Inside STL: The lists

devblogs.microsoft.com/oldnewthing/20230804-00

August 4, 2023

Raymond Chen

The C++ standard library type `list` represents a doubly-linked list, and `forward_list` is a singly-linked list. Fortunately, the implementations of both of these lists are pretty much what you expect.

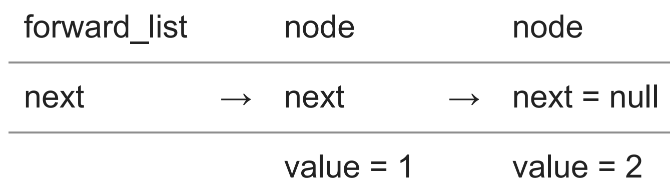
Let's start with the simpler `forward_list`.

```
template<typename T>
struct forward_list
{
    forward_list_node<T>* head;
};

template<typename T>
struct forward_list_node
{
    forward_list_node<T>* next;
    T value;
};
```

The `forward_list` itself is a pointer to the first element of the list, or `nullptr` if the list is empty. Each subsequent element contains a pointer to the next element, or `nullptr` if there is no next element.

For example, a two-element list of integers 1 and 2 is laid out like this:



forward_list		node		node
next	→	next	→	next = null
		value = 1		value = 2

The doubly-linked list `list` is also pretty simple.

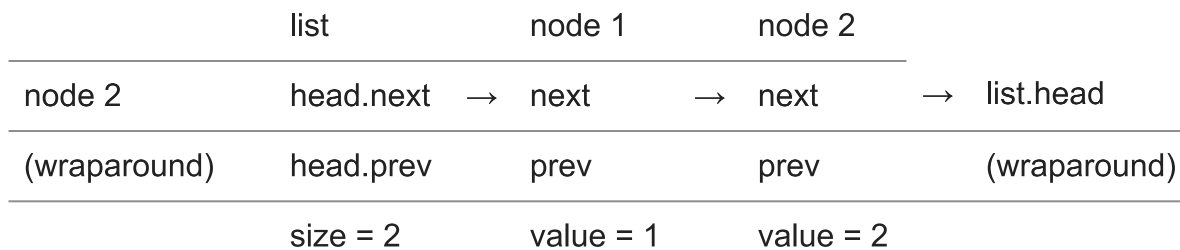
```
template<typename T>
struct list
{
    list_node_base<T> head; // or "list_node_base<T>* head;"
    size_t size;
};

template<typename T>
struct list_node_base
{
    list_node<T>* next;
    list_node<T>* prev;
};

template<typename T>
struct list_node : list_node_base<T>
{
    T value;
};
```

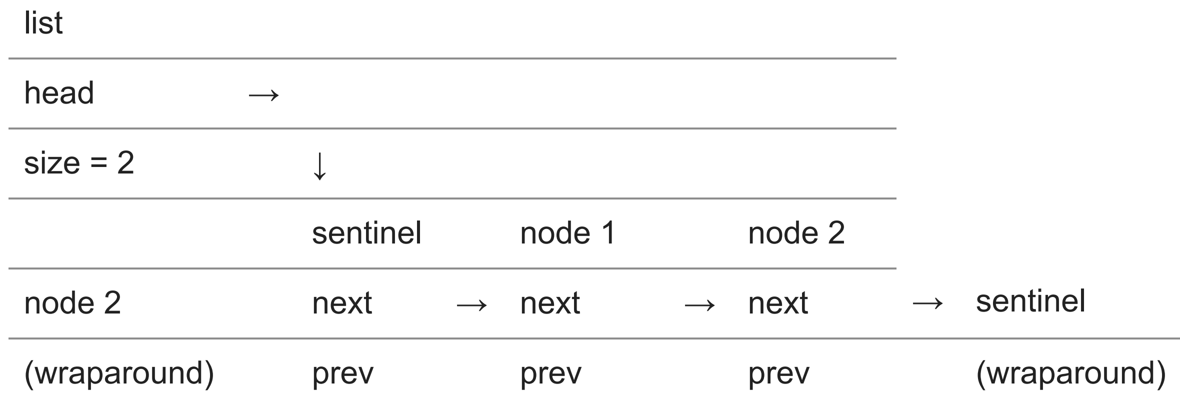
The list is actually circular, with a sentinel node (that has no value) marking the beginning and end of the list. Again, there are two patterns, depending on whether the sentinel is embedded or external.

