};
```

07번 문제(난이도 5) / 어떻게 작동할까?

```
int func()  
{  
    RVOclass data = MakeData();  
}
```

07번 문제(난이도 5) / 어떻게 작동할까?

```
RV0class MakeData()  
{  
    RV0class data;  
    return std::move(data);  
}  
  
RV0class MakeData()  
{  
    RV0class data;  
    return data;  
}
```

07번 문제(난이도 5) / 어떻게 작동할까?

# 첫 번째 예시는?

Default Called!

Move Called!

Distructor Called!

Distructor Called!

정상적으로 호출됨!

## 07번 문제 (난이도 5) / 어떻게 작동할까?

첫 번째 예시는?

Default Called!

Distructor Called!

뭔가가 없다?

07번 문제(난이도 5) / 어떻게 작동할까?



```
int func()
{
    RVOclass data = MakeData();
}
RVOclass MakeData()
{
    RVOclass data;
    return std::move(data);
}
```

MakeData()에서 생성되고 리턴!

07번 문제(난이도 5) / 어떻게 작동할까?

```
int func()  
{  
    RVOclass data = MakeData();  
}
```

```
RVOclass MakeData()  
{  
    RVOclass data;  
    return data;  
}
```

MakeData()에서 생성된 것처럼 행동한다!

07번 문제(난이도 5) / 어떻게 작동할까?

되도록이면 C++ 컴파일러를 믿고 out 포인터나  
레퍼런스를 또는 `std::move`를 사용하지 말고, 오브젝트를  
리턴 하자.

07번 문제 (난이도 5) / 결론

생각보다 많은 차이가 난다. 하지만...

정리하자면...

질문 시간!

감사합니다!