

C++ Difference between std::ref(T) and T&?

```

36 if (dev.isBored() || job.sucks()) {
37     searchJobs({flexibleHours: true, companyCulture: 100});
38 }
39 // A career site that's by developers, for developers.

```



I have some questions regarding this program:

```

#include <iostream>
#include <type_traits>
#include <functional>
using namespace std;
template <typename T> void foo ( T x )
{
    auto r=ref(x);
    cout<<boolalpha;
    cout<<is_same<T&,decltype(r)>::value;
}
int main()
{
    int x=5;
    foo (x);
    return 0;
}

```

The output is:

```
false
```

I want to know, if `std::ref` doesn't return the reference of an object, then what does it do? Basically, what is the difference between:

```

T x;
auto r = ref(x);

```

and

```

T x;
T &y = x;

```

Also, I want to know why does this difference exist? Why do we need `std::ref` or `std::reference_wrapper` when we have references (i.e. `T&`)?

c++ reference ref

edited Oct 20 '15 at 19:12

Thierry
591 5 16

asked Oct 20 '15 at 15:41

CppNITR
500 1 4 14

Possible duplicate of [How is `tr1::reference_wrapper` useful?](#) – anderas Oct 20 '15 at 15:47

Hint: what happens if you do `x = y;` in both cases? – juanchopanza Oct 20 '15 at 15:47

1 In addition to my duplicate flag (last comment): See for example [stackoverflow.com/questions/31270810/...](#) and [stackoverflow.com/questions/26766939/...](#) – anderas Oct 20 '15 at 15:48

2 @andas it's not about usefulness, it's about the difference mainly – CppNITR Oct 20 '15 at 15:48

@CppNITR Then see the questions I linked a few seconds before your comment. Especially the second one is useful. – anderas Oct 20 '15 at 15:49

3 Answers

Well `ref` constructs an object of the appropriate `reference_wrapper` type to hold a reference to an object. Which means when you apply :-

```
auto r = ref(x);
```

This return a `reference_wrapper` & not a direct reference to `x` (ie `T&`) . This `reference_wrapper` (ie `r`) instead holds `T&` .

A `reference_wrapper` is very useful when you want to emulate a `reference` of an object which can be copied (it is both `copy-constructible` and `copy-assignable`).

In C++, once you create a `reference` (say `y`) to an object (say `x`) , then `y` & `x` share the same base address . Furthermore, `y` cannot refer to any other object. Also you cannot create an array of references ie a code like this will throw an error :-

```

#include <iostream>
using namespace std;
int main()
{
    int x=5, y=7, z=8;
    int& arr[] {x,y,z};    // error: declaration of 'arr' as array of references
}

```

```
    return 0;
}
```

However this is legal :-

```
#include <iostream>
#include <functional> // for reference_wrapper
using namespace std;
int main()
{
    int x=5, y=7, z=8;
    reference_wrapper<int> arr[] {x,y,z};
    for (auto a:arr)
        cout<<a<<" ";
    return 0;
}
/* OUTPUT :-
5 7 8
*/
```

Talking about your problem with `cout<<is_same<T&,decltype(r)>::value;`, the solution is :-

```
cout<<is_same<T&,decltype(r.get())>::value; // will yield true
```

Let me show you a program :-

```
#include <iostream>
#include <type_traits>
#include <functional>
using namespace std;
int main()
{
    cout<<boolalpha;
    int x=5, y=7;
    reference_wrapper<int> r=x; // or auto r = ref(x);
    cout<<is_same<int&, decltype(r.get())>::value<<"\n";
    cout<<(&x==&r.get())<<"\n";
    r=y;
    cout<<(&y==&r.get())<<"\n";
    r.get()=70;
    cout<<y;
    return 0;
}
/* Output :-
true
true
true
70
*/
```

See here we get to know three things :-

1. `reference_wrapper` object (here `r`) can be used to create an array of references which was not possible with `T&`.
2. `r` actually acts like a real reference (see how `r.get()=70` changed the value of `y`).
3. `r` is not same as `T&` but `r.get()` is. This means that `r` holds `T&` ie as it's name suggests is a wrapper around reference `T&`.