Here's the version with an embedded sentinel:



	list		node 1		node 2		
node 2	head.next	→	next	→	next	→	list.head
(wraparound)	head.prev		prev		prev		(wraparound)
	size = 2		value = 1		value = 2		

And here's the version with an external sentinel:



list								
head	→							
size = 2		↓						
		sentinel		node 1		node 2		
node 2		next	→	next	→	next	→	sentinel
(wraparound)		prev		prev		prev		(wraparound)

value = 1 value = 2

The Microsoft implementation uses an external sentinel, whereas clang and gcc use an embedded sentinel. This means that the Microsoft implementation incurs an extra allocation for empty lists. This is a minor annoyance because you have to worry about `std::bad_alloc` exceptions at construction.

The only complication is the usual one: Storing the allocator, which is done as a compressed pair with the `head`.

The Visual Studio debugger contains a visualizer for both `forward_list` and `list` 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.

First, the forward list:

```
0:000> ?? l
class std::forward_list<int,std::allocator<int> >
   +0x000 _Mypair          : std::_Compressed_pair<std::allocator<std::_Flist_node<
int,void *> >,std::_Flist_val<std::_Flist_simple_types<int> >,1>
0:000> ?? l._Mypair
class std::_Compressed_pair<std::allocator<std::_Flist_node<int,void *> >,
std::_Flist_val<std::_Flist_simple_types<int> >,1>
   +0x000 _Myval2          : std::_Flist_val<std::_Flist_simple_types<int> >
0:000> ?? l._Mypair._Myval2
class std::_Flist_val<std::_Flist_simple_types<int> >
   +0x000 _Myhead          : 0x000001b8`7b6106a0 std::_Flist_node<int,void *>
0:000> ?? l._Mypair._Myval2._Myhead
struct std::_Flist_node<int,void *> * 0x000001b8`7b6106a0
   +0x000 _Next            : 0x000001b8`7b6106e0 std::_Flist_node<int,void *>
   +0x008 _Myval           : 0n42
0:000> ?? l._Mypair._Myval2._Myhead->_Next
struct std::_Flist_node<int,void *> * 0x000001b8`7b6106e0
   +0x000 _Next            : (null)
   +0x008 _Myval           : 0n99
```

We have to dig through a bunch of wrappers until we get to the `_Myhead`, but then it's smooth sailing dumping each node and following the `_Next` to the next node.

The layout of the `forward_list` nodes happens to match the Windows NT `SINGLE_LIST_ENTRY` structures, so you can use the debugger list commands to view them.

```
          head                 max size
          ↓                    ↓   ↓
0:000> dl 0x000001b8`7b6106a0 999 2
000001b8`7b6106a0  000001b8`7b6106e0 baadf00d`0000002a
000001b8`7b6106e0  00000000`00000000 baadf00d`00000063
↑                  ↑                 ↑        ↑
node address       _Next             padding  _Myval
```

We ask the debugger's "dump list" command (`dl`) to dump the linked list starting at our `head` (`0x000001b8`7b6106a0`), for a maximum of 999 nodes, where each node consists of two pointer-sized values. From that, we can quickly pull out the values `0x2a` (42) and `0x63` (99). If you are willing to be sloppy in the service of expediency, you could just dump the list object itself with `dl`, with the understanding that the first entry is going to be garbage. (It's trying to dump the list itself as a node.)

```
0:000> ?? &l
class std::forward_list<int,std::allocator<int> > * 0000009c`379df8c0
0:000> dl 0x0000009c`379df8c0 999 2
0000009c`379df8c0  000001b8`7b6106a0 0x000063`0000002a ← garbage first "node"
000001b8`7b6106a0  000001b8`7b6106e0 baadf00d`0000002a
000001b8`7b6106e0  00000000`00000000 baadf00d`00000063
```

You can do a little flex and do it all in one line:

```
0:000> dl @@c++(&l) 999 2
0000009c`379df8c0  000001b8`7b6106a0 0x000063`0000002a ← garbage first "node"
000001b8`7b6106a0  000001b8`7b6106e0 baadf00d`0000002a
000001b8`7b6106e0  00000000`00000000 baadf00d`00000063
```

The `@@c++(...)` notation tells the debugger that the enclosed expression is a C++ expression instead of a MASM expression.

The `dl` command stops when it reaches the maximum number of nodes, it encounters a null pointer, or the list loops back to the start.

