```
print_all(greek_gods);
cout<<"\n";
```

This should give

Click here to view code image

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

17.10 The this pointer

Note that each of our list functions takes a Link* as its first argument and accesses data in that object. That's the kind of function that we often make member functions. Could we simplify Link (or link use) by making the operations members? Could we maybe make the pointers private so that only the member functions have access to them? We could:

Click here to view code image

```
class Link {
public:
        string value;

        Link(const string& v, Link* p = nullptr, Link* s = nullptr)
                : value{v}, prev{p}, succ{s} { }

        Link* insert(Link* n) ;                     // insert n before this object
        Link* add(Link* n) ;                        // insert n after this object
        Link* erase() ;                             // remove this object from list
        Link* find(const string& s);                // find s in list
        const Link* find(const string& s) const;    // find s in const list (see §18.5.1)

        Link* advance(int n) const;                 // move n positions in list

        Link* next() const { return succ; }
        Link* previous() const { return prev; }
private:
        Link* prev;
        Link* succ;
};
```

This looks promising. We defined the operations that don't change the state of a Link into const member functions. We added (nonmodifying) next() and previous() functions so that users could iterate over lists (of Links) — those are needed now that direct access to succ and prev is prohibited. We left value as a public member because (so far) we have no reason not to; it is "just data."

Now let's try to implement Link::insert() by copying our previous global insert() and modifying it suitably:

Click here to view code image

```
Link* Link::insert(Link* n)          // insert n before p; return n
{
        Link* p = this;               // pointer to this object
        if (n==nullptr) return p;     // nothing to insert
        if (p==nullptr) return n;     // nothing to insert into
        n->succ = p;                  // p comes after n
        if (p->prev) p->prev->succ = n;
        n->prev = p->prev;            // p's predecessor becomes n's predecessor
        p->prev = n;                  // n becomes p's predecessor
```

```
        return n;
}
```

But how do we get a pointer to the object for which Link::insert() was called? Without help from the language we can't. However, in every member function, the identifier this is a pointer that points to the object for which the member function was called. Alternatively, we could simply use this instead of p:

Click here to view code image

```
Link* Link::insert(Link* n)          // insert n before this object; return n
{
        if (n==nullptr) return this;
        if (this==nullptr) return n;
        n->succ = this;                  // this object comes after n
        if (this->prev) this->prev->succ = n;
        n->prev = this->prev;          // this object's predecessor
                                            // becomes n's predecessor
        this->prev = n;                  // n becomes this object's predecessor
        return n;
}
```

This is a bit verbose, but we don't need to mention this to access a member, so we can abbreviate:

Click here to view code image

```
Link* Link::insert(Link* n)    // insert n before this object; return n
{
        if (n==nullptr) return this;
        if (this==nullptr) return n;
        n->succ = this;            // this object comes after n
        if (prev) prev->succ = n;
        n->prev = prev;          // this object's predecessor becomes n's predecessor
        prev = n;                  // n becomes this object's predecessor
        return n;
}
```

In other words, we have been using the this pointer — the pointer to the current object — implicitly every time we accessed a member. It is only when we need to refer to the whole object that we need to mention it explicitly.

Note that this has a specific meaning: it points to the object for which a member function is called. It does not point to any old object. The compiler ensures that we do not change the value of this in a member function. For example:

Click here to view code image

```
struct S {
        // . . .
        void mutate(S* p)
        {
                this = p;      // error: this is immutable
                // . . .
        }
};
```

17.10.1 More link use

Having dealt with the implementation issues, we can see how the use now looks:

Click here to view code image

```
Link* norse_gods = new Link{"Thor"};
norse_gods = norse_gods->insert(new Link{"Odin"});
norse_gods = norse_gods->insert(new Link{"Zeus"});
norse_gods = norse_gods->insert(new Link{"Freia"});

Link* greek_gods = new Link{"Hera"};
greek_gods = greek_gods->insert(new Link{"Athena"});
greek_gods = greek_gods->insert(new Link{"Mars"});
greek_gods = greek_gods->insert(new Link{"Poseidon"});
```

That's very much like before. As before, we correct our "mistakes." Correct the name of the god of war:

Click here to view code image

```
Link* p = greek_gods->find("Mars");
if (p) p->value = "Ares";
```

Move Zeus into his correct Pantheon:

Click here to view code image

```
Link* p2 = norse_gods->find("Zeus");
if (p2) {
        if (p2==norse_gods) norse_gods = p2->next();
        p2->erase();
        greek_gods = greek_gods->insert(p2);
}
```

Finally, let's print out those lists:

Click here to view code image

```
void print_all(Link* p)
{
        cout << "{ ";
        while (p) {
                cout << p->value;
                if (p=p->next()) cout << ", ";
        }
        cout << " }";
}
print_all(norse_gods);
cout<<"\n";

print_all(greek_gods);
cout<<"\n";
```

This should again give

Click here to view code image

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

So, which version do you like better: the one where insert(), etc. are member functions or the one where they are freestanding functions? In this case the differences don't matter much, but see §9.7.5.

One thing to observe here is that we still don't have a list class, only a link class. That forces us to keep worrying about which pointer is the pointer to the first element. We can do better than that — by defining a class List — but designs along the lines presented here are very common. The standard library list is presented in §20.4.

Drill

This drill has two parts. The first exercises/builds your understanding of free-store-allocated arrays and contrasts arrays with vectors:

1. Allocate an array of ten ints on the free store using new.
2. Print the values of the ten ints to cout.
3. Deallocate the array (using delete[]).
4. Write a function print_array10(ostream& os, int* a) that prints out the values of a (assumed to have ten elements) to os.
5. Allocate an array of ten ints on the free store; initialize it with the values 100, 101, 102, etc.; and print out its values.
6. Allocate an array of 11 ints on the free store; initialize it with the values 100, 101, 102, etc.; and print out its values.
7. Write a function print_array(ostream& os, int* a, int n) that prints out the values of a (assumed to have n elements) to os.
8. Allocate an array of 20 ints on the free store; initialize it with the values 100, 101, 102, etc.; and print out its values.
9. Did you remember to delete the arrays? (If not, do it.)
10. Do 5, 6, and 8 using a vector instead of an array and a print_vector() instead of print_array().

The second part focuses on pointers and their relation to arrays. Using print_array() from the last drill:

1. Allocate an int, initialize it to 7, and assign its address to a variable p1.
2. Print out the value of p1 and of the int it points to.
3. Allocate an array of seven ints; initialize it to 1, 2, 4, 8, etc.; and assign its address to a variable p2.
4. Print out the value of p2 and of the array it points to.
5. Declare an int* called p3 and initialize it with p2.
6. Assign p1 to p2.
7. Assign p3 to p2.
8. Print out the values of p1 and p2 and of what they point to.