Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 00 (2019) 1–13
Science of Computer Program-ming

Inheritance pattern in C

Nikolai G. Lehtinen[a,*]

[a]Birkeland Centre for Space Science, University of Bergen, Bergen, Norway

Abstract

We present a tutorial on how inheritance may be implemented in C programming language, which does not have built-in support for object-oriented programming (OOP). The main points of this tutorial are:

1. OOP is just a set of patterns, as all other programming paradigms.
2. Inheritance is same as an interface.
3. The "base" class must have a handle to the "derived" class.
4. It is necessary to use the "virtual function table" (VFT) for class members.

Previously suggested programming patterns did not stress all of these points and thus lead to limited and suboptimal solutions, e.g., multiple inheritance was not supported. We conclude that C is fully suitable for bug-free OOP which does not rely on obscure language features, even though at expense of a few extra lines of code.

Keywords: OOP, inheritance, C programming language, tutorial
2010 MSC: 68N15, 68N19

PROGRAM SUMMARY
Program Title: Example of inheritance implementation in C
Licensing provisions: CC0 1.0
Programming language: C
Supplementary material: `c_inheritance.zip` *(C code)*
Current version: January 16, 2019
Nature of problem: Implementation of inheritance in C programming language
Solution method: Using appropriate design patterns
Additional comments including Restrictions and Unusual features: None

1. Introduction

C language has a long history but is still relevant and widely used today [1]. It is a simple but powerful language [2]. It is used not only for low-level programming work which is impossible to perform in more advanced languages, but also for big projects like CPython, consistently being among the top used computer languages [3]. Thus, we must seriously look into implementing modern programming paradigms into C in a readable and extensible manner [4, 5].

The main idea of various programming techniques is to simplify the programmer's job. Because a human cannot hold more than seven (or a similar small number) things in its head simultaneously, some modularity or separation of work into smaller pieces is needed.

*Corresponding author.
E-mail address: nlehtinen at gmail dot com

> The point of programming languages is to prevent our poor frail human brains from being overwhelmed by a mass of detail. [6]

There are various programming paradigms which describe such "modularity." After thinking a bit, we realize that every paradigm is just a coding pattern. Even subroutines used to be a pattern once, but now they are part of virtually all programming languages [7]. Classes and objects in the object-oriented programming paradigm (OOPP) are also patterns. Classes are modules, and may be implemented as such in C. It is not necessary to simulate all of the OO features down to syntax: e.g., it is completely unnecessary to put function pointers inside **struct**s in order to simulate methods, as was suggested, e.g., in [8, sec. 34.4]. The methods can be functions in the same module as the **struct** in question. In this paper, we adopt the following convention: instead of the usual C++ syntax `object.function(arg)` where `object` belongs to a `Class`, we use C syntax `ClassFunction(object,arg)`. Declarations of such functions are in `Class.h`, and definitions are in `Class.c` (usually).

We are going to discuss the C implementation of the OOPP inheritance pattern. We must admit that we do not claim complete originality of the ideas presented here. In particular, we draw on ideas presented in the implementation of the Reactor pattern in [4, part 5]. We strongly recommend the book [4] to the reader, as it demonstrates the power of C language and its competitiveness in the modern industry with languages that are allegedly more advanced because they already have the OOPP set of patterns built into them.

We treat the "inheritance" pattern the same as an "interface." It is often recommended that "derived" classes should inherit only from abstract classes which thus serve as "interfaces." In order to extend a concrete class (usually for the purpose of code re-use), another pattern is usually recommended, namely "composition" [9]. The "base" class is included as a member of the "derived" class and the method calls on the "derived" class are forwarded (or *delegated*) to the "base." Thus, we have to be very careful in order not to abuse the inheritance capabilities when they are available.

Availability of these alternatives leads to different approaches to inheritance in different computer languages. E.g., Java, beside syntax for inheritance, also has separate syntax for interfaces. To avoid conflicting inherited implementations, it does not allow multiple inheritance. However, it still allows multiple interfaces [10].

Abusing inheritance may lead to a complicated or inconsistent code. An example of bad usage of subtyping (inheritance), in our humble opinion, can even be found in the classical C++ book [11, Section 3.2.4]:

> A smiley face is a kind of a circle.

We should check it against the *Liskov substitution principle* [12]:

> *Subtype Requirement*: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

We find that the properties ϕ_1 = "does not have eyes" and ϕ_2 = "does not have a mouth" are valid for a "circle" but not valid for a "smiley face." Thus, a "smiley face" is *not* a subtype of a "circle." In theory, this can be fixed if we define the type "circle" to have a property ψ = "has something inside," where the "something" is just "nothing" for a proper circle but substituted for "mouth and eyes" for the "smiley face." But when we write the "circle" class, are we really going to have so much foresight as to guess that we should allow it to have something inside it? Maybe the class from which the "smiley face" inherits should be called not "circle" but something like "round thing," which leaves more freedom for interpretation. Then both "circle" and "smiley face" can inherit from the "round thing."

