4
Inheritance
Code Reuse and Refinement

4.1 A Superclass — Point

In this chapter we will start a rudimentary drawing program. Here is a quick test for one of the classes we would like to have:

```c
#include "Point.h"
#include "new.h"

int main (int argc, char ** argv)
{
    void * p;
    while (* ++ argv)
    {
        switch (** argv) {
            case 'p':
                p = new(Point, 1, 2);
                break;
            default:
                continue;
        }
        draw(p);
        move(p, 10, 20);
        draw(p);
        delete(p);
    }
    return 0;
}
```

For each command argument starting with the letter `p` we get a new point which is drawn, moved somewhere, drawn again, and deleted. ANSI-C does not include standard functions for graphics output; however, if we insist on producing a picture we can emit text which Kernighan's pic [Ker82] can understand:

```
$ points p
   ." at 1,2
   ." at 11,22
```

The coordinates do not matter for the test — paraphrasing a commercial and oospeak: “the point is the message.”

What can we do with a point? `new()` will produce a point and the constructor expects initial coordinates as further arguments to `new()`. As usual, `delete()` will recycle our point and by convention we will allow for a destructor.

`draw()` arranges for the point to be displayed. Since we expect to work with other graphical objects — hence the `switch` in the test program — we will provide dynamic linkage for `draw()`.
**move()** changes the coordinates of a point by the amounts given as arguments. If we implement each graphical object relative to its own reference point, we will be able to move it simply by applying **move()** to this point. Therefore, we should be able to do without dynamic linkage for **move()**.

### 4.2 Superclass Implementation — **Point**

The abstract data type interface in **Point.h** contains the following:

```c
extern const void * Point;  /* new(Point, x, y); */
void move (void * point, int dx, int dy);
```

We can recycle the **new.c** files from chapter 2 except that we remove most methods and add **draw()** to **new.h**:

```c
void * new (const void * class, ...);
void delete (void * item);
void draw (const void * self);
```

The type description **struct Class** in **new.r** should correspond to the method declarations in **new.h**:

```c
struct Class {
    size_t size;
    void * (* ctor) (void * self, va_list * app);
    void * (* dtor) (void * self);
    void (* draw) (const void * self);
};
```

The selector **draw()** is implemented in **new.c**. It replaces selectors such as **differ()** introduced in section 2.3 and is coded in the same style:

```c
void draw (const void * self)
{
    const struct Class * const * cp = self;
    assert(self && * cp && (* cp) —> draw);
    (* cp) —> draw(self);
}
```

After these preliminaries we can turn to the real work of writing **Point.c**, the implementation of points. Once again, object-orientation has helped us to identify precisely what we need to do: we have to decide on a representation and implement a constructor, a destructor, the dynamically linked method **draw()** and the statically linked method **move()**, which is just a plain function. If we stick with two-dimensional, Cartesian coordinates, we choose the obvious representation:

```c
struct Point {
    const void * class;
    int x, y;  /* coordinates */
};
```

The constructor has to initialize the coordinates .x and .y — by now absolutely routine:
It turns out that we do not need a destructor because we have no resources to reclaim before `delete()` does away with `struct Point` itself. In `Point_draw()` we print the current coordinates in a way which `pic` can understand:

```c
static void Point_draw (const void * _self)
{
    const struct Point * self = _self;
    printf("\".\" at %d,%d\n", self->x, self->y);
}
```

This takes care of all the dynamically linked methods and we can define the type descriptor, where a null pointer represents the non-existing destructor:

```c
static const struct Class _Point = {
    sizeof(struct Point), Point_ctor, 0, Point_draw
};
const void * Point = & _Point;
```

`move()` is not dynamically linked, so we omit `static` to export it from `Point.c` and we do not prefix its name with the class name `Point`:

```c
void move (void * _self, int dx, int dy)
{
    struct Point * self = _self;
    self->x += dx, self->y += dy;
}
```

This concludes the implementation of points in `Point` together with the support for dynamic linkage in `new`.

### 4.3 Inheritance — Circle

A circle is just a big point: in addition to the center coordinates it needs a radius. Drawing happens a bit differently, but moving only requires that we change the coordinates of the center.

