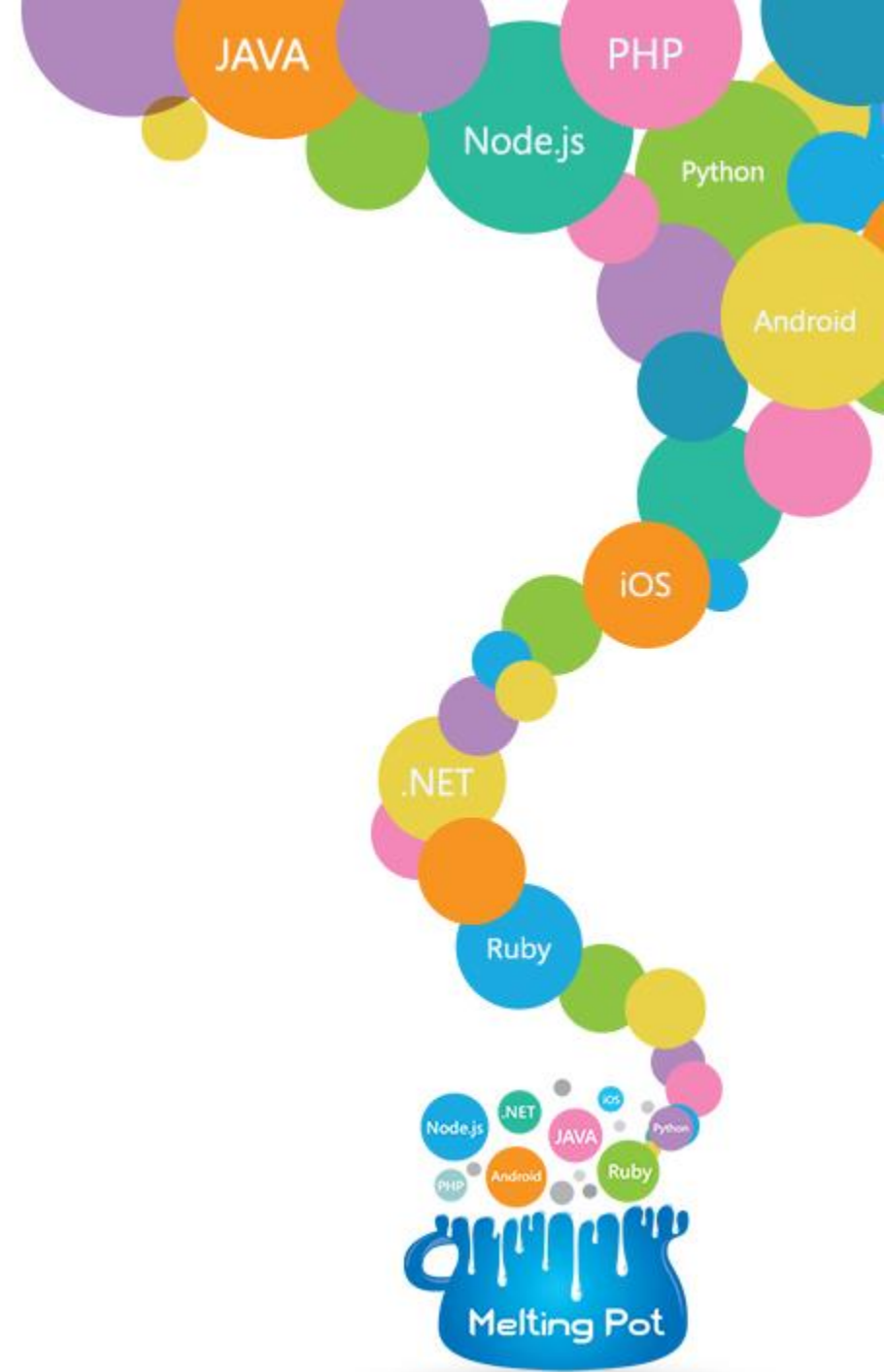


# Rvalue Reference and constexpr

김경진

Microsoft MVP(Visual C++)



vector<vector<int>>    **=default, =delete**    **atomic<T>**    **auto f() -> int**  
 user-defined literals    **thread\_local**    **array<T, N>**  
 vector<LocalType>    **C++11**    **decltype**  
**initializer lists**    **regex**    **noexcept**    **extern template**  
**constexpr**    **raw string literals**    **async**    unordered\_map<int, string>  
 template aliases    **nullptr**    **auto i = v.begin();**    delegating constructors  
**lambdas**    **override, final**    **variadic templates**    **rvalue references**  
**[] { foo(); }**    **template <typename T...>**    (move semantics)  
**unique\_ptr<T>**    **thread, mutex**    **function<>**    **future<T>**    **static\_assert(x)**  
**shared\_ptr<T>**    **for (x : coll)**    **strongly-typed enums**  
**weak\_ptr<T>**    **enum class E {...};**    tuple<int, float, string>

