need for a cast. When you think you need a cast, reconsider: Is there a way to write the code without the cast? Is there a way to redesign that part of the program so that the cast is not needed? Unless you are interfacing to other people's code or to hardware, there usually is a way. If not, expect subtle and nasty bugs. Don't expect code using reinterpret_cast to be portable.

17.9 Pointers and references

You can think of a reference as an automatically dereferenced immutable pointer or as an alternative name for an object. Pointers and references differ in these ways:

- Assignment to a pointer changes the pointer's value (not the pointed-to value).
- To get a pointer you generally need to use new or &.
- To access an object pointed to by a pointer you use * or [ ].
- Assignment to a reference changes the value of the object referred to (not the reference itself).
- You cannot make a reference refer to a different object after initialization.
- Assignment of references does deep copy (assigns to the referred-to object); assignment of pointers does not (assigns to the pointer object itself).
- Beware of null pointers.

For example:

Click here to view code image

```
int x = 10;
int* p = &x;          // you need & to get a pointer
*p = 7;               // use * to assign to x through p
int x2 = *p;          // read x through p
int* p2 = &x2;        // get a pointer to another int
p2 = p;               // p2 and p both point to x
p = &x2;              // make p point to another object
```

The corresponding example for references is

Click here to view code image

```
int y = 10;
int& r = y;           // the & is in the type, not in the initializer
r = 7;                // assign to y through r (no * needed)
int y2 = r;           // read y through r (no * needed)
int& r2 = y2;         // get a reference to another int
r2 = r;               // the value of y is assigned to y2
r = &y2;              // error: you can't change the value of a reference
                      // (no assignment of an int* to an int&)
```

Note the last example; it is not just this construct that will fail — there is no way to get a reference to refer to a different object after initialization. If you need to point to something different, use a pointer. For ideas of how to use pointers, see §17.9.3.

A reference and a pointer are both implemented by using a memory address. They just use that address differently to pro-

vide you — the programmer — slightly different facilities.

17.9.1 Pointer and reference parameters

When you want to change the value of a variable to a value computed by a function, you have three choices. For example:

Click here to view code image

```
int incr_v(int x) { return x+1; }     // compute a new value and return it
void incr_p(int* p) { ++*p; }         // pass a pointer
                                          // (dereference it and increment the result)
void incr_r(int& r) { ++r; }          // pass a reference
```

How do you choose? We think returning the value often leads to the most obvious (and therefore least error-prone) code; that is:

Click here to view code image

```
int x = 2;
x = incr_v(x);        // copy x to incr_v(); then copy the result out and assign it
```

We prefer that style for small objects, such as an int. In addition, if a "large object" has a move constructor (§18.3.4) we can efficiently pass it back and forth.

How do we choose between using a reference argument and using a pointer argument? Unfortunately, either way has both attractions and problems, so again the answer is less than clear-cut. You have to make a decision based on the individual function and its likely uses.

Using a pointer argument alerts the programmer that something might be changed. For example:

Click here to view code image

```
int x = 7;
incr_p(&x)          // the & is needed
incr_r(x);
```

The need to use & in incr_p(&x) alerts the user that x might be changed. In contrast, incr_r(x) "looks innocent." This leads to a slight preference for the pointer version.

On the other hand, if you use a pointer as a function argument, the function has to beware that someone might call it with a null pointer, that is, with a nullptr. For example:

Click here to view code image

```
incr_p(nullptr);        // crash: incr_p() will try to dereference the null pointer
int* p = nullptr;
incr_p(p);              // crash: incr_p() will try to dereference the null pointer
```

This is obviously nasty. The person who writes incr_p() can protect against this:

Click here to view code image

```
void incr_p(int* p)
{
        if (p==nullptr) error("null pointer argument to incr_p()");
```

```
        ++*p;       // dereference the pointer and increment the object pointed to
}
```

But now incr_p() suddenly doesn't look as simple and attractive as before. Chapter 5 discusses how to cope with bad arguments. In contrast, users of a reference (such as incr_r()) are entitled to assume that a reference refers to an object.