A similar noncompliance with Liskov substitution principle is committed in [13, ch. 4] where a subclass Circle is derived from a base class Point. A circle is definitely not a point! If we have a collection of various shapes like circles and rectangles, and we want to store them in an array with uniform access, is it really an array of points?

These examples show us that we should be careful when choosing inheritance over composition, and even the giants of the computer world are not immune to errors. In Section 2.3 we provide a C example of a `Smiley` class which does not inherit from `Circle` but delegates function `SmileyGetRadius` to a member circle.

Inheritance pattern implementation (as an interface) presented in this paper differs from many similar implementations that can be found on the internet. An important concept that we implement but which is completely ignored by others, is that the interface (base) *must* have the handle of the (derived) object, in order to have access to object-specific functions and members. Some authors [14, 15] go around this by using so-called *type punning* [16], i.e., implementing base and derived objects in such a way that they occupy the same space in memory, thus forming a kind

of a **union**. Type punning is widely used in projects, e.g., in GTK [17]. However, it intimately depends on how C interprets **struct**s and may be subject to bugs [18]. Moreover, type punning allows only a single base and is therefore *incompatible with multiple inheritance*.

Unlike the type-punning approach [14, 15], we do *not* consider the "derived" objects to be objects of the "base" class, so we never perform true typecasting between the derived and base objects. (However, some analogy to typecasting is discussed in Section 3.) The difference in types is obvious if we think of the "base" in terms of an interface—the real object, of course, is not of the interface class. Therefore, our approach is more logical.

2. An example of an inheritance pattern in C

Attached to this article, there is a C code example with the described implementation. The compilation instructions are given in the comments in the beginning of file `main.c`. Below, we go through the details of what the code exactly does.

Consider a program that deals with various planar shapes, like `Circle`, `Rectangle`, etc. One can find features which may be attributed to all of the different kinds of shapes, e.g., the position on the plane (x, y) or color with which the shape is drawn. The various shapes also have an area and a perimeter, which are calculated in a different manner for different types. All of the common features may be put in a base class, or an interface, which we call `Shape`. The `Shape` part provides a uniform way of accessing the instances of derived classes. For example, we can create an array of `Shapes` which are actually interfaces to more complicated derived class instances. By iterating over this array, we draw different shapes and calculate their areas and perimeters. An appropriate method will be called for each object (another way to say in OOP terminology is that each object *receives a message*). Thus, we do not have to worry about determining the objects' respective classes during the iteration.

We do not include in the text of this paper the discussion of various functions which are present in the attached code but should only be used for debugging, or to demonstrate bad programming practices. In the code, these parts are accompanied by appropriate warnings.

2.1. The interface class `Shape`

There are two levels of availability of details of this class to the outside world: for the end user and for the derived classes. These two levels correspond to the public and protected levels in C++, correspondingly.

2.1.1. Public members

The user access is described in `Shape.h`. The implementation details are hidden in an *abstract data type* (ADT):

```
// A shortcut for the handle
typedef struct ShapeInterfaceT *Shape;
```

We will return to discussing **struct** `ShapeInterfaceT` in the next Subsubsection.

In all functions below the argument of type `Shape` can be either of the derived or the pure-base type. By *pure-base* object we mean an interface without an initialized derived-object part. The user is provided with public constructor and destructor which must be called explicitly:

```
// Constructor and destructor
// "New" calls should match "Delete" + "Replace" calls to prevent memory leaks
Shape ShapeNew(float x, float y);
void ShapeDelete(Shape *shapePtr); // sets *shapePtr to NULL
```

We also provide an analog of the copy-assignment operator in C++ which replaces an old value with the new, thereby deleting the old value:

```
void ShapeReplace(Shape *old_shapePtr, Shape new_shape);
```

In order to prevent memory leaks, all `ShapeNew` operations should match corresponding `ShapeDelete` or `ShapeReplace` operations, which will be clear from the implementation details.

The user interface is provided by a set of functions. Here are object member access functions (getters and setters, e.g. for coordinates):

```
// 1. Getters and setters for data members
float ShapeGetX(Shape shape);
float ShapeGetY(Shape shape);
void ShapeSetX(Shape shape, float new_x);
void ShapeSetY(Shape shape, float new_y);
```

We can also access class members (common for a whole base or derived class):

```
// .. and for class members
char *ShapeGetClassName(Shape shape); // to demonstrate a class data member, NOT to do any dispatch over subclasses!
int ShapeCounterOfThisType(Shape shape); // virtual class method (?) Determines class by an instance.
```

These functions determine the class of the given instance `shape` and return an appropriate class data member. If the class is the base class (`shape` is a pure-base object), the first function returns `"Shape"` and the second function returns the total number of interfaces (sum of the numbers of all derived plus pure-base objects). If the class is a derived class, these functions correspondingly return an appropriate name (e.g., `"Circle"`, `"Rectangle"` etc) and the total number of relevant objects (circles, rectangles, etc).

