# TMP: template meta-programming (II)

Deep dive into C++

scc@teamt5.org

# Outline


Template 101 and type deduction

**Perfect forwarding and reference collapsing**

Variadic template and parameter pack

SFINAE

CTAD

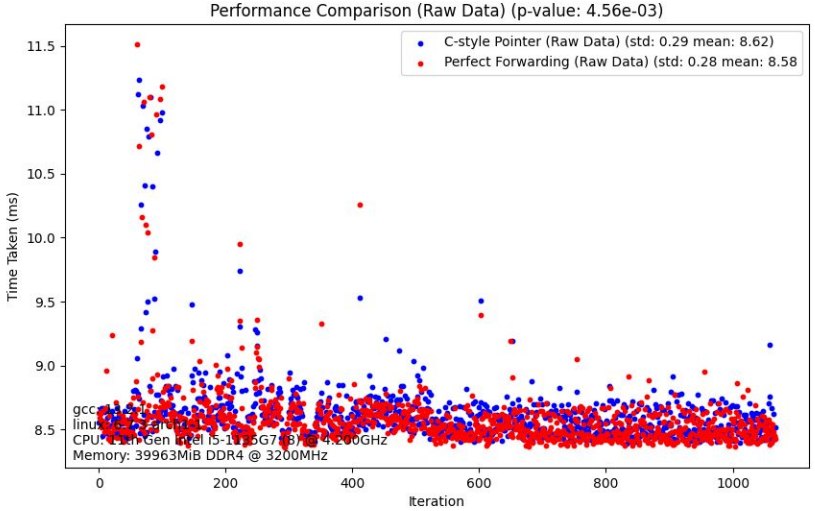
Expression template

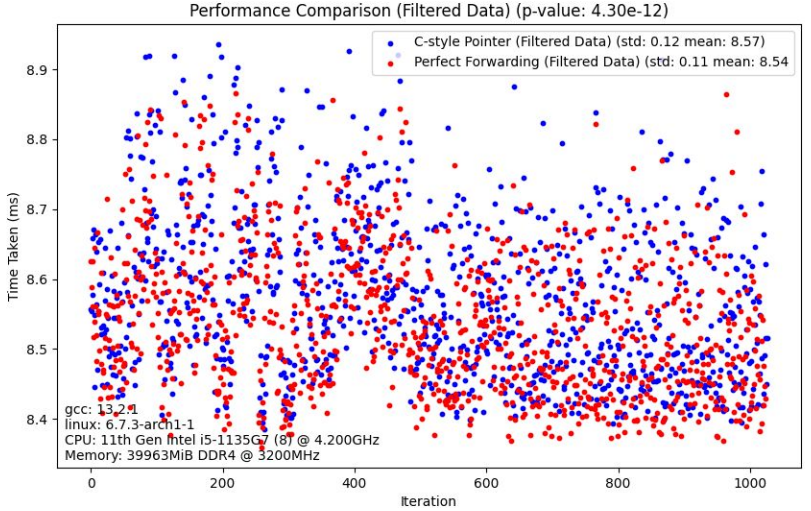
Concepts and constraints

CRTP and compile time polymorphism

