APPENDIX A

Additional Program Examples

A.1 PROCEDURAL AND OBJECT-ORIENTED ORGANIZATION

This appendix uses an extended example to illustrate some of the differences between object-oriented and conventional program organization. Sections A.1.1 and A.1.2 contain two versions of a program to manipulate geometric shapes, the second with classes and objects, the first without. The object-oriented code is written in C++, the conventional code in C. To keep the examples short, the only shapes are circles and rectangles.

The non-object-oriented code uses C structs to represent geometric shapes. For each operation on shapes, there is a function that tests the type of shape passed as an argument and branches accordingly. We refer to this program as the *typecase* version of the geometry example, as each function is implemented by a case analysis on the types of shapes.

In the object-oriented code, each shape is represented by an object. Circle objects are implemented by the circle class, which groups circle operations with the data needed to represent a circle. Similarly, the rectangle class groups the data used to represent rectangles with code to implement operations on rectangles. When an operation is done on a shape, the correct code is invoked by dynamic lookup.

Here are some general observations that you may wish to keep in mind when reading the code:

- An essential difference between the two program organizations is illustrated in the following matrix. For each function, `center`, `move`, `rotate`, and `print`, there is code for each kind of geometric shape, in this case circle and rectangle. Thus we have eight different pieces of code:

	Function			
Class	**Center**	**Move**	**Rotate**	**Print**
Circle	c_center	c_move	c_rotate	c_print
Rectangle	r_center	r_move	r_rotate	r_print

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

In the typecase version, these functions are arranged by column: the Center function contains code c_center and r_center for finding the center of a circle and a rectangle, respectively. In the object-oriented program, functions are arranged by row: The circle class contains code c_center, c_move, c_rotate, and c_print for manipulating circles. Each arrangement has some advantages when it comes to program maintenance and modification. In the object-oriented approach, adding a new shape is straightforward. The code that details how the new shape should respond to the existing operations all goes in one place: the class definition. Adding a new operation is more complicated, as the appropriate code must be added to each of the class definitions, which could be spread throughout the system. In the typecase version, the opposite is true: Adding a new operation is relatively easy, but adding a new shape is difficult.

- There is a loss of abstraction in the typecase version, as the data manipulated by rotate, print, and the other functions have to be publicly accessible. In contrast, the object-oriented solution encapsulates the data in *circle* and *square* objects. Only the methods of these objects may access this data.
- The typecase version cannot be statically type checked in C. It could be type checked in a language with a built-in typecase statement that tests the type of a struct directly. An example of such a language feature is the Simula inspect statement. Adding such a statement would require that every struct be tagged with its type, a process that requires about the same amount of space overhead as making each struct into an object.
- In the typecase version, subtyping is used in an ad hoc manner. The example is coded so that circle and rectangle have a shared field in their first location. This is a hack to implement a tagged union that could be avoided in a language providing disjoint (as opposed to C unchecked) unions.
- The running time of the two programs is roughly the same. In the typecase version, there is the space cost of an extra data field (the type tag) and the time cost, in each function, of branching according to type. In the object-oriented version, there is a hidden class or vtbl pointer in each object, requiring essentially the same space as a type tag. In the optimized C++ approach, there is one extra indirection in determining which method to invoke, which corresponds to the switch statement in the typecase version. A Smalltalk-like implementation would be less efficient in general, but for methods that are found immediately in the subclass method dictionary (or by caching), the run-time efficiency may be comparable.

