

13.12 — Class template argument deduction (CTAD) and deduction guides

 ALEX  JANUARY 11, 2024

Class template argument deduction (CTAD) C++17 (#CTAD)

Starting in C++17, when instantiating an object from a class template, the compiler can deduce the template types from the types of the object's initializer (this is called **class template argument deduction** or **CTAD** for short). For example:

```
#include <utility> // for std::pair

int main()
{
    std::pair<int, int> p1{ 1, 2 }; // explicitly specify class template std::pair<int, int> (C++11 onward)
    std::pair p2{ 1, 2 };          // CTAD used to deduce std::pair<int, int> from the initializers (C++17)

    return 0;
}
```

CTAD is only performed if no template argument list is present. Therefore, both of the following are errors:

```
#include <utility> // for std::pair

int main()
{
    std::pair<> p1 { 1, 2 }; // error: too few template arguments, both arguments not deduced
    std::pair<int> p2 { 3, 4 }; // error: too few template arguments, second argument not deduced

    return 0;
}
```

Author's note

Many future lessons on this site make use of CTAD. If you're compiling these examples using the C++14 standard (or older), you'll get an error about missing template arguments. You'll need to explicitly add such arguments to the example to make it compile.

Since CTAD is a form of type deduction, we can use literal suffixes to change the deduced type:

```
#include <utility> // for std::pair

int main()
{
    std::pair p1 { 3.4f, 5.6f }; // deduced to pair<float, float>
    std::pair p2 { 1u, 2u };     // deduced to pair<unsigned int, unsigned int>

    return 0;
}
```

Template argument deduction guides C++17 (#DeductionGuide)



In most cases, CTAD works right out of the box. However, in certain cases, the compiler may need a little extra help understanding how to deduce the template arguments properly.

You may be surprised to find that the following program (which is almost identical to the example that uses `std::pair` above) doesn't compile in C++17 (only):

```
// define our own Pair type
template <typename T, typename U>
struct Pair
{
    T first{};
    U second{};
};

int main()
{
    Pair<int, int> p1{ 1, 2 }; // ok: we're explicitly specifying the template arguments
    Pair p2{ 1, 2 };          // compile error in C++17 (okay in C++20)

    return 0;
}
```

If you compile this in C++17, you'll likely get some error about “class template argument deduction failed” or “cannot deduce template arguments” or “No viable constructor or deduction guide”. This is because in C++17, CTAD doesn't know how to deduce the template arguments for aggregate class templates. To address this, we can provide the compiler with a **deduction guide**, which tells the compiler how to deduce the template arguments for a given class template.

Here's the same program with a deduction guide:

```
template <typename T, typename U>
struct Pair
{
    T first{};
    U second{};
};

// Here's a deduction guide for our Pair (needed in C++17 only)
// Pair objects initialized with arguments of type T and U should deduce to Pair<T, U>
template <typename T, typename U>
Pair(T, U) -> Pair<T, U>;

int main()
{
    Pair<int, int> p1{ 1, 2 }; // explicitly specify class template Pair<int, int> (C++11 onward)
    Pair p2{ 1, 2 };          // CTAD used to deduce Pair<int, int> from the initializers (C++17)

    return 0;
}
```

This example should compile under C++17.



The deduction guide for our `Pair` class is pretty simple, but let's take a closer look at how it works.

```
// Here's a deduction guide for our Pair (needed in C++17 only)
// Pair objects initialized with arguments of type T and U should deduce to Pair<T, U>
template <typename T, typename U>
Pair(T, U) -> Pair<T, U>;
```

First, we use the same template type definition as in our `Pair` class. This makes sense, because if our deduction guide is going to tell the compiler how to deduce the types for a `Pair<T, U>`, we have to define what `T` and `U` are (template types). Second, on the right hand side of the arrow, we have the type that we're helping the compiler to deduce. In this case, we want the compiler to be able to deduce template arguments for objects of type `Pair<T, U>`, so that's exactly what we put here. Finally, on the left side of the arrow, we tell the compiler what kind of declaration to look for. In this case, we're telling it to look for a declaration of some object named `Pair` with two arguments (one of type `T`, the other of type `U`). We could also write this as `Pair(T t, U u)` (where `t` and `u` are the names of the parameters, but since we don't use `t` and `u`, we don't need to give them names).