# Perfect forwarding / Reference collapsing

# Statistics (4 pages object)

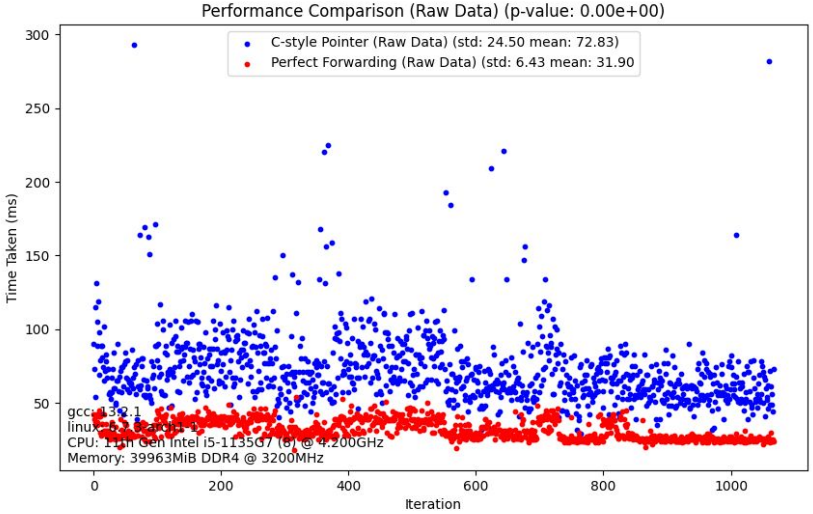


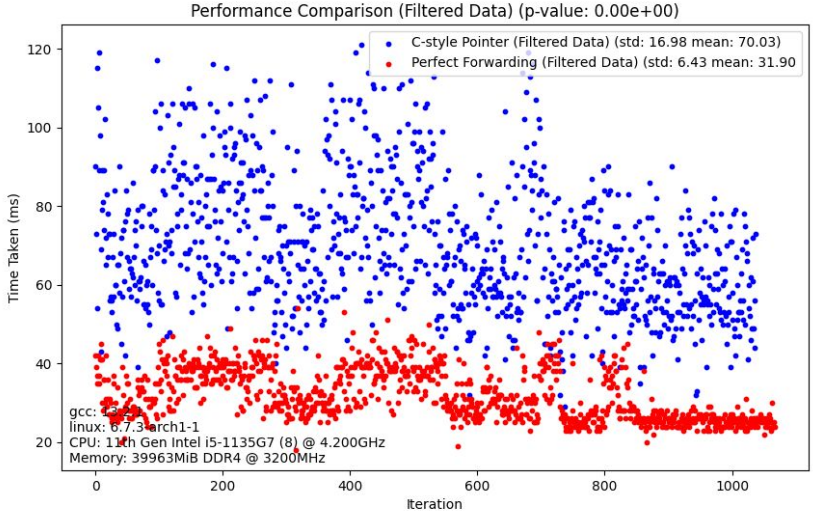




## Source code and raw data

https://gist.github.com/25077667/e7894cf9bf558613ffc900c88f98214f

# Statistics (empty object)




Source code and raw data

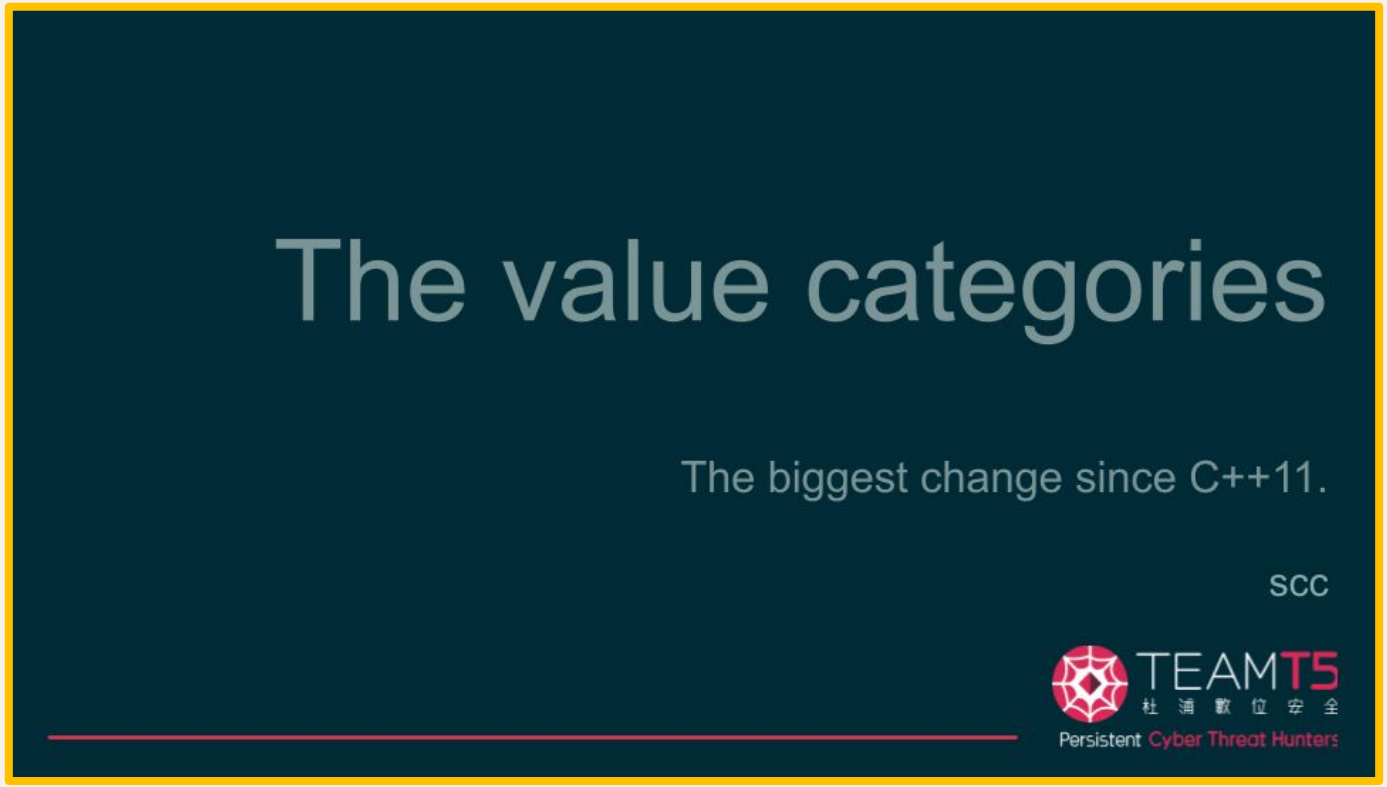
https://gist.github.com/25077667/e7894cf9bf558613ffc900c88f98214f

Typo: Time should be ns

C++ can be faster than the C.