This is where we would normally crank up our text editor and perform source code reuse. We make a copy of the implementation of points and change those parts where a circle is different from a point. `struct Circle` gets an additional component:

```c
int rad;
```

This component is initialized in the constructor

```c
self -> rad = va_arg(* app, int);
```

and used in `Circle_draw()`:
printf("circle at \%d,\%d rad \%d\n",
    self —> x, self —> y, self —> rad);

We get a bit stuck in move(). The necessary actions are identical for a point and a circle: we need to add the displacement arguments to the coordinate components. However, in one case, move() works on a struct Point, and in the other case, it works on a struct Circle. If move() were dynamically linked, we could provide two different functions to do the same thing, but there is a much better way. Consider the layout of the representations of points and circles:

<table>
<thead>
<tr>
<th>point class</th>
<th>circle class</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>rad</td>
<td></td>
</tr>
</tbody>
</table>

struct Point struct Circle

The picture shows that every circle begins with a point. If we derive struct Circle by adding to the end of struct Point, we can pass a circle to move() because the initial part of its representation looks just like the point which move() expects to receive and which is the only thing that move() can change. Here is a sound way to make sure the initial part of a circle always looks like a point:

```
struct Circle { const struct Point _; int rad; };
```

We let the derived structure start with a copy of the base structure that we are extending. Information hiding demands that we should never reach into the base structure directly; therefore, we use an almost invisible underscore as its name and we declare it to be `const` to ward off careless assignments.

This is all there is to simple inheritance: a subclass is derived from a superclass (or base class) merely by lengthening the structure that represents an object of the superclass.

Since representation of the subclass object (a circle) starts out like the representation of a superclass object (a point), the circle can always pretend to be a point — at the initial address of the circle’s representation there really is a point’s representation.

It is perfectly sound to pass a circle to move(): the subclass inherits the methods of the superclass because these methods only operate on that part of the subclass’ representation that is identical to the superclass’ representation for which the methods were originally written. Passing a circle as a point means converting from a struct Circle * to a struct Point *. We will refer to this as an up-cast from a subclass to a superclass — in ANSI-C it can only be accomplished with an explicit conversion operator or through intermediate void * values.
It is usually unsound, however, to pass a point to a function intended for circles such as `Circle_draw()`: converting from a `struct Point *` to a `struct Circle *` is only permissible if the point originally was a circle. We will refer to this as a down-cast from a superclass to a subclass — this requires explicit conversions or `void *` values, too, and it can only be done to pointers to objects that were in the subclass to begin with.

### 4.4 Linkage and Inheritance

`move()` is not dynamically linked and does not use a dynamically linked method to do its work. While we can pass points as well as circles to `move()`, it is not really a polymorphic function: `move()` does not act differently for different kinds of objects, it always adds arguments to coordinates, regardless of what else might be attached to the coordinates.

The situation is different for a dynamically linked method like `draw()`. Let us look at the previous picture again, this time with the type descriptions shown explicitly:

<table>
<thead>
<tr>
<th>Point</th>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>size</td>
<td>size</td>
</tr>
<tr>
<td>ctor</td>
<td>ctor</td>
</tr>
<tr>
<td>dtor</td>
<td>dtor</td>
</tr>
<tr>
<td>draw</td>
<td>draw</td>
</tr>
</tbody>
</table>

```plaintext
struct Point

struct Circle
```

When we up-cast from a circle to a point, we do not change the state of the circle, i.e., even though we look at the circle's `struct Circle` representation as if it were a `struct Point`, we do not change its contents. Consequently, the circle viewed as a point still has `Circle` as a type description because the pointer in its `.class` component has not changed. `draw()` is a selector function, i.e., it will take whatever argument is passed as `self`, proceed to the type description indicated by `.class`, and call the `draw` method stored there.