Putting it all together, we're telling the compiler that if it sees a declaration of a `Pair` with two arguments (of types `T` and `U` respectively), it should deduce the type to be a `Pair<T, U>`.

So when the compiler sees the definition `Pair p2{ 1, 2 };` in our program, it will say, "oh, this is a declaration of a `Pair` and there are two arguments of type `int` and `int`, so using the deduction guide, I should deduce this to be a `Pair<int, int>`".

Here's a similar example for a `Pair` that takes a single template type:

...



```
template <typename T>
struct Pair
{
    T first{};
    T second{};
};

// Here's a deduction guide for our Pair (needed in C++17 only)
// Pair objects initialized with arguments of type T and T should deduce to Pair<T>
template <typename T>
Pair(T, T) -> Pair<T>;

int main()
{
    Pair<int> p1{ 1, 2 }; // explicitly specify class template Pair<int> (C++11 onward)
    Pair p2{ 1, 2 };     // CTAD used to deduce Pair<int> from the initializers (C++17)

    return 0;
}
```

In this case, our deduction guide maps a `Pair(T, T)` (a `Pair` with two arguments of type `T`) to a `Pair<T>`.

Tip

C++20 added the ability for the compiler to automatically generate deduction guides for aggregates, so deduction guides should only need to be provided for C++17 compatibility.

Because of this, the version of `Pair` without the deduction guides should compile in C++20.

`std::pair` (and other standard library template types) come with pre-defined deduction guides, which is why our example above that uses `std::pair` compiles fine in C++17 without us having to provide deduction guides ourselves.

For advanced readers

Non-aggregates don't need deduction guides in C++17 because the presence of a constructor serves the same purpose.

Type template parameters with default values

Just like function parameters can have default arguments, template parameters can be given default values. These will be used when the template parameter isn't explicitly specified and can't be deduced.

Here's a modification of our `Pair<T, U>` class template program above, with type template parameters `T` and `U` defaulted to type `int`:

```
template <typename T=int, typename U=int> // default T and U to type int
struct Pair
{
    T first{};
    U second{};
};

template <typename T, typename U>
Pair(T, U) -> Pair<T, U>;

int main()
{
    Pair<int, int> p1{ 1, 2 }; // explicitly specify class template Pair<int, int> (C++11 onward)
    Pair p2{ 1, 2 };          // CTAD used to deduce Pair<int, int> from the initializers (C++17)

    Pair p3;                  // uses default Pair<int, int>

    return 0;
}
```

Our definition for `p3` does not explicitly specify types for the type template parameters, nor is there an initializer for these types to be deduced from. Therefore, the compiler will use the types specified in the defaults, which means `p3` will be of type `Pair<int, int>`.



CTAD doesn't work with non-static member initialization

When initializing the member of a class type using non-static member initialization, CTAD will not work in this context. All template arguments must be explicitly specified:

```
#include <utility> // for std::pair

struct Foo
{
    std::pair<int, int> p1{ 1, 2 }; // ok, template arguments explicitly specified
    std::pair p2{ 1, 2 };          // compile error, CTAD can't be used in this context
};

int main()
{
    std::pair p3{ 1, 2 };          // ok, CTAD can be used here
    return 0;
}
```

CTAD doesn't work with function parameters

CTAD stands for class template argument deduction, not class template parameter deduction, so it will only deduce the type of template arguments, not template parameters.

Therefore, CTAD can't be used in function parameters.

```
#include <iostream>
#include <utility>

void print(std::pair p) // compile error, CTAD can't be used here
{
    std::cout << p.first << ' ' << p.second << '\n';
}

int main()
{
    std::pair p { 1, 2 }; // p deduced to std::pair<int, int>
    print(p);

    return 0;
}
```

In such cases, you should use a template instead:

```
#include <iostream>
#include <utility>

template <typename T, typename U>
void print(std::pair<T, U> p)
{
    std::cout << p.first << ' ' << p.second << '\n';
}

int main()
{
    std::pair p { 1, 2 }; // p deduced to std::pair<int, int>
    print(p);

    return 0;
}
```



Next lesson

13.13 [Alias templates](#)



[Back to table of contents](#)



Previous lesson

13.11 [Class templates](#)

Leave a comment...



Name*



Email*




Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!

 Avatars from <https://gravatar.com/> are connected to your provided email address.

We and our partners share information on your use of this website to help improve your experience.



Do not sell my info: ☐

OKAY