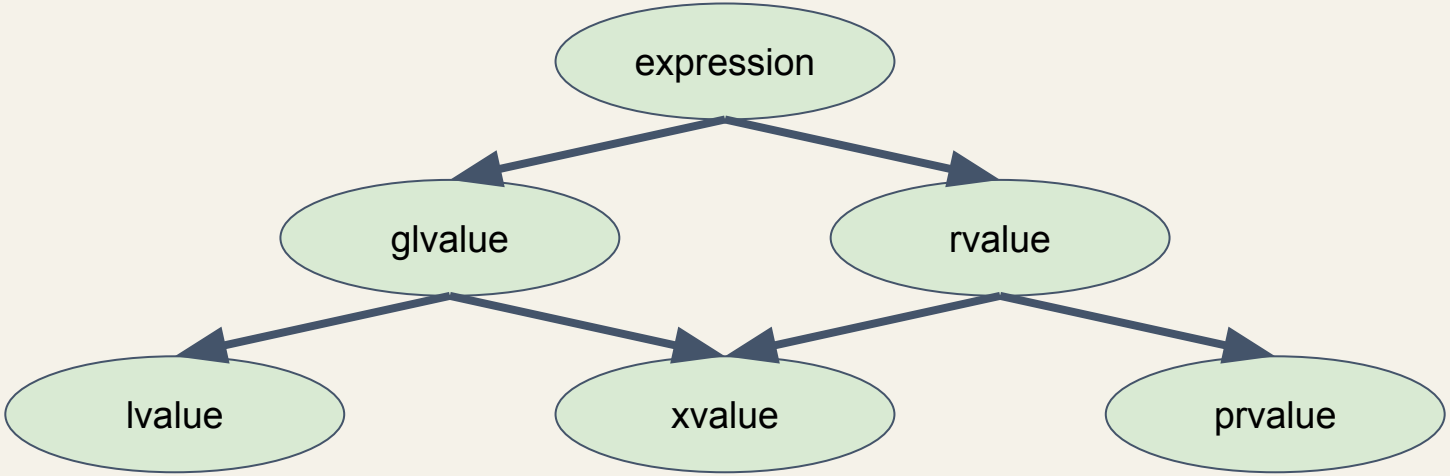
# Let’s intro. perfect forwarding

# We should know lr-value ref

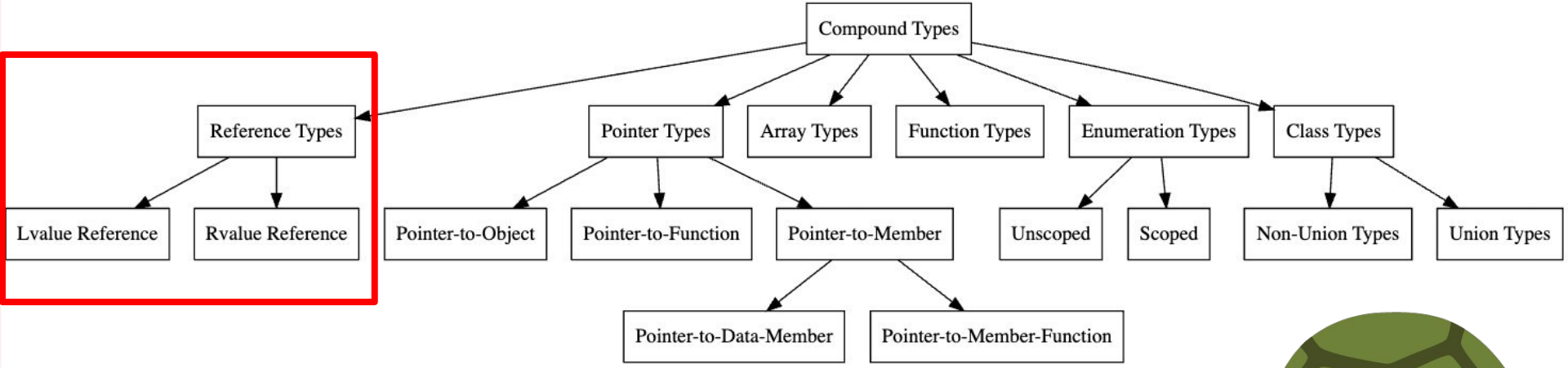


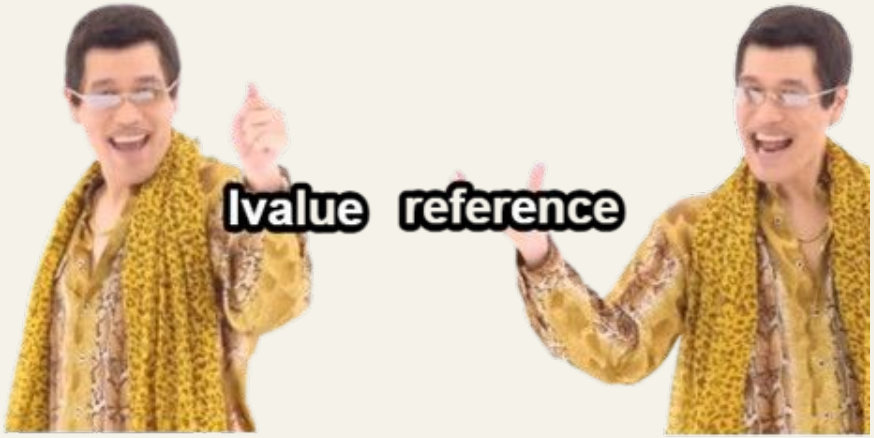