It is possible to access class members for a given class, too. Such functions do not need an instance (object), but the class name has to be specified explicitly in the name of the function. E.g., the counter of all `Shapes` may be accessed with

```
// in Shape.h
int ShapeCounterTotal(void); // class method
```

which does not need a pure-base `shape` as an argument.

Finally, there are methods:

```
// 2. Methods (interfaces to overloaded virtual methods)
float ShapeArea(Shape shape);
float ShapePerimeter(Shape shape);
void ShapeDraw(Shape shape,int color);
// A method which is not a bare interface
void ShapeTranslation(Shape shape, float dx, float dy);
```

The functions for area and perimeter calculation and drawing a shape are overloaded by subclasses so they can be considered virtual functions. If `shape` is a pure-base object, the call to a virtual function causes an error, and if it is a derived-class object, an appropriate overloaded function is called. The last function, `ShapeTranslation`, utilizes only information available in the base class. Its code is thus fully provided by the base class `Shape`, and is not overloaded by any of the subclasses. Therefore, it cannot be called a virtual function.

2.1.2. *Protected members*

The **struct** `ShapeInterfaceT` is implemented in `ShapeProtected.h`, a file which is used by derived classes, but the end user *does not* have access to. The implementation details are thus hidden from the user completely, which resembles what is known as "pimpl" idiom in C++ [19, Item 22]. The **struct** has the following fields:

```
// The implementation struct must be available to derived classes
struct ShapeInterfaceT {
    // 1. The access to the derived object, or equivalently, the object using the interface
    void *instance; // must be void* because we don't know its type yet
    // Interface elements
    // 2. Object members (usually, data members which are non−virtual by their nature)
    float x, y;
    // 3. Class members: data and methods.
    struct ShapeClassMembers *vft; // all interface methods collected in a virtual function table
}
```

The pointer `instance` provides the access to the derived object (or, equivalently, the object which uses the interface `Shape`). Its type is not known in advance, so we have to use **void** `*`. Next, there are *instance* data members **float** `x, y`,

which are individual for each object. Finally, there is a field that provides access to *class* data members and methods `struct ShapeClassMembers *vft`, which is called (mostly for historical reasons) a *virtual function table* (VFT, [20]).

If we want to track class data members which change their values, it is *necessary* to introduce VFT because these *cannot* be stored in `struct ShapeInterfaceT` which is individual to each instance of a class. A VFT is common for the whole class (base or derived) and stores all the class members. It is implemented as

```
struct ShapeClassMembers {
    char *class_name; // class data member ("static" in C++)
    int obj_counter;
    float (*area)(void* instance);
    float (*perimeter)(void* instance);
    void (*draw)(void* instance, int color);
    void (*destruct)(void *instance);
};
```

There are class data members `char *class_name`, `int obj_counter` and methods `area`, `perimeter`, `draw`, `destruct`. The methods are virtual functions, which become concrete functions when a derived class is defined, and are accessed by the user through `ShapeArea` etc., provided in `Shape.h`. In principle, we can also have methods (functions) which are specific to an *instance* (something like a derived singleton class), which *have* to be stored in `struct ShapeInterfaceT`, but we do not have them in this example.

2.1.3. Implementation details (private members)

The methods of the abstract `Shape` class are implemented in file `Shape.c`. The creator of the `Shape` looks like this:

```
static void ShapeReset(Shape shape) {
    // Auxiliary function
    shape->instance = NULL;
    shape->vft = &ShapeVFT;
}

// "New" calls like this should match "Delete" + "Replace" calls
Shape ShapeNew(float x, float y) {
    Shape shape = malloc(sizeof(struct ShapeInterfaceT));
    ShapeReset(shape);
    shape->vft->obj_counter++; // counting virtual shapes only
    shape->x = x; shape->y = y;
    return shape;
}
```

The object counter increment here `shape->vft->obj_counter++` applies to the base class. Another advantage of using VFT is that we do not need to copy all the class members and function pointers into each new object created, but only copy the address of the singleton `ShapeVFT`.

