Inside STL: The pair and the compressed pair

devblogs.microsoft.com/oldnewthing/20230801-00

August 1, 2023

Raymond Chen

The C++ language comes with a standard library. When you're debugging your C++ code, you may have to go digging inside the implementation to extract information from crash dumps. This mini-series is going to look at how various C++ standard library types are implemented by the three major implementations (clang, gcc, and msvc).

We'll start with the lowly `std::pair`. Its definition is quite simple.

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};
```

The names of the members of `std::pair` are required by the C++ language standard, so you don't see any variation here. Here's how it looks in the Windows debugger:

```
0:000> ?? t
struct std::pair<int,int>
   +0x000 first            : 0n42
   +0x004 second           : 0n99
```

Compressed pairs are a fancy kind of pair that are used internally by C++ library implementations. Empty classes are often used in the C++ standard library: Allocators, hash objects, comparison objects. The C++ language requires that even empty classes have a nonzero size if they are standalone objects or members of other classes.

Rather than wasting a byte per object to hold these empty classes, the standard library implementation uses a trick: The C++ language does not require empty *base classes* to have a nonzero size, provided the base class's type does not match that of the first member of the class.[1] Treating empty base classes as having zero size is known as the *empty base optimization* or EBO.

Okay, so suppose you want a class that acts like a `std::pair<T1, T2>` where `T1` is an empty class. Instead of the naïve definition

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};
```

you could use this:

```
template<typename T1, typename T2>
struct fancy_pair_t1 : T1
{
    T2 second;
};
```

If `T1` is an empty class, then using it as a base class allows the compiler to shrink it down to zero bytes. Similarly, if you suspect the second type of being an empty class, you can derive from `T2` and keep `T1` as a member.

Inside the standard library implementation, there's code that determines at compile time whether to use `fancy_pair_t1`, `fancy_pair_t2`, or a traditional `pair`.

Just a reminder that the compressed pair is not part of the official C++ language standard library. It's a helper class that is used internally by standard library implementations to squeeze out wasted bytes.

Bonus chatter: The Boost library comes with an implementation of `compressed_pair`.

Bonus bonus chatter: The introduction of the `[[no_unique_address]]` attribute in C++20 avoids the need for compressed pairs in most if not all cases. You can get the same effect as an old-style compressed pair with the much simpler

```
template<typename T1, typename T2>
struct compressed_pair
{
    [[no_unique_address]] T1 first;
    [[no_unique_address]] T2 second;
};
```

Nevertheless, C++ standard library implementations continue to use their internal compressed pair types. There are a variety of reasons for this.

- It allows the same header to be used by both C++17 and C++20 clients.
- It preserves ABI compatibility.
- There is a general reluctance to make changes to working code without a concrete benefit.

[1] If the first member of the class has the same type as the base class, then you would have the problem of two separate objects of the same type at the same address.