# lr-value

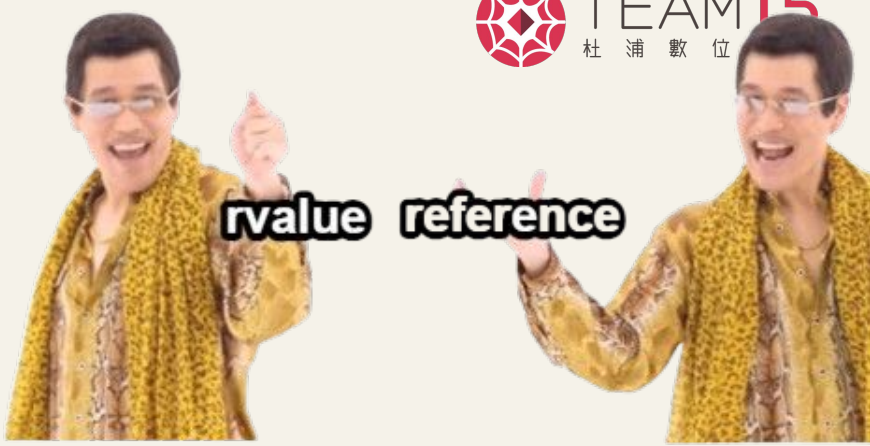
TEAMT5
杜 浦 數 位 安 全



# Reference type





lvalue reference

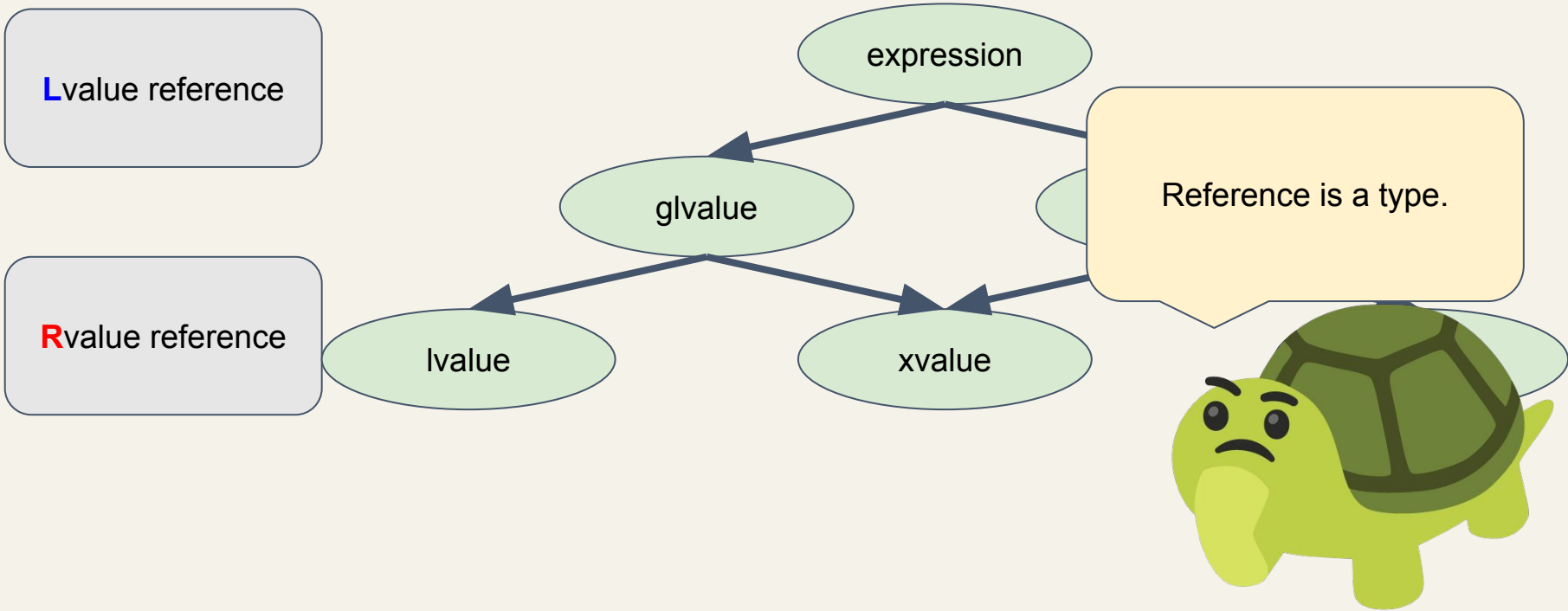

rvalue reference


lvalue reference


rvalue reference

# Reference

Declares a named variable as a reference, that is, an alias to an already-existing object or function.