If you really want to get fancy, you can use the `!list` command, which lets you customize even further how the elements are displayed.

```
0:000> !list -t scratch!LIST_ENTRY.Flink -x "? dwo(@$extret+8)" 0000009c`379df8c0
Evaluate expression: 42 = 00000000`0000002a ← garbage first "node"

Evaluate expression: 42 = 00000000`0000002a

Evaluate expression: 99 = 00000000`00000063
```

Dumping a `list` has a similar initial annoyance, followed by straightforward pointer-chasing. The only trick is knowing when to stop!

```
0:000> ?? l
class std::list<int,std::allocator<int> >
   +0x000 _Mypair          : std::_Compressed_pair<std::allocator<std::_List_node<
int,void *> >,std::_List_val<std::_List_simple_types<int> >,1>
0:000> ?? l._Mypair
class std::_Compressed_pair<std::allocator<std::_List_node<int,void *> >,
std::_List_val<std::_List_simple_types<int> >,1>
   +0x000 _Myval2          : std::_List_val<std::_List_simple_types<int> >
0:000> ?? l._Mypair._Myval2
class std::_List_val<std::_List_simple_types<int> >
   +0x000 _Myhead          : 0x0000017f`a63cf020 std::_List_node<int,void *>
   +0x008 _Mysize          : 2
0:000> ?? l._Mypair._Myval2._Myhead
struct std::_List_node<int,void *> * 0x0000017f`a63cf020
   +0x000 _Next            : 0x0000017f`a63d2730 std::_List_node<int,void *>
   +0x008 _Prev            : 0x0000017f`a63d2780 std::_List_node<int,void *>
   +0x010 _Myval           : 0n-1163005939
0:000> ?? l._Mypair._Myval2._Myhead->_Next
struct std::_List_node<int,void *> * 0x0000017f`a63d2730
   +0x000 _Next            : 0x0000017f`a63d2780 std::_List_node<int,void *>
   +0x008 _Prev            : 0x0000017f`a63cf020 std::_List_node<int,void *>
   +0x010 _Myval           : 0n42
0:000> ?? l._Mypair._Myval2._Myhead->_Next->_Next
struct std::_List_node<int,void *> * 0x0000017f`a63d2780
   +0x000 _Next            : 0x0000017f`a63cf020 std::_List_node<int,void *>
   +0x008 _Prev            : 0x0000017f`a63d2730 std::_List_node<int,void *>
   +0x010 _Myval           : 0n99
```

At this point, we stop because the `_Next` value matches our sentinel value. If we didn't recognize this, we would just follow the `_Next` pointer which takes us back to the sentinel:

```
0:000> ?? l._Mypair._Myval2._Myhead->_Next->_Next->_Next
struct std::_List_node<int,void *> * 0x0000017f`a63cf020
   +0x000 _Next            : 0x0000017f`a63d2730 std::_List_node<int,void *>
   +0x008 _Prev            : 0x0000017f`a63d2780 std::_List_node<int,void *>
   +0x010 _Myval           : 0n-1163005939
```

The layout of the `list` matches the Windows NT `LIST_ENTRY`, so the `dl` command can once again be pressed into service.

```
0:000> dl 0x0000017f`a63cf020 999 3
0000017f`a63cf020  0000017f`a63d2730 0000017f`a63d2780
0000017f`a63cf030  baadf00d`baadf00d ← garbage value in sentinel
0000017f`a63d2730  0000017f`a63d2780 0000017f`a63cf020
0000017f`a63d2740  baadf00d`0000002a ← value 42
0000017f`a63d2780  0000017f`a63cf020 0000017f`a63d2730
0000017f`a63d2790  baadf00d`00000063 ← value 99
```

The nodes are a little harder to read since they spill over two lines. The first line of each node shows the `_Next` and `_Prev` pointers. The second line of each node shows the value.

As a parlor trick, you can use the `dlb` command to dump the doubly-linked list *backward*.

```
0:000> dlb 0x0000017f`a63cf020 999 3
0000017f`a63cf020  0000017f`a63d2730 0000017f`a63d2780
0000017f`a63cf030  baadf00d`baadf00d ← garbage value in sentinel
0000017f`a63d2780  0000017f`a63cf020 0000017f`a63d2730
0000017f`a63d2790  baadf00d`00000063 ← value 99
0000017f`a63d2730  0000017f`a63d2780 0000017f`a63cf020
0000017f`a63d2740  baadf00d`0000002a ← value 42
```

Since the nodes are so clumsy to read, this is a case where the `!list` command comes in handy.

```
0:000> !list -t scratch!LIST_ENTRY.Flink -x "? dwo(@$extret+0x10)" 0000009c`379df8c0
Evaluate expression: 3131961357 = 00000000`baadf00d ← garbage value in sentinel

Evaluate expression: 42 = 00000000`0000002a
```

I’m not sure why the debugger gives up just before reaching the last node.

Okay, so the `list` and `forward_list` aren’t particularly tricky, but we’ll need them later. Stay tuned.