I hope this answer is more than enough to explain your doubts.

edited Dec 13 '15 at 6:29

answered Oct 20 '15 at 18:08



Ankit Acharya
1,006 7 19

1: No, a `reference_wrapper` can be *reassigned*, but it cannot "hold reference to more than one objects". 2/3: Fair point about where `.get()` is appropriate - *but* unsuffixed `r` can be used the same as `T&` in cases where `r`'s conversion operator can be invoked unambiguously - so no need to call `.get()` in many cases, including several in your code (which is difficult to read due to lack of spaces). – [underscore_d](#) Dec 11 '15 at 17:00

@[underscore_d](#) `reference_wrapper` can hold an array of references if you are not sure then you can try it yourself. Plus `.get()` is used when you want to change the value of the object the `reference_wrapper` is holding ie `r=70` is illegal so you have to use `r.get()=70`. Try yourself !!!!! – [Ankit Acharya](#) Dec 12 '15 at 14:03

Show me a single `reference_wrapper` holding more than one reference. – [underscore_d](#) Dec 12 '15 at 14:55

cpp.sh/37kkb Check this out !!! – [Ankit Acharya](#) Dec 12 '15 at 20:12

- 1 @AnkitAcharya Yes :-) but to be precise, *effective* result aside, *itself* only refers to one object. Anyway, you're of course right that unlike a normal ref, the `wrapper` can go in a container. This is handy, but I think people misinterpret this as more advanced than it really is. If I want an array of 'refs', I usually skip the middleman with `vector<Item*>`, which is what the `wrapper` boils down to... and hope the anti-pointer purists don't find me. The convincing use-cases for it are different and more complex. – [underscore_d](#) Dec 13 '15 at 10:45

Want to work remotely?

Find remote jobs based on technology

Get started

`std::reference_wrapper` is recognized by standard facilities to be able to pass objects by reference in pass-by-value contexts.

For example, `std::bind` can take in the `std::ref()` to something, transmit it by value, and unpacks it back into a reference later on.

```
void print(int i) {
    std::cout << i << '\n';
}

int main() {
    int i = 10;

    auto f1 = std::bind(print, i);
    auto f2 = std::bind(print, std::ref(i));

    i = 20;

    f1();
    f2();
}
```

This snippet outputs :

```
10
20
```

The value of `i` has been stored (taken by value) into `f1` at the point it was initialized, but `f2` has kept an `std::reference_wrapper` by value, and thus behaves like it took in an `int&`.

edited Oct 20 '15 at 15:57

answered Oct 20 '15 at 15:48



Quentin

38.3k 6 125

Up

OR SIGN IN WITH



Join Stack Overflow to learn, share knowledge, and build your career.

could you elaborate with some small code – CppNITR Oct 20 '15 at 15:50

1 @CppNITR sure ! Give me a moment to assemble a small demo :) – Quentin Oct 20 '15 at 15:51

1 what about the difference between T& & ref(T) – CppNITR Oct 20 '15 at 16:01

2 @CppNITR `std::ref(T)` returns a `std::reference_wrapper`. It's little more than a wrapped pointer, but is recognize by the library as "hey, I'm supposed to be a reference ! Please turn me back into one once you're done passing me around". – Quentin Oct 20 '15 at 16:32

A reference (`T&` or `T&&`) is a special element in C++ language. It allows to manipulate an object *by reference* and has special use cases in the language. For example, you cannot create a standard container to hold references: `vector<T&>` is ill formed and generates a compilation error.

A `std::reference_wrapper` on the other hand is a C++ object able to hold a reference. As such, you can use it in standard containers.

`std::ref` is a standard function that returns a `std::reference_wrapper` on its argument. In the same idea, `std::cref` returns `std::reference_wrapper` to a const reference.

One interesting property of a `std::reference_wrapper`, is that it has an operator `T& () const noexcept`; . That means that **even if it is a true object**, it can be automatically converted to the reference that it is holding. So:

- as it is a copy assignable object, it can be used in containers or in other cases where references are not allowed
- thanks to its operator `T& () const noexcept`; , it can be used anywhere you could use a reference, because it will be automatically converted to it.

edited Oct 20 '15 at 17:50

answered Oct 20 '15 at 16:09



Serge Ballesta

65.7k 9 49 113