why does the array decay to a pointer in a template function

Asked 10 years, 5 months ago Modified 6 years, 1 month ago Viewed 3k times

I don't understand why the array decays to a pointer in a template function.

If you look at the following code: When the parameter is forced to be a reference (function f1) it does not decay. In the other function f it decays. Why is the type of T in function f not const char (buff&)[3] but rather const char* (if I understand it correctly)?

```
#include <iostream>

template <class T>
void f(T buff) {
    std::cout << "f:buff size:" << sizeof(buff) << std::endl;       //prints 4
}

template <class T>
void f1(T& buff) {
    std::cout << "f:buff size:" << sizeof(buff) << std::endl;       //prints 3
}

int main(int argc, char *argv[]) {
    const char buff[3] = {0,0,0};
    std::cout << "buff size:" << sizeof(buff) << std::endl;         //prints 3
    f(buff);
    f1(buff);
    return 0;
}
```

c++ arrays templates sizeof

Share Edit Follow Flag

asked Oct 17, 2011 at 18:20

David Feurle
2,657 • 21 • 36

If you simply passed an `int` to `f`, then `T` would be `int`, not `int&`. Therefore, you should be asking something like "Why is the type of T in function f not `const char [3]` but rather `const char*`?" (note the missing `&` compared to your answer) – Aaron McDaid Jul 24, 2015 at 13:53

... (follow on from my last comment). The stupidest thing about the C/C++ language is that if you put `const char [3]` in your parameters, the compiler will silently rewrite it as a `const char *`. This doesn't happen with local variables, for example. I really think this should lead to warnings nowadays (from C++ compilers at least) – Aaron McDaid Jul 24, 2015 at 14:14

5 Answers

Sorted by: Highest score (default)

15

It is because *arrays* cannot be passed *by value* to a function. So in order to make it work, the array decays into a pointer which then gets passed to the function *by value*.

In other words, passing an array by value is akin to *initializing* an array with another array, but in C++ *an array* cannot be *initialized* with another array:

```
char buff[3] = {0,0,0};
char x[3] = buff; //error
```

So if an array appears on the right hand side of `=`, the left hand side has to be either `pointer` or `reference` type:

```
char *y = buff; //ok - pointer
char (&z)[3] = buff; //ok - reference
```

Demo : http://www.ideone.com/BIfSv

It is exactly for the same reason `auto` is inferred differently in each case below (note that `auto` comes with C++11):

```
auto a = buff; //a is a pointer - a is same as y (above)
std::cout << sizeof(a) << std::endl; //sizeof(a) == sizeof(char*)

auto & b = buff; //b is a reference to the array - b is same as z (above)
std::cout << sizeof(b) << std::endl; //sizeof(b) == sizeof(char[3])
```

Output:

```
4 //size of the pointer
3 //size of the array of 3 chars
```

Demo : http://www.ideone.com/aXcF5

Share Edit Follow Flag

edited Feb 24, 2016 at 18:33

Patryk
20.2k • 38 • 115 • 225

answered Oct 17, 2011 at 18:39

Nawaz
340k • 110 • 645 • 828

10

Because arrays can not be passed by value as a function parameter.
When you pass them by value they decay into a pointer.

In this function:

```
template <class T>
void f(T buff) {
```

T can not be `char (&buff)[3]` as this is a reference. The compiler would have tried `char (buff)[3]` to pass by value but that is not allowed. So to make it work arrays decay to pointers.

Your second function works because here the array is passed by reference:

```
template <class T>
void f1(T& buff) {

// Here T& => char (&buff)[3]
```

Share Edit Follow Flag

edited Oct 17, 2011 at 18:27

David Rodríguez - dribeas
198k • 21 • 283 • 478

answered Oct 17, 2011 at 18:23

Martin York
245k • 82 • 318 • 541

I believe the reason they cannot be passed by value is related to the lack of copy/assignment. (Though why that's all missing is beyond me) – Mooing Duck Oct 17, 2011 at 18:33

@MooingDuck: In C++ the reason is that the particular behavior was inherited from C. In C the reason would be different, of course... – David Rodríguez - dribeas Oct 17, 2011 at 18:37

2 @MooingDuck : And of course that's one of the things that makes `std::array<>` immediately superior to raw C-arrays. – ildjarn Oct 17, 2011 at 18:43

so the question is, we want to keep the `f(T stuff)` signature but force the template deduction to be of reference type. can we use `std::add_reference` or `boost::ref` or something, on the client side (call site), and then we have our pass-by-ref-as-original-array-type (not decayed) like we want ? – v.oddou Jun 18, 2015 at 6:21

1 @v.oddou: You could actually try this the code is relatively short. Also this has nothing to do withe the current question. But `std::add_reference` is not going to do anything for you (as it is just tmp and just allows you to define a type (which will be `char (&buff)[3]`). This still will not bind to the first function. But `boost::ref` will build an object of type `boost::reference_wrapper` which can be passed by value. – Martin York Jun 18, 2015 at 13:06

I

4

To quote from spec, it says

> (14.8.2.1/2) If P is not a reference type: — If A is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of A for type deduction; otherwise

So, in your case, It is clear that,

```
template <class T>
void f1(T& buff) {
    std::cout << "f:buff size:" << sizeof(buff) << std::endl;       //prints 3
}
```

doesn't decay into pointer.

Share Edit Follow Flag

answered Oct 17, 2011 at 18:31

cpx
16.1k • 19 • 83 • 139

1

The reason basically boils down to type deduction when matching the different overloads. When you call `f` the compiler deduces the type to be `const char[3]` which then decays into `const char*` because *that's what arrays do*. This is done in the same exact way that in `f(1)` the compiler deduces T to be `int` and not `int&`.

In the case of `f1` because the argument is taken by reference, then the compiler again deduces T to be `const char[3]`, but it takes a reference to it.

Nothing really surprising, but rather consistent if it were not for the *decay* of arrays to pointers when used as function arguments...

Share Edit Follow Flag

answered Oct 17, 2011 at 18:27

David Rodríguez - dribeas
198k • 21 • 283 • 478

1

Because functions can't have arrays as arguments. They can have array references though.

Share Edit Follow Flag

answered Oct 17, 2011 at 18:23

K-ballo
78.4k • 19 • 151 • 166