A word cloud of C++ 11 features centered around the text "C++ 11". The features are arranged in a circular pattern around the central text. The features include:

- vector<vector<int>>>
- =default, =delete
- atomic<T>
- auto f() -> int
- array<T, N>
- decltype
- noexcept
- extern template
- unordered\_map<int, string>
- delegating constructors
- rvalue references (move semantics)
- static\_assert(x)
- future<T>
- function<>
- strongly-typed enums
- enum class E {...};
- tuple<int, float, string>
- for (x : coll)
- thread, mutex
- shared\_ptr<T>
- weak\_ptr<T>
- unique\_ptr<T>
- lambdas
- [] { foo(); }
- template aliases
- nullptr
- auto i = v.begin();
- variadic templates
- template <typename T...>
- override, final
- constexpr
- raw string literals
- async
- initializer lists
- vector<LocalType>
- thread\_local
- user-defined literals
- regex



# Agenda

---

Rvalue Reference

Move Semantics

Perfect Forwarding

constexpr

I

# Rvalue Reference

# C언어의 Lvalue & Rvalue

|               |   |
|---------------|---|
| <b>Lvalue</b> | 대입 연산자(=)를 기준으로 <b>왼쪽과 오른쪽</b> 에 모두 사용될 수 있는 값<br>Lvalue = "Left Value" |
| <b>Rvalue</b> | 대입 연산자(=)를 기준으로 <b>오른쪽에만</b> 사용될 수 있는 값<br>Rvalue = "Right Value"       |

# C++의 Lvalue & Rvalue

- L과 R은 더 이상 Left, Right를 의미하지 않음
- Left, Right 개념은 잊어버리고, Lvalue와 Rvalue를 단순한 고유명사로 기억하자

|               |  |
|---------------|--|
| <b>Lvalue</b> | 표현식이 종료된 이후에도 없어지지 않고 지속되는 개체<br>(예: 모든 변수)      |
| <b>Rvalue</b> | 표현식이 종료되면 더 이상 존재하지 않은 일시적인 개체<br>(예: 상수, 임시 객체) |

# C++의 Lvalue & Rvalue

오른쪽 코드에서  
Rvalue를 찾아보자

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    int x = 3;
    const int y = x;
    int z = x + y;
    int* p = &x;

    cout << string("Hello");

    ++X;
    X++;
}
```



# C++의 Lvalue & Rvalue

오른쪽 코드에서  
Rvalue를 찾아보자

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    int x = 3;
    const int y = x;
    int z = x + y;
    int* p = &x;

    cout << string("Hello");

    ++x;
    x++;
}
```

# C++의 Lvalue & Rvalue

Q. 그래도 Lvalue인지 Rvalue인지 헷갈린다면?

# C++의 Lvalue & Rvalue

Q. 그래도 Lvalue인지 Rvalue인지 헷갈린다면?

A. 주소 연산자 &를 붙여서 에러가 나면 Rvalue

```
&(++x);  
&(x++);    // error C2102: '&' requires l-value
```

# Rvalue Reference (&& 참조자)

- 지금까지 사용했던 참조자(&)는 Lvalue 참조자
- C++11 표준에 Rvalue를 참조하기 위한 Rvalue Reference가 추가됨
- Rvalue 참조자 문법

```
type-id && cast-expression
```

```
ex) int&& n = rvalue();
```

# Rvalue Reference (&& 참조자)

Lvalue Reference :  
Lvalue만 참조 가능

Rvalue Reference :  
Rvalue만 참조 가능

```
int rvalue()
{
    return 10;
}

int main()
{
    int lvalue = 10;

    int& a = lvalue;
    int& b = rvalue();           // error C2440

    int&& c = lvalue;           // error C2440
    int&& d = rvalue();
}
```

# Rvalue Reference (&& 참조자)

Q. Rvalue Reference는 Lvalue일까? Rvalue일까?

# Rvalue Reference (&& 참조자)

Q. Rvalue Reference는 Lvalue일까? Rvalue일까?

A. Lvalue (Rvalue Reference  $\neq$  Rvalue)

※ 이후에 나올 내용을 이해하는데 중요한 개념

2

# Move Semantics



# Move Semantics 도입 배경

- 코드 곳곳에서 발생하는 불필요한 Rvalue 복사 과정,  
이로 인한 오버헤드

```
std::string a, b = "Hello ", c = "world";  
a = b + c;
```

```
std::string appendString(std::string param);  
std::string result = appendString("Hello");
```

