What's up with std::piecewise_construct and std::forward_as_tuple?

devblogs.microsoft.com/oldnewthing/20220428-00

April 28, 2022

Raymond Chen

There is this curious marker type in the C++ standard library called `piecewise_construct_t`, and an inline variable `piecewise_construct` that produces it. What's the deal with this guy, and his friend `std::forward_as_tuple`?

STL explained it to me.

We'll start with `std::forward_as_tuple`: This takes its arguments and produces a tuple of corresponding references.

```
int x;

// produces std::tuple<int&, std::string&&>
std::forward_as_tuple(x, std::string(L"hello"));
```

Note that this potentially contains a tuple of rvalue references, which means that you had better use the tuple before the end of the statement, before the temporaries are destroyed. If you don't, then you have a use-after-free bug:

```
auto values = std::forward_as_tuple(x, std::string(L"hello"));
std::get<1>(values); // dangling reference to destructed string
```

The purpose of `std::forward_as_tuple` is to wrap up a bunch of parameters, probably a parameter pack, into a single object that can be passed as a single non-pack parameter. This is handy if you need to work with a pattern that accepts only a single parameter, so you use the tuple as a way to aggregate multiple objects into one.

```
template<typename...Args>
auto f(Args&&...args)
{
    auto t = std::forward_as_tuple(std::forward<Args>(args)...);
    ... do something with t ...
}
```

A case where you may want to wrap up a bunch of parameters into a single parameter is if you need to pass *multiple* groups of parameters. The C++ language doesn't let you write

```
template<typename...Group1,
         typename...Group2>
void something(Group1&&... group1,
               Group2&&... group2)
{
    ...
}
```

because there's no way to know where the first group ends and the second group begins. But you can do this if you pass the parameters as perfectly-forwarded tuples of references.

```
template<typename...Group1,
         typename...Group2>
void something(std::tuple<Group1&&...> group1,
               std::tuple<Group2&&...> group2)
{
    ...
}
```

This is where `std::piecewise_construct` enters the story.

The `std::pair` wants to let you construct the components in place. The existing two-parameter constructor tries to construct the first element from the first parameter and the second element from the second parameter:

```
struct T1
{
    T1(int);
    T1(int, std::string);
};

struct T2
{
    T2(std::string);
    T2();
};

auto f()
{
    // uses T1(a1) and T2(a2)
    return std::pair<T1, T2>(a1, a2);
}
```

If you want to use a `T1` or `T2` constructor that takes any number of parameters other than one, you can pack them into tuples, and use the marker value `std::piecewise_construct` to say, "Hey, like, don't pass the tuples as-is to the constructors. Instead, unpack the tuples and invoke the constructors with the tuple elements."

```
auto f()
{
    // without piecewise_construct: tries to use
    //      T1(std::tuple<int, char const*>(42, "hello"))
    // and  T2(std::tuple<>())
    // (neither of which works)
    return std::pair<T1, T2>(
        std::make_tuple(42, "hello"),
        std::make_tuple());
}

auto f()
{
    // with piecewise_construct:
    // uses T1(42, "hello")
    // and  T2()
    return std::pair<T1, T2>(
        std::piecewise_construct,
        std::make_tuple(42, "hello"),
        std::make_tuple());
}
```

The `T1` and `T2` are constructed in place directly into the pair, and in this case, it means that they are constructed in place directly into the return value (due to copy elision).

Now, you could have done this without `std::piecewise_construct`:

```
// uses T1(42, "hello") with T1(T1 const&)
// and  T2()            with T2(T2 const&)
return std::pair<T1, T2>({ 42, "hello" }, {});
```

but this does not construct the `T1` and `T2` objects in place. It uses the `pair(T1 const&, T2 const&)` constructor. That means that it creates a temporary `T1(42, "hello")` and passes it as a `T1 const&` to `T1`'s constructor, which will copy it. Similarly, it creates a temporary `T2()` object, and then copies the temporary to the pair's `T2`.

Not only is it wasteful, but it also requires that `T1` and `T2` be copyable, which may not be possible for the `T1` and `T2` you need.

If your `T1` and `T2` are at least movable, you could use

```
// uses T1(42, "hello") with T1(T1 &&)
// and  T2()            with T2(T2 &&)
return std::pair<T1, T2>(T1{ 42, "hello" }, T2{});
```

That saves you a copy, but it's not quite the same as just constructing in place from the original parameters.

The `std::piecewise_construct` marker works well in conjunction with `std::forward_as_tuple`:

```
template<typename...Args>
auto make_t1_with_default_t2(Args&&...args)
{
    return std::pair<T1, T2>(
        std::piecewise_construct,
        std::forward_as_tuple(std::forward<Args>(args)...),
        std::make_tuple());
}
```

The `std::piecewise_construct` marker pattern propagates into all of the `emplace` methods, since emplacement is built out of the constructor.

```
std::vector<std::pair<T1, T2>> v;

template<typename...Args>
auto add_t1_with_default_t2(Args&&...args)
{
    v.emplace_back(
        std::piecewise_construct,
        std::forward_as_tuple(std::forward<Args>(args)...),
        std::make_tuple());
}
```

In particular, you're likely to use it with `std::map::emplace`, since that takes a pair.

```
std::map<T1, T2> m;

m.emplace(
    std::piecewise_construct,
    std::make_tuple(42, "hello"),
    std::make_tuple());
```

Now you won't be scared when you see `std::piecewise_construct`. It's just a marker that means, "I'm going to construct multiple things, and the constructor parameters are provided as tuples, so you know where one set of parameters ends and the next one begins."

Raymond Chen

Follow