A subclass inherits the statically linked methods of its superclass — those methods operate on the part of the subclass object which is already present in the superclass object. A subclass can choose to supply its own methods in place of the dynamically linked methods of its superclass. If inherited, i.e., if not overwritten, the superclass' dynamically linked methods will function just like statically linked methods and modify the superclass part of a subclass object. If overwritten, the subclass' own version of a dynamically linked method has access to the full representation of a subclass object, i.e., for a circle `draw()` will invoke `Circle_draw()` which can consider the radius when drawing the circle.
4.5 Static and Dynamic Linkage

A subclass inherits the statically linked methods of its superclass and it can choose to inherit or overwrite the dynamically linked methods. Consider the declarations for `move()` and `draw()`:

```c
void move (void * point, int dx, int dy);
void draw (const void * self);
```

We cannot discover the linkage from the two declarations, although the implementation of `move()` does its work directly, while `draw()` is only the selector function which traces the dynamic linkage at runtime. The only difference is that we declare a statically linked method like `move()` as part of the abstract data type interface in `Point.h`, and we declare a dynamically linked method like `draw()` with the memory management interface in `new.h`, because we have thus far decided to implement the selector function in `new.c`.

Static linkage is more efficient because the C compiler can code a subroutine call with a direct address, but a function like `move()` cannot be overwritten for a subclass. Dynamic linkage is more flexible at the expense of an indirect call — we have decided on the overhead of calling a selector function like `draw()`, checking the arguments, and locating and calling the appropriate method. We could forgo the checking and reduce the overhead with a macro* like

```c
#define draw(self) 
  ( (* (struct Class **) self) -> draw (self))
```

but macros cause problems if their arguments have side effects and there is no clean technique for manipulating variable argument lists with macros. Additionally, the macro needs the declaration of `struct Class` which we have thus far made available only to class implementations and not to the entire application.

Unfortunately, we pretty much decide things when we design the superclass. While the function calls to the methods do not change, it takes a lot of text editing, possibly in a lot of classes, to switch a function definition from static to dynamic linkage and vice versa. Beginning in chapter 7 we will use a simple preprocessor to simplify coding, but even then linkage switching is error-prone.

In case of doubt it is probably better to decide on dynamic rather than static linkage even if it is less efficient. Generic functions can provide a useful conceptual abstraction and they tend to reduce the number of function names which we need to remember in the course of a project. If, after implementing all required classes, we discover that a dynamically linked method was never overwritten, it is a lot less trouble to replace its selector by its single implementation, and even waste its slot in `struct Class`, than to extend the type description and correct all the initializations.

---

* In ANSI-C macros are not expanded recursively so that a macro may hide a function by the same name.
4.6 Visibility and Access Functions

We can now attempt to implement `Circle_draw()`. Information hiding dictates that we use three files for each class based on a “need to know” principle. `Circle.h` contains the abstract data type interface; for a subclass it includes the interface file of the superclass to make declarations for the inherited methods available:

```c
#include "Point.h"
extern const void * Circle; /* new(Circle, x, y, rad) */
```

The interface file `Circle.h` is included by the application code and for the implementation of the class; it is protected from multiple inclusion.

The representation of a circle is declared in a second header file, `Circle.r`. For a subclass it includes the representation file of the superclass so that we can derive the representation of the subclass by extending the superclass:

```c
#include "Point.r"
struct Circle { const struct Point _; int rad; };
```

The subclass needs the superclass representation to implement inheritance: `struct Circle` contains a `const struct Point`. The point is certainly not constant — `move()` will change its coordinates — but the `const` qualifier guards against accidentally overwriting the components. The representation file `Circle.r` is only included for the implementation of the class; it is protected from multiple inclusion.

Finally, the implementation of a circle is defined in the source file `Circle.c` which includes the interface and representation files for the class and for object management:

```c
#include "Circle.h"
#include "Circle.r"
#include "new.h"
#include "new.r"
static void Circle_draw (const void * _self)
{
    const struct Circle * self = _self;
    printf("circle at %d,%d rad %d\n",
           self->_.x, self->_.y, self->rad);
}
```

In `Circle_draw()` we have read point components for the circle by invading the subclass part with the “invisible name” `_`. From an information hiding perspective this is not such a good idea. While reading coordinate values should not create major problems we can never be sure that in other situations a subclass implementation is not going to cheat and modify its superclass part directly, thus potentially playing havoc with its invariants.