# Move Semantics 도입 배경

- 코드 곳곳에서 발생하는 불필요한 Rvalue 복사 과정,  
이로 인한 오버헤드

```
std::string a, b = "Hello ", c = "world";  
a = b + c;
```



std::string 임시 객체

```
std::string appendString(std::string param);  
std::string result = appendString("Hello");
```



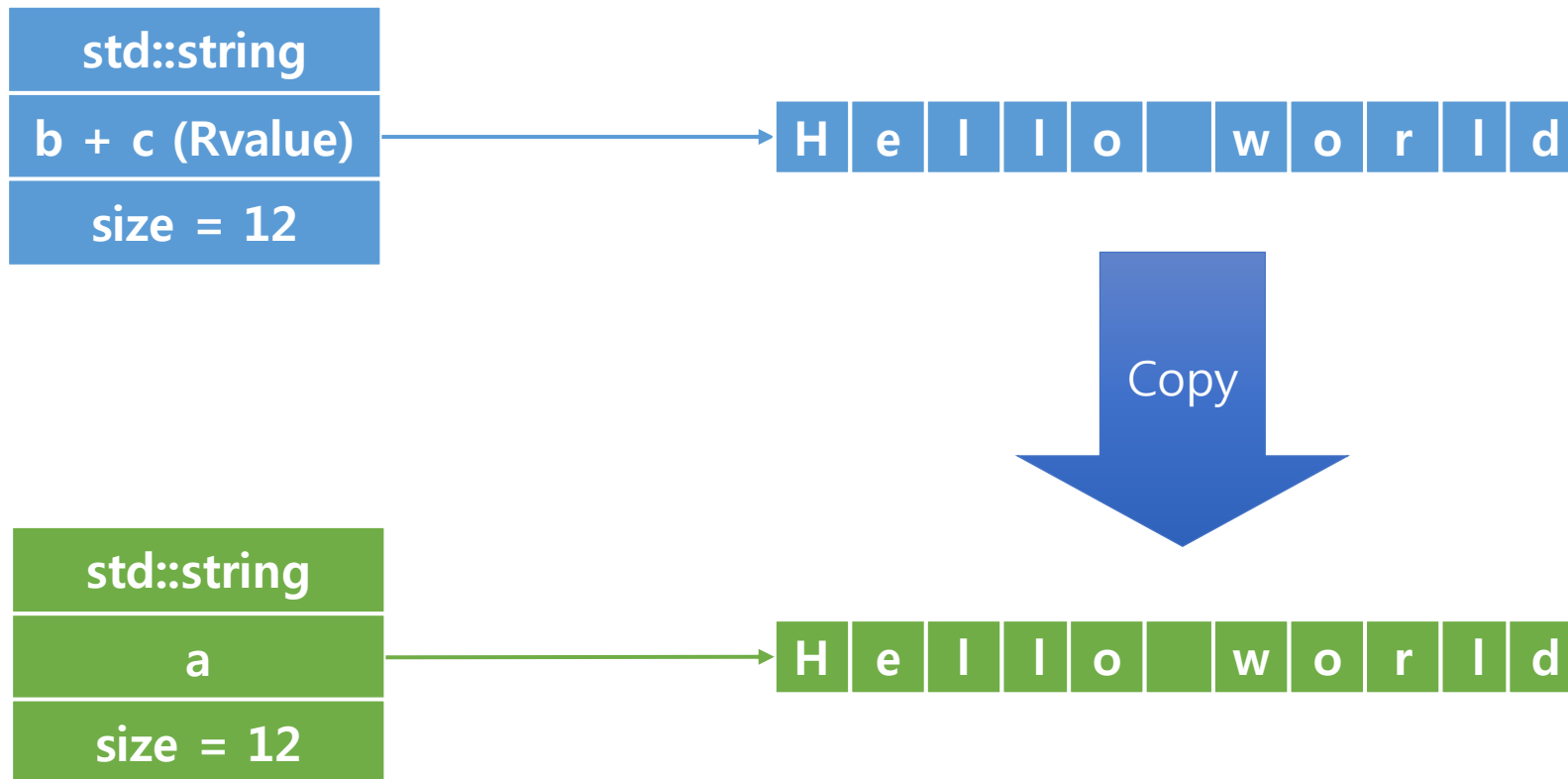
std::string 임시 객체



std::string 임시 객체

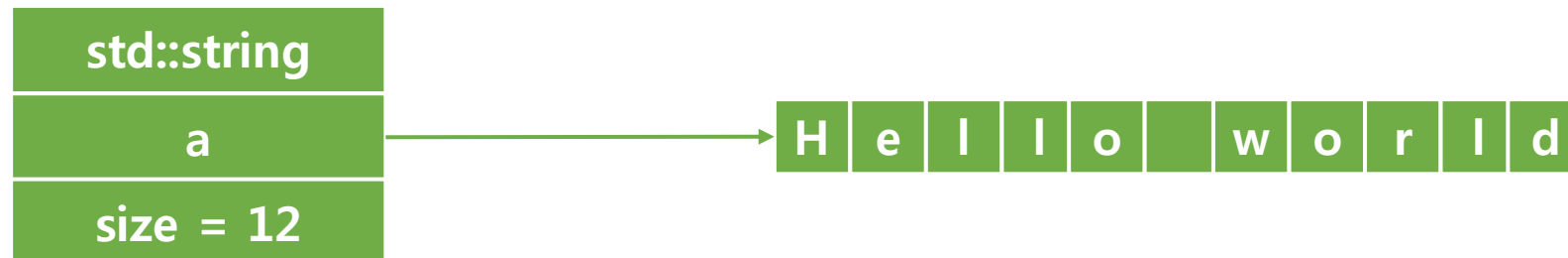
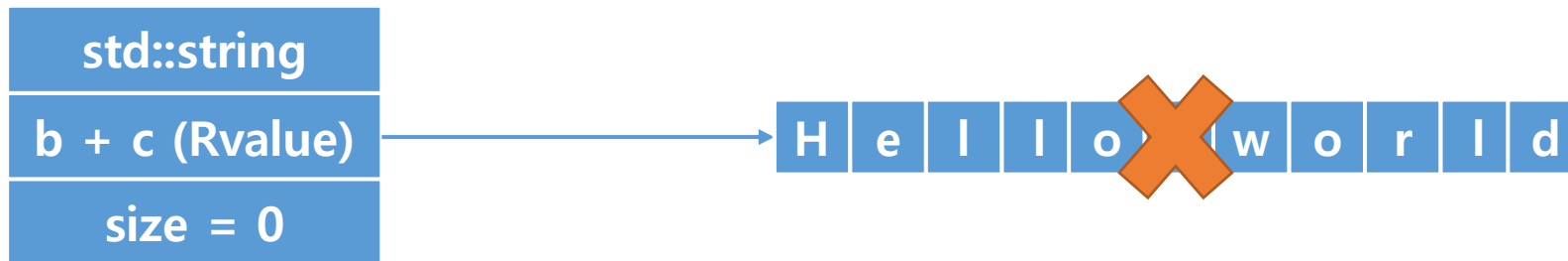
# Move Semantics 도입 배경

```
std::string a, b = "Hello ", c = "world";  
a = b + c;
```