If "passing nothing" (passing no object) is acceptable from the point of view of the semantics of the function, we must use a pointer argument. Note: That's not the case for an increment operation — hence the need for throwing an exception for p==nullptr.

So, the real answer is: "The choice depends on the nature of the function":

- For tiny objects prefer pass-by-value.
- For functions where "no object" (represented by nullptr) is a valid argument use a pointer parameter (and remember to test for nullptr).
- Otherwise, use a reference parameter.

See also §8.5.6.

17.9.2 Pointers, references, and inheritance

In §14.3, we saw how a derived class, such as Circle, could be used where an object of its public base class Shape was required. We can express that idea in terms of pointers or references: a Circle* can be implicitly converted to a Shape* because Shape is a public base of Circle. For example:

Click here to view code image

```
void rotate(Shape* s, int n);           // rotate *s n degrees

Shape* p = new Circle{Point{100,100},40};
Circle c {Point{200,200},50};
rotate(p,35);
rotate(&c,45);
```

And similarly for references:

Click here to view code image

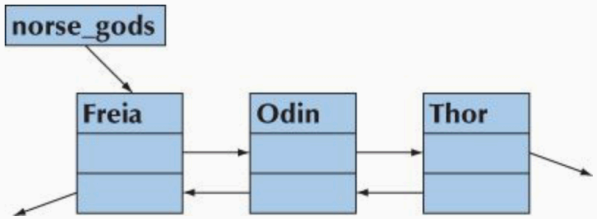
```
void rotate(Shape& s, int n);       // rotate s n degrees

Shape& r = c;
rotate(r,55);
rotate(*p,65);
rotate(c,75);
```

This is crucial for most object-oriented programming techniques (§14.3–4).

17.9.3 An example: lists

Lists are among the most common and useful data structures. Usually, a list is made out of "links" where each link holds some information and pointers to other links. This is one of the classical uses of pointers. For example, we could represent a short list of Norse gods like this:



A list like this is called a *doubly-linked list* because given a link, we can find both the predecessor and the successor. A list where we can find only the successor is called a *singly-linked list*. We use doubly-linked lists when we want to make it easy to remove an element. We can define these links like this:

Click here to view code image

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

That is, given a Link, we can get to its successor using the succ pointer and to its predecessor using the prev pointer. We use the null pointer to indicate that a Link doesn't have a successor or a predecessor. We can build our list of Norse gods like this:

Click here to view code image

```
Link* norse_gods = new Link{"Thor",nullptr,nullptr};
norse_gods = new Link{"Odin",nullptr,norse_gods};
norse_gods->succ->prev = norse_gods;
norse_gods = new Link{"Freia",nullptr,norse_gods};
norse_gods->succ->prev = norse_gods;
```

We built that list by creating the Links and tying them together as in the picture: first Thor, then Odin as the predecessor of Thor, and finally Freia as the predecessor of Odin. You can follow the pointer to see that we got it right, so that each succ and prev points to the right god. However, the code is obscure because we didn't explicitly define and name an insert operation:

Click here to view code image

```
Link* insert(Link* p, Link* n)     // insert n before p (incomplete)
{
        n->succ = p;                // p comes after n
      p->prev->succ = n;            // n comes after what used to be p's predecessor
      n->prev = p->prev;            // p's predecessor becomes n's predecessor
      p->prev = n;                  // n becomes p's predecessor
```

```
        return n;
}
```

This works provided that p really points to a Link and that the Link pointed to by p really has a predecessor. Please convince yourself that this really is so. When thinking about pointers and linked structures, such as a list made out of Links, we invariably draw little box-and-arrow diagrams on paper to verify that our code works for small examples. Please don't be too proud to rely on this effective low-tech design technique.

