Inside STL: The map, set, multimap, and multiset

devblogs.microsoft.com/oldnewthing/20230807-00

August 7, 2023

Raymond Chen

The complexity requirements of the C+ standard library `map`, `set`, `multimap`, and `multiset` basically narrow the possible implementation options to AVL trees and red-black trees.

In practice, everybody seems to choose a red-black tree, with an explicit sentinel node. The sentinel node also doubles as a header. The structure of the tree is as follows:

```
struct tree
{
    node_base header; // or "node_base* header;"
    size_t size;
};

struct node_base
{
    node_base* parent;
    node_base* left;
    node_base* right;
};

struct node : node_base
{
    bool color; // red or black
    payload data;
};
```

The tree remembers the number of elements it holds (because `size()` is required to be *O*(1)) and has a sentinel node, called the “header”. The sentinel node is either embedded in the tree object or is a separately-allocated node, depending on the implementation.

The sentinel node uses its pointers in an unusual manner.

- `parent` points to the root of the tree.
- `left` points to the smallest item in the tree.
- `right` points to the largest item in the tree.

If you follow the `header.parent` to the root of the tree, then what you find is a standard red-black binary tree, with the `left` node pointing to the subtree of smaller elements, the `right` node pointing to the subtree of larger elements, and the `parent` node pointing to the parent element.

The header node is used to represent `end()`, the one-past-the-end iterator.

For a `map` or `multimap`, the payload is a `std::pair<Key, T>`. For a `set` or `multiset`, the payload is a `Key`.

Here's an example of how a `map` is laid out in memory:

```
std::map<int, std::string> m = {
    { 42, "forty-two" },
    { 99, "ninety-nine" },
};
```

First, with an internal header node:

m			A				
header.parent	A	↔	parent	m.header			
header.left	A		left			B	
header.right	B		right	B	↔	parent	A
size	2		color	black		left	
			data	42, `"forty-two"`		right	
						color	red
						data	99, `"ninety-nine"`

And again with an external header node:

m			H			A				
header	H	→	parent	A	↔	parent	H			
size	2		left	A		left			B	
			right	B		right	B	↔	parent	A
						color	black		left	

	data	42, `"forty-two"`	right	
			color	red
			data	99, `"ninety-nine"`

The map object itself has two members: The `header` is (or points to) our sentinel/header node `H`, and the `size` is 2, remembering that we have two nodes in our tree.

The header node's `parent` points to the root of the tree `A`. This is confusing at first, because the member named "parent" actually points to the "child", if you think of the header node as the "super-parent" of the entire tree. The header node's `left` and `right` members point to the smallest and largest nodes of the tree, respectively. In our case, the smallest node is `A` (42) and the largest node is `B` (99). Finally, the `color` and `data` of the header node are not used.

The root of our tree is the node `A`. It remembers that its parent is the header node `H`, but the interesting parts are the `left` and `right` pointers. There is no left subtree, so the `left` is null; the right subtree is rooted at `B`. Finally, the node also has a `color` (black in this case), and its `data` holds the payload.

The last node in the tree is `B`, representing the single-node right subtree of `A`. Its parent is `A`, and its `left` and `right` nodes are null because it is a leaf node. Node `B` also has a `color` (red in this case), and its `data` holds the payload.

The tree also needs a comparer and allocator. Those classes are typically empty, so the tree saves them as compressed pairs with the rest of the data. (Since there are two usually-empty things, you have a compressed pair inside a compressed pair.)

The Microsoft implementation uses an external header sentinel node, whereas clang and gcc use an internal header sentinel node. The Microsoft implementation also has these extra quirks:

- There is an additional member of each node called `_Isnil` that is `true` for the header node and that is `false` for everybody else.
- Everywhere you see null in the above diagram, the Microsoft implementation substitutes a pointer to the header node.

The `_Isnil` member is technically redundant: You can detect that you have the header node by comparing it to the map's header. However, the Visual C++ compiler uses a raw node pointer as its iterator, which means that it doesn't have access to the containing tree to compare against the header. The information has to be internal to the node.

The Visual Studio debugger contains a visualizer to view the contents of a tree-based container more conveniently, but if you need to dig out the contents manually, here's how you can do it with the Microsoft implementation of the standard library. I've left in the template explosion to make it look more like what you're likely to encounter in the debugger.

```
0:000> ?? m
class std::map<int,std::basic_string<char,std::char_traits<char>,std::allocator<char>
>,std::less<int>,std::allocator<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > > >
   +0x000 _Mypair          :
std::_Compressed_pair<std::less<int>,std::_Compressed_pair<std::allocator<std::_Tree_n
 const ,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void
*> >,std::_Tree_val<std::_Tree_simple_types<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > > >,1>,1>
```

The full name for the `std::map<int, std::string>` is this horrible monstrosity because `std::string` is an alias for `std::basic_string<char>`, which is in turn a shorthand for `std::basic_string<char, std::char_traits<char>, std::allocator<char>>`. So everywhere you see the huge type `std::basic_string<char, std::char_traits<char>, std::allocator<char>>`, remember that's just a `std::string`.