A.1.1 Shape Program: Typecase Version

```
#include <stdio.h>
#include <stdlib.h>

/* We use the following enumeration type to "tag" shapes.  */
/* The first field of each shape struct stores what particular */
/* kind of shape it is.                                      */

enum shape_tag {Circle, Rectangle};
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
/* The following struct pt and functions new_pt and copy_pt are */
/* used in the implementations of the circle and rectangle    */
/* shapes below.                                              */

struct pt {
  float x;
  float y;
};

struct pt* new_pt(float xval, float yval) {
   struct pt* p = (struct pt *)malloc(sizeof(struct pt));
   p->x = xval;
   p->y = yval;
   return p;
};

struct pt* copy_pt(struct pt* p) {
   struct pt* q = (struct pt *)malloc(sizeof(struct pt));
   q->x = p->x;
   q->y = p->y;
   return q;
};

/* This struct is used to get some static type checking in the   */
/* operation functions (center, move, rotate, and print) below. */

struct shape {
    enum shape_tag tag;
};

/* The following circle struct is our representation of a circle. */
/* The first field is a type tag to indicate that this struct    */
/* represents a circle. The second field stores the circle's     */
/* center and the third its radius.                              */

struct circle {
    enum shape_tag tag;
    struct pt* cnter;
    float radius;
 };
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
/* The function new_circle creates a circle struct from a given */
/* center point and radius. It sets the type tag to "Circle".*/

struct circle* new_circle(struct pt* cn, float r) {
    struct circle* c = (struct circle*)malloc(sizeof(struct circle));
    c->cnter=copy_pt(cn);
    c->radius=r;
    c->tag=Circle;
    return c;
};

/* The following rectangle struct is our representation of a  */
/* rectangle. The first field is used to indicate that this   */
/* struct represents a rectangle. The next two fields store   */
/* the rectangle's topleft and bottom right corners.          */

struct rectangle {
    enum shape_tag tag;
    struct pt* topleft;
    struct pt* botright;
};

/* The function new_rectangle creates a rectangle in the location */
/* specified by parameters tl and br. It sets the rectangle's     */
/* type tag to "Rectangle".                                       */

struct rectangle* new_rectangle(struct pt* tl, struct pt* br) {
    struct rectangle* r = (struct rectangle*)malloc(sizeof(struct rectangle*));
    r->topleft=copy_pt(tl);
    r->botright=copy_pt(br);
    r->tag=Rectangle;
    return r;
};

/* The center function returns the center point of whatever shape */
/* it is passed. Because the code to compute the center of a      */
/* shape depends on whether the shape is a Circle or a Rectangle, */
/* the function consists of a switch statement that branches      */
/* according to the type tag of the shape s. Within the Circle    */
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
/* case, for example, we know the shape in question is actually a  */
/* circle, and hence that it has a "cnter" component storing    */
/* the circle's center. Note that we need to insert a typecast    */
/* to instruct the compiler that we have a circle and not just a    */
/* shape. Note also that this program organization depends on      */
/* the typetags, which are simply struct fields, being set     */
/* correctly. If some programmer incorrectly modifies a type tag    */
/* field, the program will no longer work and the problem can not */
/* be detected at compile time because of the typecasts.      */

struct pt* center (struct shape* s) {
   switch (s->tag) {
   case Circle: {
     struct circle* c = (struct circle*) s;
     return c->cnter;
   };

   case Rectangle: {
     struct rectangle* r = (struct rectangle*) s;
     struct pt* p = new_pt((r->botright->x - r->topleft->x)/2,
                 (r->botright->x - r->topleft->x)/2);
     return p;
   };
  };
};

/* The move function moves the shape s dx units in the x-direction */
/* and dy units in the y-direction. Because the code to move a    */
/* shape depends on the kind of shape, this function is a switch  */
/* statement that branches depending on the value of the "tag"     */
/* field. Within the individual cases, typecasts are used to      */
/* convert the generic shape s to a circle or rectangle as        */
/* appropriate.                                          */

void move (struct shape* s,float dx, float dy) {
   switch (s->tag) {
       case Circle: {
         struct circle* c = (struct circle*) s;
         c->cnter->x += dx;
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
        c->cnter->y += dy;
        break;
      };

      case Rectangle: {
        struct rectangle* r = (struct rectangle*) s;
        r->topleft->x += dx;
        r->topleft->y += dy;
        r->botright->x += dx;
        r->botright->y += dy;
      };
  };
};

/* The rotate function rotates the shape s ninety degrees. Since  */
/* the code depends on the kind of shape to be rotated, this    */
/* function is a switch statement that branches according to the */
/* type tag.                                          */
void rotate (struct shape* s) {
  switch (s->tag) {
      case Circle:
        break;
      case Rectangle: {
        struct rectangle* r = (struct rectangle*)s;
        float d;
         d = ((r->botright->x - r->topleft->x) -
           (r->topleft->y - r->botright->y))/2.0;
         r->topleft->x += d;
         r->topleft->y += d;
         r->botright->x -= d;
         r->botright->y -= d;
        break;
      };
  };
};

/* The print function prints a descriptive statement about the  */
/* location and kind of shape s. This function is again a switch */
/* statement that branches according to the "tag" field.       */
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
void print (struct shape* s) {
  switch (s->tag) {
     case Circle: {
       struct circle* c = (struct circle*) s;
       printf("circle at %.1f %.1f radius %.1f \n",c->cnter->x,
            c->cnter->y, c->radius);
       break;
     };

     case Rectangle:{
       struct rectangle* r = (struct rectangle*) s;
       printf("rectangle at %.1f %.1f %.1f %.1f \n",
          r->topleft->x, r->topleft->y,
          r->botright->x, r->botright->y);
     };
  };
};

/* The body of this program just tests some of the above functions. */

main() {

pt* origin = new_pt(0,0);
pt* p1 = new_pt(0,2);
pt* p2 = new_pt(4,6);

shape* s1 = new_circle(origin,2);
shape* s2 = new_rectangle(p1,p2);

print(s1);
print(s2);

rotate(s1);
rotate(s2);

move(s1,1,1);
move(s2,1,1);

print(s1);
 print(s2);

};
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

A.1.2 Shape Program: Object-Oriented Version