https://godbolt.org/z/sE3M8aE1j

TEAMT5 杜浦數位安全

```
void processLvalueReference(int& lref) {
    std::cout << "Lvalue reference: " << lref << std::endl;
}

void processRvalueReference(int&& rref) {
    std::cout << "Rvalue reference: " << rref << std::endl;
}

int main() {
    int a = 5;
    processLvalueReference(a); // a is an lvalue

    processRvalueReference(10); // 10 is an rvalue
    return 0;
}
```

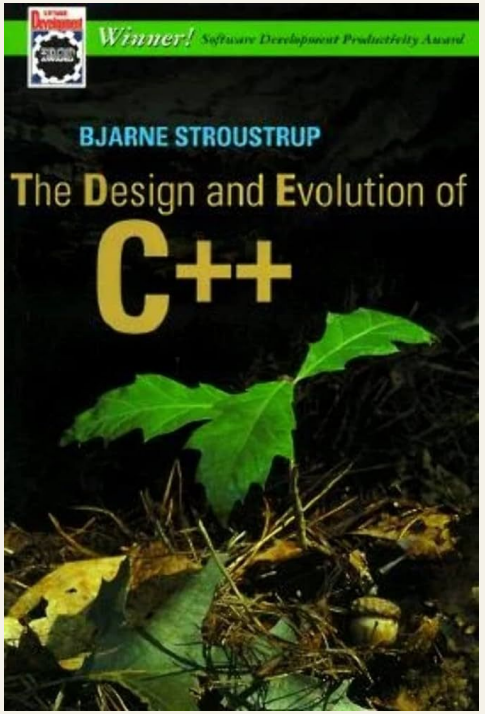
```
Lvalue reference: 5
Rvalue reference: 10
```

A named rvalue reference is considered an lvalue.

# Named rvalue-ref is considered an lvalue

This can be a bit counterintuitive because we usually associate rvalue references with temporary objects or "rvalues".

However, once you give a name to an rvalue reference, you **can refer to it more than once**, so in the context of the language, it is treated as an lvalue.



# So, the perfect forwarding

TEAMT5
杜 浦 數 位 安 全

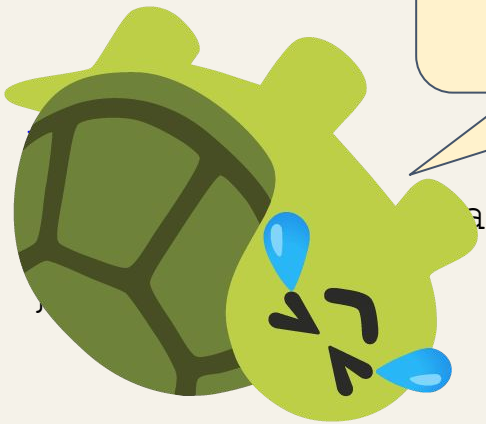
```
#include <vector>

template<typename T>
class Vector {
    std::vector<T> data;

public:
    void push_bach(T&& v)   { data.push_back(v); }
};
```

# So, the perfect forwarding

TEAMT5
杜 浦 數 位 安 全

```
#include <vector>

template<typename T>
class Vector {
    std::vecto
```

It becomes an l-value!!



```
ach(T&& v)   { data.push_back(v); }
```

# So, the perfect forwarding

TEAMT5
杜 浦 數 位 安 全

```
#include <vector>

t
c
```

L-value needs an extra copy!