That version of insert() is incomplete because it doesn't handle the cases where n, p, or p->prev is nullptr. We add the appropriate tests for the null pointer and get the messier, but correct, version:

Click here to view code image

```
Link* insert(Link* p, Link* n)     // insert n before p; return n
{
        if (n==nullptr) return p;
        if (p==nullptr) return n;
        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;
}
```

Given that, we could write

Click here to view code image


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

Now all the error-prone fiddling with the prev and succ pointers has disappeared from sight. Pointer fiddling is tedious and error-prone and *should* be hidden in well-written and well-tested functions. In particular, many errors in conventional code come from people forgetting to test pointers against nullptr — just as we (deliberately) did in the first version of insert().

Note that we used default arguments (§15.3.1, §A.9.2) to save users from mentioning predecessors and successors in every constructor use.

17.9.4 List operations

The standard library provides a list class, which we will describe in §20.4. It hides all link manipulation, but here we will elaborate on our notion of list based on the Link class to get a feel for what goes on "under the covers" of list classes and see more examples of pointer use.

What operations does our Link class need to allow its users to avoid "pointer fiddling"? That's to some extent a matter of taste, but here is a useful set:

- The constructor

- insert: insert before an element
- add: insert after an element
- erase: remove an element
- find: find a Link with a given value
- advance: get the *n*th successor

We could write these operations like this:

Click here to view code image

```
Link* add(Link* p, Link* n)       // insert n after p; return n
{
        // much like insert (see exercise 11)
}

Link* erase(Link* p)              // remove *p from list; return p's successor
{
        if (p==nullptr) return nullptr;
        if (p->succ) p->succ->prev = p->prev;
        if (p->prev) p->prev->succ = p->succ;
        return p->succ;
}

Link* find(Link* p, const string& s)          // find s in list;
                                              // return nullptr for "not found"
{
        while (p) {
                if (p->value == s) return p;
                p = p->succ;
        }
        return nullptr;
}
```



```
Link* advance(Link* p, int n)                 // move n positions in list
                                              // return nullptr for "not found"
        // positive n moves forward, negative backward
{
        if (p==nullptr) return nullptr;
        if (0<n) {
                while (n--) {
                        if (p->succ == nullptr) return nullptr;
                        p = p->succ;
                }
        }
        else if (n<0) {
                while (n++) {
                        if (p->prev == nullptr) return nullptr;
                        p = p->prev;
                }
        }
        return p;
}
```

Note the use of the postfix n++. This form of increment ("post-increment") yields the value before the increment as its value.

17.9.5 List use

As a little exercise, let's build two lists:

Click here to view code image

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

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

"Unfortunately," we made a couple of mistakes: Zeus is a Greek god, rather than a Norse god, and the Greek god of war is Ares, not Mars (Mars is his Latin/Roman name). We can fix that:

Click here to view code image

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

Note how we were cautious about find() returning a nullptr. We think that we know that it can't happen in this case (after all, we just inserted Mars into greek_gods), but in a real example someone might change that code.

Similarly, we can move Zeus into his correct Pantheon:

Click here to view code image

```
Link* p = find(norse_gods,"Zeus");
if (p) {
        erase(p);
        insert(greek_gods,p);
}
```

Did you notice the bug? It's quite subtle (unless you are used to working directly with links). What if the Link we erase() is the one pointed to by norse_gods? Again, that doesn't actually happen here, but to write good, maintainable code, we have to take that possibility into account:

Click here to view code image

```
Link* p = find(norse_gods, "Zeus");
if (p) {
        if (p==norse_gods) norse_gods = p–>succ;
        erase(p);
        greek_gods = insert(greek_gods,p);
}
```

While we were at it, we also corrected the second bug: when we insert Zeus *before* the first Greek god, we need to make greek_gods point to Zeus's Link. Pointers are extremely useful and flexible, but subtle.

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–>succ) cout << ", ";
        }
        cout << " }";
}
print_all(norse_gods);
cout<<"\n";
```

```
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
```