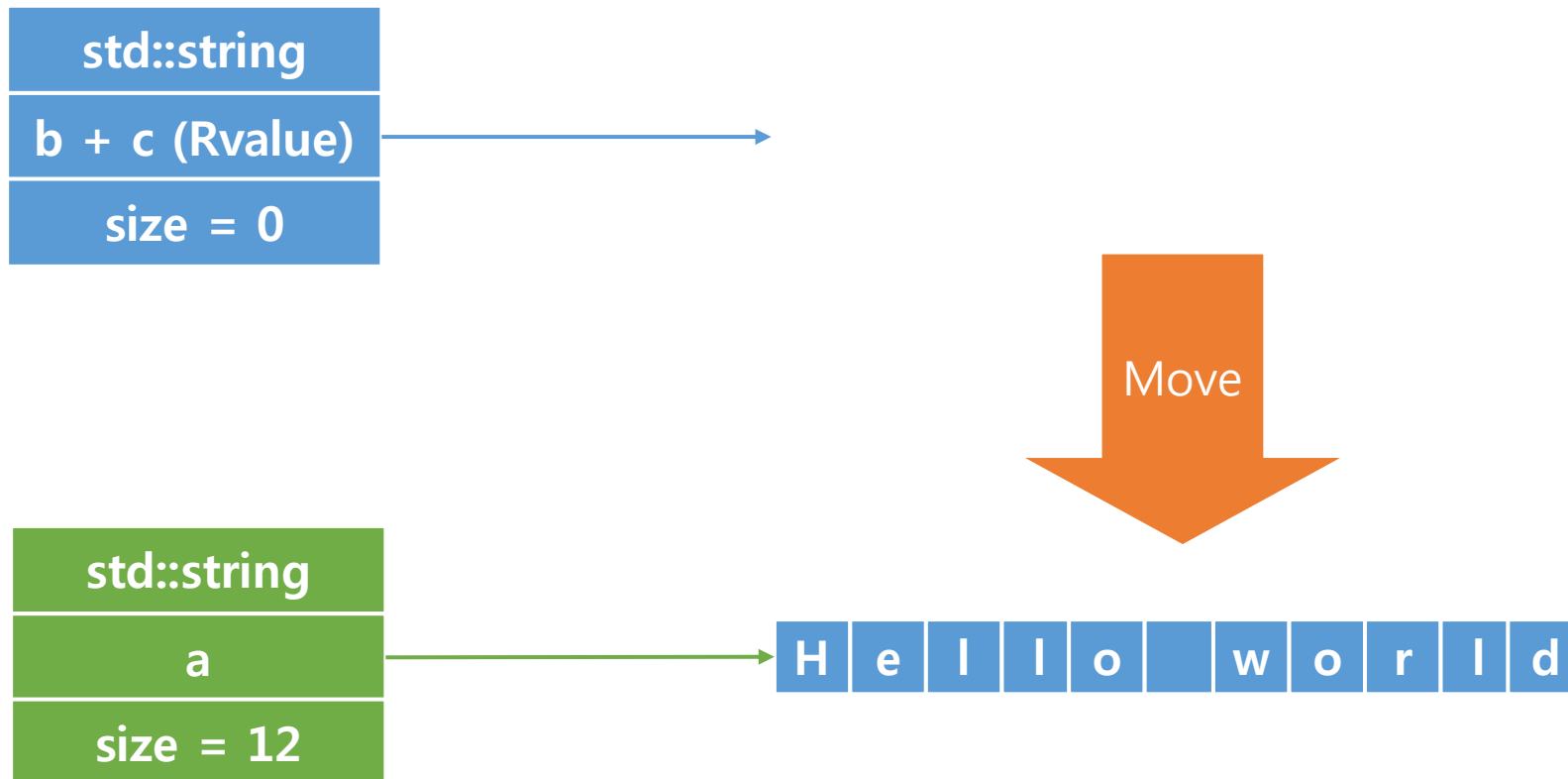
# Move Semantics 도입 배경

```
std::string a, b = "Hello ", c = "world";  
a = b + c;
```



# Move Semantics 도입 배경

```
std::string a, b = "Hello ", c = "world";  
a = b + c;
```



# Move Semantics 도입 배경

- 임시 객체(Rvalue)의 복사  
→ 이동이 되는 것이 상식적인 개념
- 하지만 C++11 이전에는 이러한 개념을  
언어 수준에서 구현할 방법이 없었음
- C++11에 이르러 객체의 이동이라는 개념이 도입됨  
→ Move Semantics

# Move Semantics 구현

Q. 내 코드에서 Move Semantics를 지원하려면?

# Move Semantics 구현

Q. 내 코드에서 Move Semantics를 지원하려면?

A-1. Move 생성자, Move 대입 연산자 구현

```
MemoryBlock(MemoryBlock&& other);  
MemoryBlock& operator=(MemoryBlock&& other);
```



# Move Semantics 구현

## Demo

# Move Semantics 구현

Q. 내 코드에서 Move Semantics를 지원하려면?

A-2. Rvalue Reference를 파라미터로 받는 함수 작성

```
std::string operator+(std::string&& left, std::string&& right)
{
    ...
}

std::string s = std::string("H") + "e" + "ll" + "o";
```

# 변화된 코딩 패러다임

- 컨테이너에 객체를 삽입할 때 더 이상 포인터를 넣지 않아도 됨

```
std::vector<MemoryBlock*> v1;
```



```
std::vector<MemoryBlock> v2;
```

- vector 컨테이너와 같은 대규모 리소스를 반환하는 함수 작성 가능

```
void createVector(std::vector<MemoryBlock>& v) { ... }
```



```
std::vector<MemoryBlock> createVector() { ... }
```

# std::move 함수

Q. std::move 함수가 수행하는 일은?

# std::move 함수

Q. std::move 함수가 수행하는 일은?

A. 파라미터를 '무조건' Rvalue Reference로 타입캐스팅

```
template<class T>
typename remove_reference<T>::type&&
move(T&& _Arg) {
    return static_cast<remove_reference<T>::type&&>(_Arg);
}
```

※ 절대 이름에 낚이지 말자!!

std::move 호출만으로는 아무것도 이동시키지 않음...