Efficiency dictates that a subclass reach into its superclass components directly. Information hiding and maintainability require that a superclass hide its own representation as best as possible from its subclasses. If we opt for the latter, we should provide access functions for all those components of a superclass which a subclass is allowed to look at, and modification functions for those components, if any, which the subclass may modify.
Access and modification functions are statically linked methods. If we declare them in the representation file for the superclass, which is only included in the implementations of subclasses, we can use macros, because side effects are no problem if a macro uses each argument only once. As an example, in `Point.r` we define the following access macros:

```c
#define x(p) (((const struct Point *)(p)) -> x)
#define y(p) (((const struct Point *)(p)) -> y)
```

These macros can be applied to a pointer to any object that starts with a `struct Point`, i.e., to objects from any subclass of our points. The technique is to up-cast the pointer into our superclass and reference the interesting component there. `const` in the cast blocks assignments to the result. If `const` were omitted

```c
#define x(p) (((struct Point *)(p)) -> x)
```

a macro call `x(p)` produces an l-value which can be the target of an assignment. A better modification function would be the macro definition

```c
#define set_x(p,v) (((struct Point *)(p)) -> x = (v))
```

which produces an assignment.

Outside the implementation of a subclass we can only use statically linked methods for access and modification functions. We cannot resort to macros because the internal representation of the superclass is not available for the macros to reference. Information hiding is accomplished by not providing the representation file `Point.r` for inclusion into an application.

The macro definitions demonstrate, however, that as soon as the representation of a class is available, information hiding can be quite easily defeated. Here is a way to conceal `struct Point` much better. Inside the superclass implementation we use the normal definition:

```c
struct Point {
    const void * class;
    int x, y; /* coordinates */
};
```

For subclass implementations we provide the following opaque version:

```c
struct Point {
    const char _ [ sizeof( struct {
    const void * class;
    int x, y; /* coordinates */
    }));
};
```

This structure has the same size as before, but we can neither read nor write the components because they are hidden in an anonymous interior structure. The catch is that both declarations must contain identical component declarations and this is difficult to maintain without a preprocessor.

---

* In ANSI-C, a parametrized macro is only expanded if the macro name appears before a left parenthesis. Elsewhere, the macro name behaves like any other identifier.
4.7 Subclass Implementation — Circle

We are ready to write the complete implementation of circles, where we can choose whatever techniques of the previous sections we like best. Object-orientation prescribes that we need a constructor, possibly a destructor, Circle_draw(), and a type description Circle to tie it all together. In order to exercise our methods, we include Circle.h and add the following lines to the switch in the test program in section 4.1:

```c
    case 'c':
        p = new(Circle, 1, 2, 3);
        break;
```

Now we can observe the following behavior of the test program:

```
$ circles p c
"." at 1,2
"." at 11,22
circle at 1,2 rad 3
circle at 11,22 rad 3
```

The circle constructor receives three arguments: first the coordinates of the circle’s point and then the radius. Initializing the point part is the job of the point constructor. It consumes part of the argument list of new(). The circle constructor is left with the remaining argument list from which it initializes the radius.

A subclass constructor should first let the superclass constructor do that part of the initialization which turns plain memory into the superclass object. Once the superclass constructor is done, the subclass constructor completes initialization and turns the superclass object into a subclass object.

For circles this means that we need to call Point_ctor(). Like all dynamically linked methods, this function is declared static and thus hidden inside Point.c. However, we can still get to the function by means of the type descriptor Point which is available in Circle.c:

```c
    static void * Circle_ctor (void * _self, va_list * app)
    {
        struct Circle * self =
            ((const struct Class *) Point) -> ctor(_self, app);
        self -> rad = va_arg(* app, int);
        return self;
    }
```

It should now be clear why we pass the address app of the argument list pointer to each constructor and not the va_list value itself: new() calls the subclass constructor, which calls its superclass constructor, and so on. The supermost constructor is the first one to actually do something, and it gets first pick at the left end of the argument list passed to new(). The remaining arguments are available to the next subclass and so on until the last, rightmost arguments are consumed by the final subclass, i.e., by the constructor directly called by new().

Destruction is best arranged in the exact opposite order: delete() calls the subclass destructor. It should destroy its own resources and then call its direct superclass destructor which can destroy the next set of resources and so on. Construc-
Inheritance — Code Reuse and Refinement

Inheritance happens superclass before subclass, destruction happens in reverse, subclass before superclass, circle part before point part. Here, however, nothing needs to be done.