es an l-value!!

```
ach(T&& v)   { data.push_back(v); }
```

# So, the perfect forwarding

TEAMT5 杜浦數位安全

```
#include <vector>

template<typename T>
class Vector {
    std::vector<T> data;

public:
void push_bach(T&& v) { data.push_back( std::forward<T>(v)); }
};
```

# So, the perfect forwarding

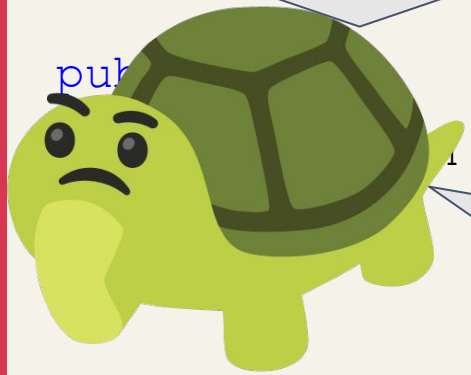
TEAMT5 杜浦數位安全

```
#include <vector>


template<typename T>
clas

pub
```

```
(T&
```

```
ck( std::forward<T>(v)); }
```

So, what is perfect forwarding?



Why not just std::move?

# Why not just std::move?


The std::move make the variable becomes rvalue ref.

But we might not always need rvalue reference.

```
int main() {
    std::string msg;
    logger.info(std::move(msg)); // if it takes moving semantics
    // The msg becomes undefined behavior here.
}
```

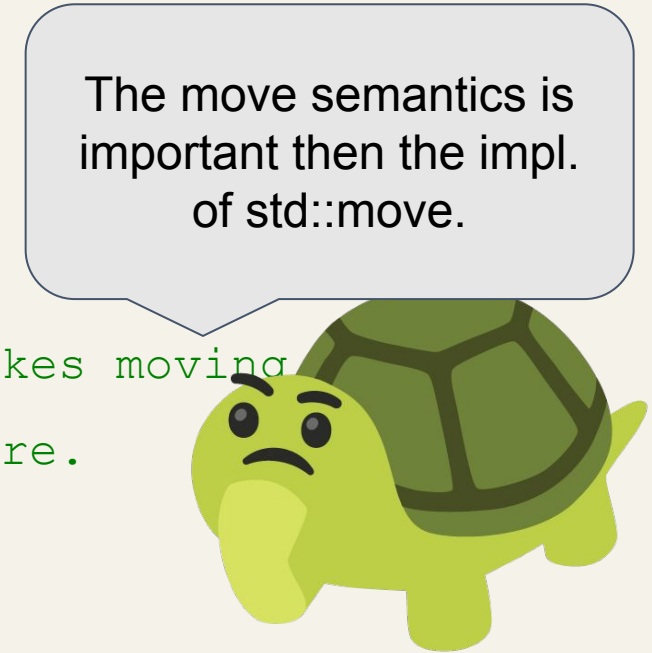
# Why not just std::move?

TEAMT5 杜浦數位安全

The std::move make the variable becomes rvalue ref.

But we might not always need rvalue reference.

```
int main() {
    std::string msg;
    logger.info(std::move(msg)); // if it takes moving
    // The msg becomes undefined behavior here.
}
```