# std::move 함수

- Lvalue를 Rvalue로 취급하고 싶을 때 사용  
(컴파일러에게 해당 객체가 이동해도 무관한 객체임을 알려주기 위해)

```
class Person {  
public:  
    void setName(std::string&& newName) {  
        name = newName;  
    }  
    Person(Person&& other) {  
        name = other.name;  
    }  
    std::string name;  
};
```

# std::move 함수

- Lvalue를 Rvalue로 취급하고 싶을 때 사용  
(컴파일러에게 해당 객체가 이동해도 무관한 객체임을 알려주기 위해)

```
class Person {  
public:  
    void setName(std::string&& newName) {  
        name = newName;  Rvalue 복사 발생  
    }  
    Person(Person&& other) {  
        name = other.name;  Rvalue 복사 발생  
    }  
    std::string name;  
};
```

# std::move 함수

- Lvalue를 Rvalue로 취급하고 싶을 때 사용  
(컴파일러에게 해당 객체가 이동해도 무관한 객체임을 알려주기 위해)

```
class Person {  
public:  
    void setName(std::string&& newName) {  
        name = std::move(newName);  
    }  
    Person(Person&& other) {  
        name = std::move(other.name);  
    }  
    std::string name;  
};
```



3

Perfect Forwarding



# && 참조자의 두 얼굴

오른쪽 코드에서  
Rvalue Reference  
를 찾아보자

① `void foo(string&& param);`

② `string&& var1 = string();`

③ `auto&& var2 = var1;`

④ `template<typename T>  
void foo(T&& param);`

# && 참조자의 두 얼굴

오른쪽 코드에서  
Rvalue Reference  
를 찾아보자

- ① `void foo(string&& param);`  
↓  
Rvalue Reference (O)
- ② `string&& var1 = string();`  
↓  
Rvalue Reference (O)
- ③ `auto&& var2 = var1;`  
↓  
Rvalue Reference (X)
- ④ `template<typename T>  
void foo(T&& param);`  
↓  
Rvalue Reference (X)

# && 참조자의 두 얼굴

- && 참조자가 템플릿 함수의 템플릿 파라미터 또는 auto 타입과 함께 사용되었을 경우

→ Universal Reference

```
template<typename T>  
void foo(T&& param);
```



Universal Reference

```
auto&& var2 = var1;
```



Universal Reference

# Universal Reference

- Lvalue와 Rvalue를 모두 참조할 수 있는 포괄적(Universal) 레퍼런스
- 반드시 Rvalue Reference와 구분되어야 함
- Perfect Forwarding 구현을 위한 열쇠

# Universal Reference

- Lvalue와 Rvalue를 모두 참조할 수 있는 포괄적(Universal) 레퍼런스
- 반드시 Rvalue Reference와 구분되어야 함
- Perfect Forwarding 구현을 위한 열쇠



- Old C++ 에서는 해결할 수 없는 문제가 하나 있었다.

# Forwarding Problem

- Old C++ 에서는 해결할 수 없는 문제가 하나 있었다.



# Forwarding Problem

- `make_shared` 함수와 같은 `factory` 함수 작성

```
template<typename T, typename Arg>
T* factory(Arg& arg)
{
    return new T(arg);
}
```

```
struct X
{
    X(int& n) {}
};

int n = 0;
X* px = factory<X>(n);
```

# Forwarding Problem

- `make_shared` 함수와 같은 `factory` 함수 작성

```
template<typename T, typename Arg>
T* factory(Arg& arg)
{
    return new T(arg);
}
```

```
struct Y
{
    Y(const int& n) {}
};
```

```
// error C2664: cannot convert argument 1 from 'int' to 'int &'
Y* py = factory<Y>(10);
```

# Forwarding Problem

- 결국 non-const 버전과 const 버전을 모두 제공해야 함

```
template<typename T, typename Arg>
T* factory(Arg& arg)
{
    return new T(arg);
}
```

```
template<typename T, typename Arg>
T* factory(const Arg& arg)
{
    return new T(arg);
}
```

# Forwarding Problem

- 작성해야 하는 함수의 개수는  $2^n$  으로 증가  
( $n$  = 파라미터 개수)
- 가변인자 함수는 답이 없음

# Forwarding Problem

- 작성해야 하는 함수의 개수는  $2^n$  으로 증가  
( $n$  = 파라미터 개수)
- 가변인자 함수는 답이 없음
- 하지만 Universal Reference가 출동하면 어떨까?