We have worked on Circle_draw() before. We use visible components and code the representation file Point.r as follows:

```c
struct Point {
    const void * class;
    int x, y;       /* coordinates */
};
#define x(p) (((const struct Point *)(p)) —> x)
#define y(p) (((const struct Point *)(p)) —> y)
```

Now we can use the access macros for Circle_draw():

```c
static void Circle_draw (const void * _self)
{
    const struct Circle * self = _self;
    printf("circle at %d,%d rad %d
",
           x(self), y(self), self —> rad);
}
```

move() has static linkage and is inherited from the implementation of points. We conclude the implementation of circles by defining the type description which is the only globally visible part of Circle.c:

```c
static const struct Class _Circle = {
    sizeof(struct Circle), Circle_ctor, 0, Circle_draw
};
const void * Circle = & _Circle;
```

While it looks like we have a viable strategy of distributing the program text implementing a class among the interface, representation, and implementation file, the example of points and circles has not exhibited one problem: if a dynamically linked method such as Point_draw() is not overwritten in the subclass, the subclass type descriptor needs to point to the function implemented in the superclass. The function name, however, is defined static there, so that the selector cannot be circumvented. We shall see a clean solution to this problem in chapter 6. As a stopgap measure, we would avoid the use of static in this case, declare the function header only in the subclass implementation file, and use the function name to initialize the type description for the subclass.

4.8 Summary

The objects of a superclass and a subclass are similar but not identical in behavior. Subclass objects normally have a more elaborate state and more methods — they are specialized versions of the superclass objects.

We start the representation of a subclass object with a copy of the representation of a superclass object, i.e., a subclass object is represented by adding components to the end of a superclass object.
A subclass inherits the methods of a superclass: because the beginning of a subclass object looks just like a superclass object, we can up-cast and view a pointer to a subclass object as a pointer to a superclass object which we can pass to a superclass method. To avoid explicit conversions, we declare all method parameters with `void *` as generic pointers.

Inheritance can be viewed as a rudimentary form of polymorphism: a superclass method accepts objects of different types, namely objects of its own class and of all subclasses. However, because the objects all pose as superclass objects, the method only acts on the superclass part of each object, and it would, therefore, not act differently on objects from different classes.

Dynamically linked methods can be inherited from a superclass or overwritten in a subclass — this is determined for the subclass by whatever function pointers are entered into the type description. Therefore, if a dynamically linked method is called for an object, we always reach the method belonging to the object’s true class even if the pointer was up-casted to some superclass. If a dynamically linked method is inherited, it can only act on the superclass part of a subclass object, because it does not know of the existence of the subclass. If a method is overwritten, the subclass version can access the entire object, and it can even call its corresponding superclass method through explicit use of the superclass type description.

In particular, constructors should call superclass constructors back to the ultimate ancestor so that each subclass constructor only deals with its own class’ extensions to its superclass representation. Each subclass destructor should remove the subclass’ resources and then call the superclass destructor and so on to the ultimate ancestor. Construction happens from the ancestor to the final subclass, destruction takes place in the opposite order.

Our strategy has a glitch: in general we should not call dynamically linked methods from a constructor because the object may not be initialized completely. `new()` inserts the final type description into an object before the constructor is called. Therefore, if a constructor calls a dynamically linked method for an object, it will not necessarily reach the method in the same class as the constructor. The safe technique would be for the constructor to call the method by its internal name in the same class, i.e., for points to call `Points_draw()` rather then `draw()`.

To encourage information hiding, we implement a class with three files. The interface file contains the abstract data type description, the representation file contains the structure of an object, and the implementation file contains the code of the methods and initializes the type description. An interface file includes the superclass interface file and is included for the implementation as well as any application. A representation file includes the superclass representation file and is only included for the implementation.

Components of a superclass should not be referenced directly in a subclass. Instead, we can either provide statically linked access and possibly modification methods for each component, or we can add suitable macros to the representation file of the superclass. Functional notation makes it much simpler to use a text edi-
tor or a debugger to scan for possible information leakage or corruption of invariants.