```
#include <stdio.h>

/* The following class pt is used in the implementations of */
/* the shape objects below. Since pt is a class in this    */
/* version of the program, instead of simply a struct, we  */
/* may include the "new_pt" and "copy_pt" functions of     */
/* the typecase version within the pt class. For           */
/* convenience, both these functions are named "pt";       */
/* they are differentiated by the static types of their    */
/* arguments.                                              */

class pt {
public:
 float x;
 float y;
 pt(float xval, float yval) {x = xval; y=yval;};
 pt(pt* p) {x = p->x; y = p->y;};
};

/* Class shape is an example of a "pure abstract base class",      */
/* which means that it exists solely to provide an interface to    */
/* classes derived from it. Since it provides no implementations   */
/* for the methods center, move, rotate, and print, no "shape"     */
/* objects can be created. Instead, we use this class as a base    */
/* class. Our circle and rectangle shapes will be derived from     */
/* it. This class is useful because it allows us to write          */
/* functions that expect "shape" objects as arguments. Since       */
/* our circles and rectangles are subtypes of shape, we may pass   */
/* them to such functions in a type-safe way.                      */

class shape {
public:
 virtual pt* center()=0;
 virtual void move(float dx, float dy)=0;
 virtual void rotate()=0;
 virtual void print()=0;
};
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
/* Class circle, defined below, consolidates the code for circles */
/* from the center, move, rotate, and print functions in the    */
/* typecase version. It also contains the object constructor    */
/* "circle", corresponding to the function "new_circle" in      */
/* the typecase version. Note that in this version of the    */
/* program, the compiler guarantees that the circle move method, */
/* for example, is called on a circle. We do not have to rely on */
/* programmers keeping the tag field accurate for the program to  */
/* work correctly.                                              */

class circle : public shape {
    pt* cnter;
    float radius;
   public:
    circle(pt* cn, float r)
        {cnter = new pt(cn); radius = r;};
    pt* center()
        {return cnter;};
    void move(float dx, float dy)
        {cnter->x += dx;
         cnter->y += dy;
        };
    void rotate() {};
    void print ()
        { printf("circle at %.1f %.1f radius %.1f \n",
                cnter->x, cnter->y, radius);
        };
};

/* Class rectangle, defined below, consolidates the code for */
 /* rectangles from the center, move, rotate, and print functions */
/* in the typecase version. It also contains the object         */
/* constructor "rectangle", corresponding to the function    */
/* new_rectangle in the typecase version.                    */

class rectangle : public shape {
  private:
    pt* topleft;
    pt* botright;
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
  public:
    rectangle(pt* tl, pt* br)
      {topleft=new pt(tl);botright=new pt(br);};
    pt* center()
      {pt* p = new
       pt((botright->x - topleft->x)/2, (botright->x - topleft->x)/2);
       return p;};
    void move(float dx,float dy)
      {topleft->x += dx;
       topleft->y += dy;
       botright->x += dx;
       botright->y += dy;
      };
    void rotate ()
      { float d;
       d = ((botright->x - topleft->x) -
            (topleft->y - botright->y))/2.0;
       topleft->x += d;
       topleft->y += d;
       botright->x -= d;
       botright->y -= d;
      };
    void print ()
      { printf("rectangle coordinates %.1f %.1f %.1f %.1f \n",topleft->x,
              topleft->y, botright->x, botright->y);
      };
};

main() {

pt* origin = new pt(0,0);
pt* p1 = new pt(0,2);
pt* p2 = new pt(4,6);

shape* s1 = new circle(origin, 2 );
shape* s2 = new rectangle(p1, p2);

s1->print();
s2->print();
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press

```
s1->rotate();
s2->rotate();

s1->move(1,1);
s2->move(1,1);

s1->print();
s2->print();

}
}
```

https://doi.org/10.1017/CBO9780511804175.017 Published online by Cambridge University Press