# Perfect Forwarding 1단계

- Universal Reference를 이용한 구현

```
template<typename T, typename Arg>
T* factory(Arg&& arg)
{
    return new T(arg);
}
```

```
int n = 0;
X* px = factory<X>(n);           // OK
Y* py = factory<Y>(10);          // OK
```

# 템플릿 타입 추론 규칙

```
template <typename T>  
void deduce(T&& param);
```

```
int n = 0;  
deduce(n); // Lvalue 전달: T -> int& 로 추론  
  
deduce(10); // Rvalue 전달: T -> int 로 추론
```

# 템플릿 타입 추론 규칙

```
template <typename T>  
void deduce(T&& param);
```

```
int n = 0;  
deduce(n); // Lvalue 전달: T -> int& 로 추론  
  
deduce(10); // Rvalue 전달: T -> int 로 추론
```



```
void deduce(int& && param); // Lvalue 전달  
Void deduce(int&& param); // Rvalue 전달
```



# Reference Collapsing Rules

| Reference 경합 | Reference 붕괴 |
|--------------|--------------|
| T& &         | T&           |
| T& &&        | T&           |
| T&& &        | T&           |
| T&& &&       | T&&          |



```
void deduce(int& param);           // Lvalue 전달
```

```
Void deduce(int&& param);          // Rvalue 전달
```

# Perfect Forwarding 2단계

- 마지막으로 해결해야할 문제
- Lvalue  $\rightarrow$  Lvalue, Rvalue  $\rightarrow$  Rvalue로 전달해야 완벽한 포워딩

```
template<typename T, typename Arg>
T* factory(Arg&& arg)
{
    return new T(arg);
}
```

# Perfect Forwarding 2단계

- 마지막으로 해결해야할 문제
- Lvalue → Lvalue, Rvalue → Rvalue로 전달해야 완벽한 포워딩

```
template<typename T, typename Arg>
T* factory(Arg&& arg)
{
    return new T(std::forward<Arg>(arg));
}
```

# std::forward 함수

- 파라미터를 '조건에 따라' Rvalue Reference로 타입캐스팅  
→ Rvalue/Rvalue Reference일 경우에만 Rvalue Reference로 타입 캐스팅
- Reference Collapsing Rules를 이용

```
template<class T>
T&& forward(remove_reference<T>::type&& _Arg) {
    return (static_cast<T&&>(_Arg));
}
```

※ 이것도 이름에 낚이지 말자!!

std::forward 호출만으로는 아무것도 전달하지 않음...

# std::move 와 std::forward

Q. 둘 중 어떤 함수를 사용해야 할 지 헷갈린다면?

# std::move 와 std::forward

Q. 둘 중 어떤 함수를 사용해야 할 지 헷갈린다면?

A. Rvalue Reference 에는 std::move 함수,

Universal Reference 에는 std::forward 함수 사용

4

constexpr

# constexpr 핵심 엿보기

- 컴파일 타임 처리
- 메타 프로그래밍



# 컴파일 타임과 런타임

|        |                         |
|--------|-------------------------|
| 컴파일 타임 | 컴파일러가 바이너리 코드를 만들어내는 시점 |
| 런타임    | 프로그램을 실행하여 실제로 동작되는 시점  |

# 컴파일 타임 처리

- 런타임에 수행할 작업을 컴파일 타임에 미리 수행하여 상수화  
→ 컴파일 시간은 다소 늘어나지만 런타임 퍼포먼스는 증가

# 메타 프로그래밍

- '프로그램에 대한 프로그래밍'

예) Excel의 매크로 함수, lex & yacc 구문 분석기

- C++ 메타 프로그래밍

→ 컴파일 타임에 처리되는 일련의 작업을 프로그래밍 하는 것

# 메타 프로그래밍

- C++ 메타 프로그래밍의 목적
  - 프로그램이 실행되기 전에 최대한 많은 일을 해둠으로써  
퍼포먼스 증가
- 템플릿 메타 프로그래밍
  - 난이도가 높고, 가독성이 떨어짐
- constexpr을 이용하면 아주 쉽게 메타 프로그래밍 가능

# constexpr specifier

- `const`, `static`과 같은 한정자 (변수, 함수에 사용)
- '컴파일 타임에 값을 도출하겠다' 라는 의미를 가짐

# constexpr 변수

- '변수의 값을 컴파일 타임에 결정하여 상수화 하겠다' 라는 의미
- 반드시 상수 식으로 초기화 되어야 함

```
constexpr int n = 0;           // OK  
constexpr int m = std::time(NULL); // error C2127
```

# constexpr 함수

- '함수 파라미터에 상수식이 전달될 경우, 함수 내용을 컴파일 타임에 처리하겠다' 라는 의미

```
constexpr int square(int x) {  
    return x * x;  
}
```