This VFT stores the pure-base class members and methods:

```
static struct ShapeClassMembers ShapeVFT = {
    .class_name = "Shape", // or could be just NULL
    .obj_counter = 0, // we count the virtual shapes separately
    .area = VirtualShapeArea,
    .perimeter = VirtualShapePerimeter,
    .draw = VirtualShapeDraw,
    .destruct = VirtualShapeDestruct
};
```

The placeholder functions `VirtualShapeArea` etc., are implemented in order to avoid segfaults if the virtual functions are called by mistake. For example:

```
static float VirtualShapeArea(void *instance) {
    fprintf(stderr,"error: call to virtual function ShapeArea\n");
```

```
    return 0.;
}
```

Their output may be used for diagnostics instead of just terminating the execution. (Exiting on error is turned on with `make OPT=-DEXIT_ON_VIRTUAL_CALL` when compiling the code.) If the `Shape` interface is not attached to any concrete (derived) class, then the `instance` argument of these virtual functions must be `NULL`, so that we can check it as an extra level of safety. (Checking for such internal consistencies is turned on with `make OPT=-DCHECK_INTERNAL`.) The virtual functions are declared as **static** and are invisible outside the file.

The access functions available to the user are implemented as one-liners:

```
// Methods
float ShapeArea(Shape shape) { return shape->vft->area(shape->instance); }
float ShapePerimeter(Shape shape) { return shape->vft->perimeter(shape->instance); }
void ShapeDraw(Shape shape,int color) { shape->vft->draw(shape->instance, color); }

// Data member access (getters and setters), making them available to the user
float ShapeGetX(Shape shape) { return shape->x; }
float ShapeGetY(Shape shape) { return shape->y; }
void ShapeSetX(Shape shape, float new_x) { shape->x = new_x; }
void ShapeSetY(Shape shape, float new_y) { shape->y = new_y; }

// Class member access
char *ShapeGetClassName(Shape shape) { return shape->vft->class_name; }
int ShapeCounterOfThisType(Shape shape) { return shape->vft->obj_counter; }
int ShapeCounterTotal(void) { return ShapeVFT.obj_counter; }
```

The usage of the `instance` pointer is completely hidden from the user. The only disadvantage of the VFT design pattern that we see here is that we have to perform an extra operation of dereferencing and member access, namely `shape->vft->area` etc., which would not be needed if `area` were part of the **struct** `ShapeInterfaceT *shape`. However, this is far outweighed by the fact that the variable class members (like the object counters) are impossible without VFT.

An example of a non-virtual method (i.e. one that does not use `shape->vft`):

```
void ShapeTranslation(Shape shape, float dx, float dy) {
    shape->x += dx; shape->y += dy;
    printf("Moving shape by (dx=%f, dy=%f) to the new position at (%f, %f)\n", dx, dy, shape->x,
    shape->y);
}
```

The destructor of the interface `Shape` should take care of calling an appropriate destructor for the derived part of the class and freeing the memory occupied by the interface itself. The destructors for the derived parts are hidden from the user. The reason for this is discussed in Section 3. The destructor is implemented in the following way:

```
void ShapeDelete(Shape *shapePtr) {
    // We don't check if shapePtr is NULL, we should not use addresses to anything but Shape
    Shape shape = *shapePtr;
    if (!shape) { fprintf(stderr,"error: double delete of the shape interface\n"); exit(1); }
    void *inst = shape->instance;
    if (inst) {
        shape->vft->destruct(inst); // destruct the derived part
        ShapeReset(shape); // important, restore virtual shape class members
    } // if inst==NULL, then it means that some other interface already destructed it
    shape->vft->obj_counter--; // counting virtual shapes only
    free(shape);
    *shapePtr = NULL; // this way we can tell it does not point to anything anymore
}
```

The decrement applies to the base class because `shape->vft` is restored to point to `ShapeVFT` after destruction of the derived object part. The `Shape` which is passed to it by reference is changed to `NULL` in order to indicate that the object

has been destructed and to ensure that any future attempts to use it will give an error instead of a segfault. There is more discussion of safe programming practices in Section 3. It is not an error when `ShapeDelete` is called without a derived part. Its absence could be due either to the fact that it was never created, or, in a more complicated case with "multiple inheritance", i.e., multiple interfaces, that it had been destructed by another interface.

Sometimes we need to replace an object, which includes the destruction of the old one. This is equivalent to the C++ copy-assignment operator:

```
void ShapeReplace(Shape *old_shapePtr, Shape new_shape) {
    ShapeDelete(old_shapePtr);
    *old_shapePtr = new_shape;
}
```

For each `malloc`, there should be a `free`. Thus, each `ShapeNew` operation should have a matching `ShapeDelete` or `ShapeReplace`. This will be our safe programming rule in order to avoid memory leaks. More safety rules are discussed in Section 3.