On top of that, the `std::map` itself comes with a default comparer of `std::less<int>` and a default allocator of `std::allocator<std::pair<int, std::string>>`. But wait, `std::string` is itself an alias for that huge type name, so the default allocator's full name is `std::allocator<std::pair<int, std::basic_string<char, std::char_traits<char>, std::allocator<char>>>>`. And that's why when you declare a `std::map<int, std::string>`, you get this ridiculous type name in the debugger. That's the name the compiler has to deal with.

Okay, so we have the map. We need to dig through the compressed pairs to find the header pointer and size.

```
0:000> ?? m._Mypair
class
std::_Compressed_pair<std::less<int>,std::_Compressed_pair<std::allocator<std::_Tree_n
 const ,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void
*> >,std::_Tree_val<std::_Tree_simple_types<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > > >,1>,1>
   +0x000 _Myval2          :
std::_Compressed_pair<std::allocator<std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
>,std::_Tree_val<std::_Tree_simple_types<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > > >,1>
0:000> ?? m._Mypair._Myval2
class std::_Compressed_pair<std::allocator<std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
>,std::_Tree_val<std::_Tree_simple_types<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > > >,1>
   +0x000 _Myval2          : std::_Tree_val<std::_Tree_simple_types<std::pair<int
const ,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > > >
0:000> ?? m._Mypair._Myval2._Myval2
class std::_Tree_val<std::_Tree_simple_types<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > > > >
   +0x000 _Myhead          : 0x0000019a`d09cc720 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x008 _Mysize          : 2
```

We follow the header pointer to find the header node.

```
0:000> ?? m._Mypair._Myval2._Myval2._Myhead
struct std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *> *
0x0000019a`d09cc720
   +0x000 _Left            : 0x0000019a`d09c91f0 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x008 _Parent          : 0x0000019a`d09c91f0 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x010 _Right           : 0x0000019a`d09c9270 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x018 _Color           : 1 ''
   +0x019 _Isnil           : 1 ''
   +0x020 _Myval           : std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >
```

We can confirm that we're on the right track because the `_Isnil` is `true`, which is the case only for the header node. Follow the `_Parent` to get to the root of the tree.

```
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent
struct std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *> *
0x0000019a`d09c91f0
   +0x000 _Left            : 0x0000019a`d09cc720 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x008 _Parent          : 0x0000019a`d09cc720 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x010 _Right           : 0x0000019a`d09c9270 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x018 _Color           : 1 ''
   +0x019 _Isnil           : 0 ''
   +0x020 _Myval           : std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >
```

The payload of the root node is hiding inside the `_Myval` pair, so we'll have to expand it out:

```
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval
struct std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >
   +0x000 first            : 0n42
   +0x008 second           :
std::basic_string<char,std::char_traits<char>,std::allocator<char> >
```

We see immediately that the key of the root node is 42. We're lucky that our map is keyed by `int`, which is a simple type. If not, we'd have had to dig into the members of the `first` to figure out what it is.

Let's demonstrate that by digging into the `second` of the pair, which holds the mapped value.

```
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval.second
class std::basic_string<char,std::char_traits<char>,std::allocator<char> >
   +0x000 _Mypair          :
std::_Compressed_pair<std::allocator<char>,std::_String_val<std::_Simple_types<char>
>,1>
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval.second._Mypair
class
std::_Compressed_pair<std::allocator<char>,std::_String_val<std::_Simple_types<char>
>,1>
   +0x000 _Myval2          : std::_String_val<std::_Simple_types<char> >
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval.second._Mypair._Myval2
class std::_String_val<std::_Simple_types<char> >
   +0x000 _Bx              : std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x010 _Mysize          : 9
   +0x018 _Myres           : 0xf
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent-
>_Myval.second._Mypair._Myval2._Bx
union std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x000 _Buf             : [16]  "forty-two"
   +0x000 _Ptr             : 0x77742d79`74726f66  "--- memory read error at address
0x77742d79`74726f66 ---"
   +0x000 _Alias           : [16]  "forty-two"
```

In this case, the mapped type is a `std::string`, so we use the string-dumping technique we learned about some time ago.

So we've found that the root of the tree is `{ 42, "forty-two" }`. The `_Left` member points back to the header, so there is no left subtree. The `_Right` member holds some other pointer value, so we have a right subtree. Let's explore it.

```
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Right
struct std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *> *
0x0000019a`d09c9270
   +0x000 _Left            : 0x0000019a`d09cc720 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x008 _Parent          : 0x0000019a`d09c91f0 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x010 _Right           : 0x0000019a`d09cc720 std::_Tree_node<std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >,void *>
   +0x018 _Color           : 0 ''
   +0x019 _Isnil           : 0 ''
   +0x020 _Myval           : std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >
0:000> ?? t._Mypair._Myval2._Myval2._Myhead->_Parent->_Right->_Myval
struct std::pair<int const
,std::basic_string<char,std::char_traits<char>,std::allocator<char> > >
   +0x000 first            : 0n99
   +0x008 second           :
std::basic_string<char,std::char_traits<char>,std::allocator<char> >
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Right-
>_Myval.second._Mypair._Myval2._Bx
union std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x000 _Buf             : [16]  "ninety-nine"
   +0x000 _Ptr             : 0x6e2d7974`656e696e  "--- memory read error at address
0x6e2d7974`656e696e ---"
   +0x000 _Alias           : [16]  "ninety-nine"
0:000>
```