- 전달되는 파라미터에 따라 컴파일타임, 런타임 처리가 결정됨

```
int n;  
std::cin >> n;  
square(n);    // 런타임 처리  
  
square(10);   // 컴파일 타임 처리
```

# constexpr 함수 제한 조건

- 함수 내에서는 하나의 표현식만 사용할 수 있으며, 반드시 리터럴 타입을 반환해야 함

```
constexpr LiteralType func() { return expression; }
```



# constexpr 함수로 변환

- if / else 구문
- for / while 루프
- 변수 선언

# constexpr 함수로 변환

- if / else 구문 → 삼항 연산자 ( $x > y ? x : y$ )
- for / while 루프 → 재귀 호출
- 변수 선언 → 파라미터 전달

# 2~n 소수의 합 구하기

```
bool IsPrime(int number) {  
    if (number <= 1)  
        return false;  
  
    for (int i = 2; i * i <= number; ++i)  
        if (number % i == 0)  
            return false;  
  
    return true;  
}  
  
int PrimeSum(int number) {  
    int sum = 0;  
    for (int i = number; i >= 2; --i)  
        if (IsPrime(i))  
            sum += i;  
  
    return sum;  
}
```

# IsPrime 템플릿 메타프로그래밍 버전

```
struct false_type
{
    typedef false_type type;
    enum { value = 0 };
};

struct true_type
{
    typedef true_type type;
    enum { value = 1 };
};

template<bool condition, class T, class U>
struct if_
{
    typedef U type;
};

template <class T, class U>
struct if_ < true, T, U >
{
    typedef T type;
};
```

```
template<size_t N, size_t c>
struct is_prime_impl {
    typedef typename if_<(c*c > N),
        true_type,
        typename if_ < (N % c == 0),
            false_type,
            is_prime_impl<N, c + 1> > ::type > ::type type;
    enum { value = type::value };
};

template<size_t N>
struct is_prime {
    enum { value = is_prime_impl<N, 2>::type::value };
};

template <>
struct is_prime <0> {
    enum { value = 0 };
};

template <>
struct is_prime <1> {
    enum { value = 0 };
};
```

# IsPrime 템플릿 메타프로그래밍 버전



어때요 ? 참 쉽죠 ?

# constexpr을 이용한 메타 프로그래밍

## Demo

# constexpr 관련 라이브러리

- Sprout C++ Libraries (Bolero MURAKAMI)  
<http://bolero-murakami.github.io/Sprout/>



# C++14 constexpr 제한 조건 완화

- 변수 선언 가능(static, thread\_local 제외)
- if / switch 분기문 사용 가능
- range-based for 루프를 포함한 모든 반복문 사용 가능



# Summary

| 키워드                       | 내용                                      |
|---------------------------|---|
| <b>Rvalue</b>             | 표현식이 종료되면 사라지는 임시적인 개체                  |
| <b>Rvalue Reference</b>   | && 참조자를 사용하여 Rvalue 참조 가능               |
| <b>Move Semantics</b>     | 객체의 이동, Move 생성자, Move 대입 연산자 구현        |
| <b>Perfect Forwarding</b> | Universal Reference, std::forward 함수 이용 |
| <b>constexpr</b>          | 컴파일 타임 처리, 메타 프로그래밍                     |

Thank you!



# 참고 자료

- Effective Modern C++ (Scott Meyers)
- <http://blogs.embarcadero.com/jtembarcadero/2012/11/12/my-top-5-c11-language-and-library-features-countdown/>
- <http://msdn.microsoft.com/en-us/library/dd293668.aspx>
- <http://blogs.msdn.com/b/vcblog/archive/2009/02/03/rvalue-references-c-0x-features-in-vc10-part-2.aspx>
- <http://msdn.microsoft.com/en-us/library/dd293665.aspx>
- <http://www.codeproject.com/Articles/397492/Move-Semantics-and-Perfect-Forwarding-in-Cplusplus>
- <http://www.codeproject.com/Articles/453022/The-new-Cplusplus-rvalue-reference-and-why-you>
- <http://kholdstare.github.io/technical/2013/11/23/moves-demystified.html>
- <http://en.cppreference.com/w/cpp/language/constexpr>
- <http://blog.smartbear.com/c-plus-plus/using-constexpr-to-improve-security-performance-and-encapsulation-in-c/>
- <http://www.codeproject.com/Articles/417719/Constants-and-Constant-Expressions-in-Cplusplus>
- <http://cpptruths.blogspot.kr/2011/07/want-speed-use-constexpr-meta.html>
- <http://enki-tech.blogspot.kr/2012/09/c11-compile-time-calculator-with.html>
- <http://en.wikipedia.org/wiki/C%2B%2B14>



Microsoft