2.2. *A derived class, on the example of `Circle`*

The publicly available info about `Circle` can be very limited. Here is what is in `Circle.h`:

```
#include "Shape.h"

typedef struct CircleCDT *Circle;

// Access to the interface (base class)
Shape CircleShape(Circle circle);
// Derived class constructor when the base "shape" is already constructed.
// There is no need for matching destructor because the deletion of the
// derived part is done automatically when the interface (base) is deleted
Circle CircleCreate(Shape shape, float radius);
// Circle methods
int CircleCounter(void); // class method
float CircleGetRadius(Circle circle);
void CircleSetRadius(Circle circle, float new_radius);
```

The user-accessible `Circle` is an ADT, and the internals stored in the *concrete data type* (CDT) are not user-accessible.

The implementation is in file `Circle.c`. The derived object stores the base plus its own members:

```
#include "Shape.h"
#include "ShapeProtected.h"
#include "Circle.h"
struct CircleCDT {
    Shape shape;
    float radius;
};
```

A simple access to the base-class part is provided by a one-liner which may be considered a *typecasting operator* from the derived to the base class:

```
Shape CircleShape(Circle circle) { return circle->shape; }
```

The creation of the `Circle` object is done only after the underlying interface `Shape` has been created. It is implemented in the following way:

```
Circle CircleCreate(Shape shape, float radius) {
    Circle circle = malloc(sizeof(struct CircleCDT));
    // Attach the base (this must be in every derived class)
    if (shape) circle->shape = shape;
    else { fprintf(stderr,"error: base not initalized in CircleCreate\n"); exit(1); }
    shape->instance = circle;
```

```
    // Overload the inherited members (virtual functions)
    shape->vft = &CircleVFT;
    // Done with preliminaries! Now, the proper initialization
    // 1. Do whatever is needed with derived class members
    shape->vft->obj_counter++;
    // 2. Initialize the derived object members
    circle->radius = radius;
    printf("Creating a Circle at (%f, %f) of radius=%f\n", shape->x, shape->y, radius);
    return circle;
}
```

The VFT of the base class is replaced in the creator by the class's own VFT:

```
static struct ShapeClassMembers CircleVFT = {
    .class_name = "Circle",
    .obj_counter = 0, // not constant!
    .area = CircleArea,
    .perimeter = CirclePerimeter,
    .draw = CircleDraw,
    .destruct = CircleDestruct
};
```

The increment `shape->vft->obj_counter++` in the creator applies to the number of `Circle` objects because `shape->vft` points to `CircleVFT`. Since the creator takes a base object and returns a derived object, in may be looked at as a *typecasting operator* (which, however, takes additional arguments like `radius`).

We implement concrete functions `CircleArea`, `CirclePerimeter`, `CircleDraw` and `CircleDestruct` using the appropriate signatures from the `Shape` interface. They are declared as **static** and are therefore not visible to the user directly, but are accessed through `Shape`'s interface. For example:

```
// This destructs only the derived additions
static void CircleDestruct(void *instance) {
    Circle circle = (Circle) instance;
    Shape shape = circle->shape;
    printf("Destructing circle at (%f,%f)\n", shape->x, shape->y);
    shape->vft->obj_counter--;
    // Could be more work, if anything extra was allocated in "CircleCreate"
    free(circle);
    // Keep "shape", this method destructs only the "derived" stuff
}
```

As we said, this destructor is hidden from the user and is only called by `ShapeDelete` (see the code in the previous Subsection). Notice the decrement `shape->vft->obj_counter--` which only applies to the number of `Circle` objects because `shape->vft` points to `CircleVFT`. An explicit destructor of a `Circle` may be dangerous because the `shape` member inside it may have other references to it which cannot be updated to have a `NULL` value. It is necessary to have only a single useable reference to each `Shape` object for safe programming, as discussed in Section 3.

We have access to the total number of instances of a given derived class:

```
int CircleCounter(void) { return CircleVFT.obj_counter; }
```

It is also possible to access it with **int** `ShapeCounterOfThisType(Shape shape)` if `shape` is an interface to a `Circle`.

Other derived classes implemented in the attached code are `Rectangle` (which is also rather simple, like `Circle`; calculations of the area and perimeter have, of course, different implementations), and `Smiley`, about which in detail in the next Subsection.

2.3. The `Smiley` class: both composition and interface/inheritance

The `Smiley` class is also derived from `Shape` and the code for it has the same pattern as `Circle`. As suggested in Section 1, it is *not* a derived class of the `Circle` class. Instead, we use composition, and store the big circle as member `face`:

```
// This is in Smiley.c
#define NFACEELEM 4
struct SmileyCDT {
    Shape shape;
    Shape elements[NFACEELEM]; // Face, eyes and mouth
    bool isForeground[NFACEELEM]; // true if foreground, false if background
    Circle face; // composition over inheritance!
    Circle eyes[2];
    Rectangle mouth;
};
```

(The `bool` type is defined in `<stdbool.h>`.) In `Smiley.h` we have, of course, the ADT definition `typedef struct SmileyCDT *Smiley`.

