Constructing nodes of a hand-made linked list, how hard can it be?

devblogs.microsoft.com/oldnewthing/20240819-00

August 19, 2024

Raymond Chen

Suppose you are writing your own circular doubly-linked list structure.

```
struct node
{
    node* prev;
    node* next;
};
```

A natural choice for the default constructor is to make the node the sole element of a circular doubly-linked list.

```
struct node
{
    node* prev = this;
    node* next = this;
};
```

What if you also want to add a node after an existing node? Well, we could add a constructor for that.

```
struct node
{
    node* prev = this;
    node* next = this;

    node() = default;

    // Construct a node after a specific node
    node(node* other) :
        prev(other),
        next(other->next)
    {
        prev->next = this;
        next->prev = this;
    }
};
```

(Note that the “construct after another node” constructor takes the other `node` by pointer, rather than by reference, so that it won’t be mistaken for a copy constructor.)

But maybe you also want to have a “before” constructor that inserts the new node before an existing node.

```
struct node
{
    node* prev = this;
    node* next = this;

    node() = default;

    // Construct a node after a specific node
    node(node* other) :
        prev(other),
        next(other->next)
    {
        prev->next = this;
        next->prev = this;
    }

    // Construct a node before a specific node
    node(node* other) :
        prev(other->prev),
        next(other)
    {
        prev->next = this;
        next->prev = this;
    }
};
```

Uh-oh, we have two constructors with the same parameter list. That’s not going to work.

And really, even if we had only one of the constructors, it would still be a bad design because it’s not clear whether the newly-created item goes before or after.

A traditional solution to this is to use a factory method, so that the method name tells you whether it’s going before or after.

```
struct node
{
    node* prev = this;
    node* next = this;

    node() = default;

    static node create_after(node* other)
    {
        node n{ other->prev, other };
        n.prev->next = &n;
        n.next->prev = &n;
        return n;
    }

    static node create_before(node* other)
    {
        node n{ other, other->next };
        n.prev->next = &n;
        n.next->prev = &n;
        return n;
    }
};

// usage
node a;
node b = node::create_after(&a); // a->b
node c = node::create_before(&b); // a->c->b
```

Unfortunately, this doesn’t work, or at least isn’t guaranteed to work, because copy elision is not guaranteed for named return values.

I tried to think of ways to get guaranteed copy elision while simultaneously having access to the address of the object, but the only way I could come up with was to overload the constructor, say by using a tag type.

```
struct place_after_t {};
inline constexpr place_after_t place_after{};

struct place_before_t {};
inline constexpr place_before_t place_before{};

struct node
{
    node* prev = this;
    node* next = this;

    node() = default;

    // Construct a node after a specific node
    node(place_after_t, node* other) :
        prev(other),
        next(other->next)
    {
        prev->next = this;
        next->prev = this;
    }

    // Construct a node before a specific node
    node(place_before_t, node* other) :
        prev(other->prev),
        next(other)
    {
        prev->next = this;
        next->prev = this;
    }

    static node create_after(node* other)
    {
        return { place_after, other };
    }

    static node create_before(node* other)
    {
        return { place_before, other };
    }
};

// Sample usage 1: Using tags
node a;
node b(place_after, &a); // list is a->b
node c(place_before, &b); // list is  a->c->b

// Sample usage 2: Using factories
node a;
node b = node::create_after(&a); // list is a->b
node c = node::create_before(&b); // list is  a->c->b
```

Or maybe we are looking at it the wrong way, and really the functionality we want are `create_after` and `create_before` instance methods.

```
struct node
{
    node* prev = this;
    node* next = this;

    node() = default;

    node create_after() { return { place_after, this }; }
    node create_before() { return { place_before, this }; }

private:
    struct place_after_t {};
    inline constexpr place_after_t place_after{};

    struct place_before_t {};
    inline constexpr place_before_t place_before{};

    node(place_after_t, node* other) :
        prev(other),
        next(other->next)
    {
        prev->next = this;
        next->prev = this;
    }

    node(place_before_t, node* other) :
        prev(other->prev),
        next(other)
    {
        prev->next = this;
        next->prev = this;
    }
};

// Sample usage
node a;
node b = a.create_after(); // list is a->b
node c = b.create_before(); // list is a->c->b
```

The moral of the story, I think, is that if you want to force copy elision of an object whose address must be known before it is returned, you have to do it from the constructor, because that’s the only time guaranteed copy elision will give you the object’s address.