Structured binding may be the new hotness, but we'll always have std::tie

devblogs.microsoft.com/oldnewthing/20200925-00

September 25, 2020

Raymond Chen

C++17 introduced structured binding, which lets you assign an expression to multiple variables.

```
auto [a,b] = std::pair(1, "hello");
// int a = 1
// char const* b = "hello"
```

However, this is for creating new variables to hold the result. If you want to assign the result to existing variables, then you can use the old standby `std::tie`.

```
int a;
char const* b;
std::tie(a, b) = std::pair(1, "hello");
```

This comes in handy in C++/WinRT if you have a `winrt::com_array<T>` and you need to return it in its ABI form of a `uint32_t` coupled with a `T*`.

```
winrt::com_array<int32_t> CalculateResult();

HRESULT GetInt32Array(uint32_t* size, int32_t** value) try
{
  *size = 0;
  *value = nullptr;
  std::tie(*size, *value) = winrt::detach_abi(CalculateResult());
  return S_OK;
}
catch (...) { return winrt::to_hresult(); }
```

When applied to a `com_array`, the `detach_abi` function returns a `std::pair` representing the size of the conformant array and a pointer to the start of the array. This is a form ready to be assigned to the tie of the two output parameters.

The type of the pointer part of the return value of `detach_abi(com_array<T> a)` is a pointer to the C++/WinRT ABI representation of `T`. Here are some examples:

T	detach_abi(com_array<T>) returns
int32_t	std::pair<uint32_t, int32_t*>
hstring	std::pair<uint32_t, void**>
ISomething	std::pair<uint32_t, mystery_abi*>

- If you have a `com_array` of a scalar type, then you will get a pointer to a conformant array of that scalar type.
- If you have a `com_array` of a string type, then you will get a pointer to a conformant array of `void*`.
- If you have a `com_array` of a reference type, then you will get a pointer to a conformant array of mystery pointers.

In the last case, you should just treat the resulting pointer as if it were a `void**`.

```
HRESULT GetNames(uint32_t* size, HSTRING** value) try
{
  *size = 0;
  *value = nullptr;
  std::tie(*size, reinterpret_cast<void*&>(*value)) =
    winrt::detach_abi(CalculateNames());
  return S_OK;
}
catch (...) { return winrt::to_hresult(); }
HRESULT GetSomethingArray(uint32_t* size, ISomething*** value) try
{
  *size = 0;
  *value = nullptr;
  std::tie(*size, reinterpret_cast<void*&>(*value)) =
    winrt::detach_abi(CalculateSomethings());
  return S_OK;
}
catch (...) { return winrt::to_hresult(); }
```

Note that in both cases we reinterpret-cast the output pointer to just `void*`. Any pointer type can be assigned to `void*`, so we just use that to soak up the C++/WinRT ABI pointer, without needing to know what it actually is.[1]

[1] The C++/WinRT ABI requires that all data pointers have the same size and representation, so this sort of type pun is legal from an ABI point of view.

Raymond Chen

Follow