4.9 Is It or Has It? — Inheritance vs. Aggregates

Our representation of a circle contains the representation of a point as the first component of `struct Circle`:

```c
struct Circle { const struct Point _; int rad; };
```

However, we have voluntarily decided not to access this component directly. Instead, when we want to inherit we cast up from `Circle` back to `Point` and deal with the initial `struct Point` there.

There is another way to represent a circle: it can contain a point as an aggregate. We can handle objects only through pointers; therefore, this representation of a circle would look about as follows:

```c
struct Circle2 { struct Point * point; int rad; };
```

This circle does not look like a point anymore, i.e., it cannot inherit from `Point` and reuse its methods. It can, however, apply point methods to its point component; it just cannot apply point methods to itself.

If a language has explicit syntax for inheritance, the distinction becomes more apparent. Similar representations could look as follows in C++:

```cpp
struct Circle : Point { int rad; }; // inheritance

struct Circle2 {
    struct Point point; int rad;     // aggregate
};
```

In C++ we do not necessarily have to access objects only as pointers.

Inheritance, i.e., making a subclass from a superclass, and aggregates, i.e., including an object as component of some other object, provide very similar functionality. Which approach to use in a particular design can often be decided by the is-it-or-has-it? test: if an object of a new class is just like an object of some other class, we should use inheritance to implement the new class; if an object of a new class has an object of some other class as part of its state, we should build an aggregate.

As far as our points are concerned, a circle is just a big point, which is why we used inheritance to make circles. A rectangle is an ambiguous example: we can describe it through a reference point and the side lengths, or we can use the endpoints of a diagonal or even three corners. Only with a reference point is a rectangle some sort of fancy point; the other representations lead to aggregates. In our arithmetic expressions we could have used inheritance to get from a unary to a binary operator node, but that would substantially violate the test.

4.10 Multiple Inheritance

Because we are using plain ANSI-C, we cannot hide the fact that inheritance means including a structure at the beginning of another. Up-casting is the key to reusing a
superclass method on objects of a subclass. Up-casting from a circle back to a point is done by casting the address of the beginning of the structure; the value of the address does not change.

If we include two or even more structures in some other structure, and if we are willing to do some address manipulations during up-casting, we could call the result multiple inheritance: an object can behave as if it belonged to several other classes. The advantage appears to be that we do not have to design inheritance relationships very carefully — we can quickly throw classes together and inherit whatever seems desirable. The drawback is, obviously, that there have to be address manipulations during up-casting before we can reuse methods of the superclasses.

Things can actually get quite confusing very quickly. Consider a text and a rectangle, each with an inherited reference point. We can throw them together into a button — the only question is if the button should inherit one or two reference points. C++ permits either approach with rather fancy footwork during construction and up-casting.

Our approach of doing everything in ANSI-C has a significant advantage: it does not obscure the fact that inheritance — multiple or otherwise — always happens by inclusion. Inclusion, however, can also be accomplished as an aggregate. It is not at all clear that multiple inheritance does more for the programmer than complicate the language definition and increase the implementation overhead. We will keep things simple and continue with simple inheritance only. Chapter 14 will show that one of the principal uses of multiple inheritance, library merging, can often be realized with aggregates and message forwarding.

4.11 Exercises

Graphics programming offers a lot of opportunities for inheritance: a point and a side length defines a square; a point and a pair of offsets defines a rectangle, a line segment, or an ellipse; a point and an array of offset pairs defines a polygon or even a spline. Before we proceed to all of these classes, we can make smarter points by adding a text, together with a relative position, or by introducing color or other viewing attributes.

Giving move() dynamic linkage is difficult but perhaps interesting: locked objects could decide to keep their point of reference fixed and move only their text portion.

Inheritance can be found in many more areas: sets, bags, and other collections such as lists, stacks, queues, etc. are a family of related data types; strings, atoms, and variables with a name and a value are another family.

Superclasses can be used to package algorithms. If we assume the existence of dynamically linked methods to compare and swap elements of a collection of objects based on some positive index, we can implement a superclass containing a sorting algorithm. Subclasses need to implement comparison and swapping of their objects in some array, but they inherit the ability to be sorted.