When I'm dumping tree structures in the debugger at this low level, I generally ignore all the type information because it's so voluminous. I rely on the member variable names to tell me what I'm looking at. Here's what the above debug session looks like to me:

```
0:000> ?? m
class std::map<int,std::basic_string<char, ⟦ ... ⟧ > >
   +0x000 _Mypair       : ⟦ ... ⟧ « ugh, a wrapper »
0:000> ?? m._Mypair
class ⟦ ... ⟧
   +0x000 _Myval2       : ⟦ ... ⟧ « ugh, another wrapper »
0:000> ?? m._Mypair._Myval2
class ⟦ ... ⟧
   +0x000 _Myval2       : ⟦ ... ⟧ « ugh, another wrapper »
0:000> ?? m._Mypair._Myval2._Myval2
class ⟦ ... ⟧
   +0x000 _Myhead       : 0x0000019a`d09cc720 ⟦ ... ⟧ « yay, the header, end in c720
»
   +0x008 _Mysize       : 2
0:000> ?? m._Mypair._Myval2._Myval2._Myhead
struct ⟦ ... ⟧* 0x0000019a`d09cc720 « yup, c720, this is the header »
   ⟦ ... ⟧
   +0x008 _Parent       : 0x0000019a`d09c91f0 ⟦ ... ⟧ « here is the root node »
   ⟦ ... ⟧
```

And then we inspect the root node:

```
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent
struct ⟦ ... ⟧ * 0x0000019a`d09c91f0
   +0x000 _Left         : 0x0000019a`d09cc720 ⟦ ... ⟧ ⟪ this is c720, so no left
subtree ⟫
   ⟦ ... ⟧
   +0x010 _Right        : 0x0000019a`d09c9270 ⟦ ... ⟧ ⟪ there is a right subtree ⟫
   ⟦ ... ⟧
   +0x020 _Myval        : ⟦ ... ⟧
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval
struct ⟦ ... ⟧
   +0x000 first         : 0n42 ⟪ the key is 42 ⟫
   +0x008 second        : ⟦ ... ⟧ ⟪ the mapped value is hiding here ⟫
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval.second
class ⟦ ... ⟧
   +0x000 _Mypair       : ⟦ ... ⟧ ⟪ ugh, a wrapper ⟫
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval.second._Mypair
class ⟦ ... ⟧
   +0x000 _Myval2       : ⟦ ... ⟧ ⟪ ugh, another wrapper ⟫
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Myval.second._Mypair._Myval2
class ⟦ ... ⟧
   +0x000 _Bx           : ⟦ ... ⟧ ⟪ okay, the string data is in here ⟫
   ⟦ ... ⟧
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent-
>_Myval.second._Mypair._Myval2._Bx
union std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x000 _Buf          : [16]  "forty-two"
   +0x000 _Ptr          : 0x77742d79`74726f66  ⟪ garbage, ignore ⟫
    ⟦ ... ⟧
```

Next we follow the right subtree.

```
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Right
struct ⟦ ... ⟧ * 0x0000019a`d09c9270
   +0x000 _Left         : 0x0000019a`d09cc720 ⟦ ... ⟧ ⟪ this is c720, so no left
subtree ⟫
   ⟦ ... ⟧
   +0x010 _Right        : 0x0000019a`d09cc720 ⟦ ... ⟧ ⟪ this is c720, so no right
subtree ⟫
   ⟦ ... ⟧
   +0x020 _Myval        : ⟦ ... ⟧
0:000> ?? t._Mypair._Myval2._Myval2._Myhead->_Parent->_Right->_Myval
struct ⟦ ... ⟧
   +0x000 first         : 0n99 ⟪ the key is 99 ⟫
   +0x008 second        : ⟦ ... ⟧ ⟪ I learned the path to the _Bx from last time ⟫
0:000> ?? m._Mypair._Myval2._Myval2._Myhead->_Parent->_Right-
>_Myval.second._Mypair._Myval2._Bx
union ⟦ ... ⟧
   +0x000 _Buf          : [16]  "ninety-nine"
   +0x000 _Ptr          : 0x6e2d7974`656e696e  ⟦ ... ⟧ ⟪ garbage, ignore ⟫
   ⟦ ... ⟧
```

There, we walked the complete contents of a `std::map` in the low-level debugger. It's quite a hassle.

Bonus chatter: Iterating through the tree can be done by performing an incremental inorder walk of a binary tree. Observe that this means that the `operator++` on a tree iterator is not constant-time. However, it is *amortized* constant time, because each edge in the tree is traversed twice (once outgoing and once returning), and the number of edges is equal to the number of nodes minus one. Therefore, the number of steps to traverse the entire tree is $2n - 2$, which is $O(n)$, and therefore each individual `operator++` has amortized constant cost, as required by the standard.