Instead of inheriting the `Circle` methods, we *delegate* them to the member `face`. However, this is not always possible. In the user-accessible `Smiley.h` we have

```
// Circle methods
float SmileyGetRadius(Smiley smiley);
void SmileySetRadius(Smiley smiley, float new_radius);
```

In the implementation `Smiley.c` we have a one-liner delegating the functionality to a member:

```
float SmileyGetRadius(Smiley smiley) {
    return CircleGetRadius(smiley->face);
}
```

The code for the setter `SmileySetRadius` would not be, however, a simple one-liner like this because all the facial elements need to be resized and re-positioned when the radius of the face is changed. Code re-use, which we get by default with inheritance, is impossible here. This function is not yet implemented in the current version of the code, and the reader may do it as an excercise.

In the constructor, we create each facial feature. When we create the eyes and the mouth, we just scale them by the given overall size `radius`:

```
Smiley SmileyCreate(Shape shape, float radius) {
    int i;
    double x, y;
    Smiley smiley = malloc(sizeof(struct SmileyCDT));
    // Attach the base (this must be in every derived class)
    if (shape) smiley->shape = shape;
    else { fprintf(stderr,"error: base not initalized in SmileyCreate\n"); exit(1); }
    shape->instance = smiley;
    // Overload the inherited members (virtual functions)
    shape->vft = &SmileyVFT;
    // Done with preliminaries! Now, the proper initialization
    // 1. Do whatever is needed with derived class members
    shape->vft->obj_counter++;
    // 2. Initialize the derived object members
    x = shape->x; y = shape->y;
    printf("Creating a Smiley at (%f, %f) of radius=%f\n", x, y, radius);
    smiley->face = CircleCreate(ShapeNew(x, y), radius);
    smiley->eyes[0] = CircleCreate(ShapeNew(x - radius/3.,y), radius/10.);
    smiley->eyes[1] = CircleCreate(ShapeNew(x + radius/3.,y), radius/10.);
    smiley->mouth = RectangleCreate(ShapeNew(x, y-radius/2.), radius/4., radius/20.);
    // Associate the shapes with elements
    smiley->elements[0] = CircleShape(smiley->face);
    smiley->elements[1] = CircleShape(smiley->eyes[0]);
    smiley->elements[2] = CircleShape(smiley->eyes[1]);
    smiley->elements[3] = RectangleShape(smiley->mouth);
```

```
    smiley->isForeground[0] = true;
    for (i = 1; i < NFACEELEM; i++) smiley->isForeground[i] = false;
    return smiley;
}
```

The facial features belonging to different classes (`Circle` for eyes and face and `Rectangle` for the mouth) have to be separate fields of the **struct** `SmileyCDT`. However, their common interfaces of type `Shape` are all stored in the array `elements` which allows us to iterate over them for drawing or other operations.

The methods to calculate the perimeter and area call on the appropriate methods for the elements of the face. For example, to calculate the area we must *subtract* the areas of the holes (the mouth and the eyes, which have `isForeground[i]==false`):

```
static float SmileyArea(void *instance) {
    Smiley smiley = (Smiley) instance;
    float area = 0;
    int i;
    for (i = 0; i < NFACEELEM; i++)
        // areas of eyes and mouth are subtracted
        area += (smiley->isForeground[i] ? 1. : -1.)*ShapeArea(smiley->elements[i]);
    return area;
}
```

`SmileyPerimeter`, `SmileyDraw` and the destructor `SmileyDestruct` are implemented analogously, by cycling over all facial features (see the code for details).

2.4. The user access example in `main.c`

Let us consider an examle of how the user may use the objects. First, we have to include appropriate header files:

```
#include "Shape.h"
#include "Circle.h"
#include "Rectangle.h"
#include "Smiley.h"
#define N 5
int main(void)
{
    //...
}
```

We defined `N`, the number of `Shapes`, to be five. Let us see what happens inside function `main`:

1. Creation

```
    // ... variable declarations etc.
    Shape shapes[N];
    for (i = 0; i < N; i++) { shapes[i] = ShapeNew(x=10.*i+1.2, y=10.*i+2.4); }
    // Create concrete objects
    CircleCreate(shapes[0], radius=2.);
    RectangleCreate(shapes[1], width=3., height=4.);
    SmileyCreate(shapes[2], radius=5.);
```

   All the diverse `Shapes` are stored in the same array. Namely, we have one `Circle`, one `Rectangle` and two uninitialized base `Shapes`. The purpose of not creating any concrete objects for `shapes[3]`, `shapes[4]` is to test error recovery when they are accessed by accident.

