What is the purpose of std::make_pair vs the constructor of std::pair?

Ask Question

What is the purpose of `std::make_pair` ?

Why not just do `std::pair<int, char>(0, 'a')` ?

Is there any difference between the two methods?

c++ stl std-pair

edited Jan 6 '17 at 10:35

Ciro Santilli 新疆改造中心 六四事件 法轮功

116k 23 452 397

asked Feb 14 '12 at 1:37

user542687

5 In C++11, you

See my answer . – PlagueHammer Mar 19 '14 at 6:54

1 In C++17, `std::make_pair` is redundant. There is an answer below that details this. – Drew Dormann May 22 at 14:56

6 Answers

The difference is that with `std::pair` you need to specify the types of both elements, whereas `std::make_pair` will create a pair with the type of the elements that are passed to it, without you needing to tell it. That's

See this example from http://www.cplusplus.com/reference/std/utility/make_pair/

```
pair <int,in
pair <int,in

one = make_pa
two = make_pa
```

Aside from the implicit conversion bonus of it, if you didn't use make_pair you'd have to do

```
one = pair<in
```

every time you assigned to one, which would be annoying over time...

lited Feb 14 '12 at 1:51

swered Feb 14 '12 at 1:39

Tor Valamo

24.3k 8 62 75

Actually, the types should be deduced at compile time without the need

'12 at 1:51

@Tor Yeah, I know how to use both of them, I was just curious if there was a reason for `std::make_pair`. Apparently it is just for convenience. – user542687 Feb 14 '12 at 1:56

@Jay It would appear so. – Tor Valamo Feb 14 '12 at 1:58

13 I think you can do `one = {10, 20}` nowadays but I don't have a C++11 compiler handy to

4 Also note that `make_pair` works with unnamed types, including structs, unions, lambdas, and other doodads. – Mooing Feb 6 '15 at 22:29

As @MSalters replied above, you can now use curly braces to do this in C++11 (just verified this with a C++11 compiler):

```
pair<int, in
```

lited Mar 22 '14 at 21:06

Martin
6,505 5 26 30

swered Feb 24 '14 at 18:28

difference between using `make_pair` and explicitly calling the `pair` constructor with specified type arguments. `std::make_pair` is more convenient when the types are verbose because a template method has type deduction based on its given parameters. For example,

```
std::vector<
std::vector<

// shorter
vecOfPair.pu

 // longer
vecOfPair.pu
emptyV));
```

swered Feb 14 '12 at 1:49

devil

1,434 12 21

It's worth noting that this is a common idiom in C++ template programming

information and a nice example here.

Edit As someone suggested in the comments (since removed) the following is a slightly modified extract from the link in case it breaks.

An Object Generator allows creation of objects without explicitly specifying their types. It is based on a useful property of function templates which class templates don't have: The type parameters of a function template are deduced automatically from its actual parameters. `std::make_pair` is a simple

depending on the actual parameters of the `std::make_pair` function.

```
template <cl
std::pair <T
make_pair(T
{
  return std
}
```

lited Apr 3 '14 at 9:37

swered Mar 15 '13 at 14:35

mkm
450 4 17

I think your code is missing references `&` – Mooing Feb 6 '15 at 22:31

@duck Actually `&&` since C++11. – Justme(Jan 22 at 15:53

Class template arguments could not be

Before C++17 you could not write something like:

```
std::pair p(
```

since that would infer template types from the constructor arguments.

C++17 makes that syntax possible, and therefore `make_pair` redundant.

Before C++17, `std::make_pair` allowed us to write less verbose code:

```
MyLongClassNa
MyLongClassNa
auto p = std
```

instead of the more verbose:

```
std::pair<MyL
```

which repeats the types, and can be very long.

Type inference works in that

`make_pair` is essentially equivalent to:

```
template<cla
std::pair<T1
    return s
}
```

The same concept applies to `inserter` vs `insert_iterator`.

See also:

- Why not infer template parameter from constructor?
- https://en.wikibooks.org/wiki/More_C++_Idioms/Object_Generator

lited May 22 at 15:03

swered Jan 7 '17 at 12:26

Ciro Santilli 新疆改造中心 六四事件 法轮功

116k 23 452 397

make_pair creates an extra copy over the

simple syntax. This shows the difference (example by Rampal Chaudhary):

```
class Sample
{
    static i

    int _obj
public:
    Sample()
        _obj
    {
        std:
    }

    Sample( 
    _objectN
    {
        std:
    }

    ~Sample(
    {
        std:
    }
};
int Sample::_


int main(int
{
    Sample s
    std::map

    map.inse
    //map.in
    return 0
}
```

lited Nov 28 '13 at 14:34

swered Nov 28 '13 at 13:40

EmpZoooli
41 2

3 I am pretty sure that

if the optimization settings of the compiler are high enough. – Björn Pc Feb 19 '14 at 13:00

Why would you ever want to rely on compiler optimizations for correctness? – sjbx Dec 2 '16 at 8:46

I get the same results with both versions, and with `std::move` just inside `insert` and/or around what would be a reference to `sample`. It is only

`std::map<int,Sample const &>` that I reduce the number of constructed objects, and only when I delete the copy constructor that I eliminate all copies (obviously). After making both of those changes, my result includes one call to the default constructor and two calls to the destructor for the same object. I think I must be missing something.

'17 at 23:13

FWIW I agree that optimization and correctness should be completely independent, as this is exactly the kind of code you write as a sanity check after different optimization levels produce inconsistent results. In general I would recommend `emplace` instead of `insert` if you're just constructing a value to insert

instances.) It's not my area of expertise, if I can even say I have one, but the copy/move semantics introduced by C++11 have helped me a lot. – John P Aug 30 '17 at 23:31