# So. std::forward<T>

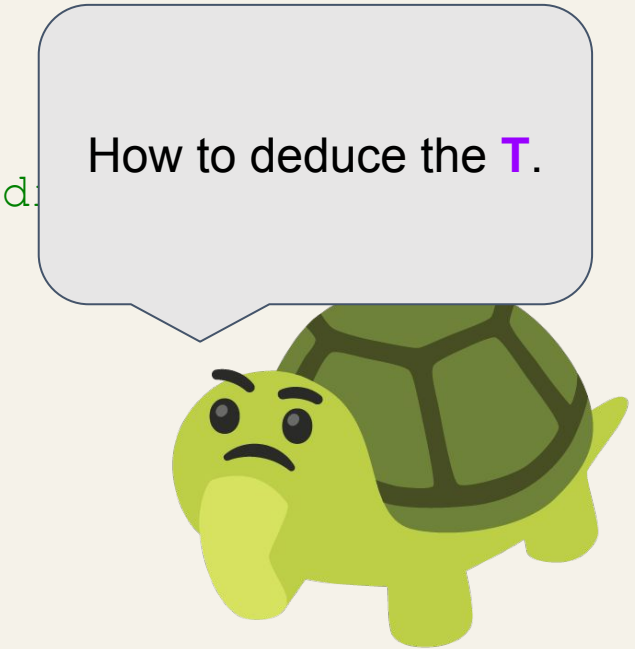
TEAMT5
杜 浦 數 位 安 全

```
template<class T>
void wrapper(T&& arg) {
  // Forward as lvalue or as rvalue, depending on T
   foo(std::forward<T>(arg));
}
```

# So. std::forward<T>

TEAMT5
杜 浦 數 位 安 全

```
template<class T>
void wrapper(T&& arg) {
  // Forward as lvalue or as rvalue, depend
   foo(std::forward<T>(arg));
}
```



# Reference collapsing

# One slide

TEAMT5
杜 浦 數 位 安 全

| argument | template | result |
|---|---|---|
| & | & | & |
| && | & | & |
| & | && | & |
| && | && | && |

Same logic as the and operation

& is false
&& is true

# One slide

TEAMT5
杜 浦 數 位 安 全

| argument | template | result |
|---|---|---|
| & | & | & |
| && | & | & |
| & | && | & |
| && | && | && |

```
template<class T>
void wrapper(const T& arg) {
    foo(std::forward<T>(arg));
}
```

# One slide

TEAMT5 杜浦數位安全

| argument | template | result |
|---|---|---|
| & | & | & |
| && | & | & |
| & | && | & |
| && | && | && |

```
template<class T>
void wrapper(const T& arg) {
    foo(std::forward<T>(arg));
}

int main() {
    std::string s;
    wrapper(s);
}
```

# One slide

TEAMT5
杜 浦 數 位 安 全

| argument | template | result |
|---|---|---|
| & | & | & |
| && | & | & |
| & | && | & |
| && | && | && |

```
template<class T>
void wrapper(const T& arg) {
    foo(std::forward<T>(arg));
}

int main() {
    std::string s;
    wrapper(std::move(s));
}
```

# One slide

TEAMT5
杜 浦 數 位 安 全

| argument | template | result |
|---|---|---|
| & | & | & |
| && | & | & |
| & | && | & |
| && | && | && |

```
template<class T>
void wrapper(T&& arg) {
    foo(std::forward<T>(arg));
}
```

# One slide

TEAMT5
杜 浦 數 位 安 全

| argument | template | result |
|---|---|---|
| & | & | & |
| && | & | & |
| & | && | & |
| && | && | && |

```
template<class T>
void wrapper(T&& arg) {
    foo(std::forward<T>(arg));
}

int main() {
    std::string s;
    wrapper(s);
}
```

# One slide

TEAMT5 杜浦數位安全

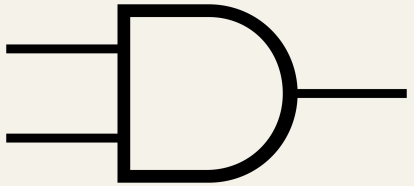
| argument | template | result |
|---|---|---|
| & | & | & |
| && | & | & |
| & | && | & |
| && | && | && |

```
template<class T>
void wrapper(T&& arg) {
    foo(std::forward<T>(arg));
}

int main() {
    std::string s;
    wrapper(std::move(s));
}
```

# Take away

TEAMT5
杜 浦 數 位 安 全

1. Perfect forwarding could be faster than C
2. Named rvalue reference is considered an lvalue
3. Be aware of the move semantics
4. Reference collapsing is similar with logical *and*



# Thank you

scc@teamt5.org

TEAMT5

杜 浦 數 位 安 全

Persistent Cyber Threat Hunters