2. User access: drawing, calculating area and perimeter etc.

```
    for (i = 0; i < N; i++) {
        Shape shape = shapes[i];
        if (!shape) continue;
        int color = i+1;
```

```
    printf("Shape #%d at (%f,%f) of color %d\n", i, ShapeGetX(shape), ShapeGetY(shape), color
    );
    printf("This shape: area=%f, perimeter=%f\n", ShapeArea(shape), ShapePerimeter(shape));
    ShapeDraw(shape, color);
}
```

Virtual placeholder functions are called for `Shapes` that do not have concrete objects associated with them. As discussed above, the output from them may be used for diagnostics of such errors.

3. Clean-up

```
for (i = 0; i < N; i++) { ShapeDelete(&(shapes[i])); }
```

The elements of the array `shapes` are now all `NULL`, in order to prevent future errors.

4. Access to class members

```
// ...
printf("There are %d objects of this shape (%s)\n",
    ShapeCounterOfThisType(shape),ShapeGetClassName(shape));
// ...
printf("\nThe class counters are:\n\tShape = %d (total)\n"
    "\tCircle = %d\n\tRectangle = %d\n\tSmiley = %d\n",
    ShapeCounterTotal(),CircleCounter(),RectangleCounter(),SmileyCounter());
```

We emphasize that we never put the concrete objects together into an array. This would be impossible since all the objects are of different types. Only the interfaces are uniform and may be vectorized.

Optionally, the object may be created first and then converted to its base class for additional operations. In that case, the code would be:

```
// code from item 1. ''Creation'' above
Circle circle = CircleCreate(ShapeNew(x=20., y=30.), radius=6.);
ShapeReplace(&shapes[0], CircleShape(circle));
```

We just have to make sure that all the `Shapes` are referred to only by a single variable. Here, the variable for the `Shape` associated with `circle` is `shapes[0]`. Such a rule is discussed in Section 3.

3. Summary and Discussion

We present very short discussions of a number of topics:

1. *Summary.* We have presented a design pattern in C for inheritance in the object-oriented paradigm which may help avoid using more complicated languages like C++ and provide scalability [5].
2. *Extensibility.* Our code is extensible by more derived classes. We can add more shape types without modifying any of the existing code for `Shape` or existing derived types. Moreover, the files do not need to be recompiled, so the new code can be linked to an existing library. A straightforward naive implementation of *polymorphism* could have made use of awkward **switch** statements (or a similar type of dispatch tables) for choosing concrete methods by a derived class ID to be applied to a given base-class object. Such dispatch tables necessarily have to grow when new derived classes are added, thus slowing down the execution and possibly breaking backward compatibility. We emphasize that our implementation avoids such problems.
3. *Virtual function tables.* A design pattern involving VFT is necessary for some tasks involving variable class members, which would be impossible if all class and object members were put into the same **struct**. Therefore, we strongly recommend to use it.
4. *Typecasting.* In the Introduction (Section 1) we mentioned that we never perform true typecasting between the "base" and "derived" objects. However, some similarities with typecasting may be seen in the functions for creation of the derived object from the base and getting the base handle from the derived:

```
Circle circle = CircleCreate(shape, radius=6.);
Shape shape = CircleShape(circle);
```

5. *Comparison with composition.* One may say that we just implemented "composition" instead of "inheritance" because the base/interface handle is included in the **struct** of the derived class. A big difference, however, is that the base/interface class was written particularly with the intention of serving as such, i.e., for the developers to be able to create the derived classes from it. One feature that reveals this intention is that a base/interface object has access to the internals of the derived object through the `shape->instance` field. Thus, it is not completely decoupled from the derived class. This access is necessary because when we treat all the diverse shape objects as `Shapes`, we only have the handles of the interfaces and not of the derived objects themselves. This was shown in the example above where they were all stored in an array.
6. *Creation and destruction order.* The general policy is that the derived object cannot exist without any of its interfaces. Thus, the interfaces must be created first and then a derived object is created taking interfaces as arguments as `derived = DerivedCreate(base1,base2,...,arg)`. The destruction of an interface triggers destruction of the derived object, for which no user-accessible destructor is available. The difference between creators of interfaces (base objects) on one hand and derived objects on the other hand is obvious if we use our naming conventions: `BaseNew` (with corresponding destructor `BaseDelete` and copy-assignment operator `BaseReplace`) versus `DerivedCreate` (without a corresponding destructor).
   This idea may be taken to extreme and all objects could be inherited from a class `Object` which would provide an interface for destruction (we could as well give it a name `Destructible` but we are open to possibility that this interface could provide more functionality). The instances of this class could be stored in a list, which is separate for each scope in the program. At the beginning of a scope, the list is initialized. As the objects are created, they are added to this list. At the end of the scope, a single call to a function could destruct all the items of this list:

```
// ... we have also "outside_scope"
{ // scope beginning marker
    Scope this_scope = ScopeNew(); // initialized list of Objects in the beginning of the scope
    // ...
    Shape shape = ShapeCreate(this_scope, x, y); // an Object instance is created for "shape" and is added to the
     current scope
    // A private reference to it would be obj = shape->object, while obj->instance==shape
    Shape shape0 = ShapeCreate(outside_scope, x0, y0); // will NOT be detructed at the end of this scope
    // ...
    ScopeDelete(this_scope); // "shape" is destructed automatically, among other objects added to this_scope
} // scope end marker
// ...
// We can still use shape0 before outside_scope is destructed
```

   As we see, we can keep track of different scopes. We do not even need to access individual objects (the `Object` interface is thus completely private) unless we want to delete them in the middle of the scope. We have not implemented such a model yet.
7. *Safe programming.* The user should keep only a single copy of each interface object. It is a pointer which is set to `NULL` by `BaseDelete`. If there is a copy of it, the copy will not be set to `NULL` and the user may try to delete it again, which will lead to a segmentation fault. If the user needs to use the same interface object in several different places in his code, it is recommended to use multiple pointers to a single object instead. Since the base object is a pointer itself (like `Shape` in the presented example), these multiple instances will be pointers to a pointer.
   An example of such an error is given in the attached code. In file `Circle.h`, we declare a function **void** `CircleDelete(Circle *circlePtr)` which is supposed to take care of destructing a Circle without first accessing its base-class part Shape. Since it operates on the Shape behind the curtains, we lose track of whether it is still valid and there is a danger of trying to delete it again.
8. *Multiple inheritance.* It may be implemented by having multiple interfaces in the same derived object. The presented pattern protects us from memory leaks if all calls to constructors `BaseXNew` are matched with calls to corresponding destructors `BaseXDelete`, where `BaseX` (e.g., `Base1`, `Base2`, etc.) are names of interface classes. The destructors of the derived objects, however, must be hidden from the user, and should be automatically called by `BaseXDelete`. If there are many bases (interfaces), the destructors for each of them have to be called

explicitly. The first call destructs the derived part, while the following calls just silently skip this step since the derived part is not present anymore.

9. *Multi-stage inheritance.* The presented pattern seems to allow multi-stage inheritance. E.g., in a two-stage inheritance, the base class is a small interface, the first-level derived class (child) is an extended interface, and the second-level derived class (grandchild) is a concrete class using the extended interface. This may be of use, e.g., when the first-level class contains many small interfaces, and is thus an "interface to a collection of interfaces." One may recognize it as the Facade pattern [9]. In our implementation, since the interfaces store a pointer to a child, it is possible to get the concrete object even from the grandparent (small interface). As for memory management, only the base objects must be explicitly deleted. The `BaseDelete` of the base class triggers destruction of the child object, which in turn triggers the destruction of the grandchild object.
10. *Diamond inheritance.* A possible difficulty with multiple interfaces in our implementation is that each interface is an object so we cannot blend them together. I.e., if two interfaces have a method with the same signature, these two methods will be separate in our code because they are called by different interface objects. Such a situation would be desirable if the same-signature method is inherited from a common grandparent, i.e., we have a diamond-shaped inheritance graph [10]. Finding a good way to deal with such a situation is a problem in other programming languages, too, so we will not try to solve it here.

[1] B. Klemens, 21st Century C, 2nd Edition, O'Reilly Media, Inc., 2015.
[2] C. Adamson, Punk rock languages: A polemic (2011).
URL `http://bit.ly/punk-lang`
[3] Tiobe index.
URL `https://www.tiobe.com/tiobe-index/`
[4] A. Tornhill, Patterns in C: Patterns, Idioms and Design Principles, Leanpub, 2015.
URL `https://leanpub.com/patternsinc`
[5] P. Hintjens, Scalable C: Writing Large-Scale Distributed C, GitBook, 2016, in progress.
URL `https://legacy.gitbook.com/book/hintjens/scalable-c/details`
[6] P. Graham, Five questions about language design (2001).
URL `http://www.paulgraham.com/langdes.html`
[7] P. Norvig, Design patterns in dynamic programming (1996).
URL `http://norvig.com/design-patterns/design-patterns.pdf`
[8] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997.
[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.
[10] GeeksforGeeks. [link].
URL `https://www.geeksforgeeks.org/java-and-multiple-inheritance/`
[11] B. Stroustrup, The C++ Programming Language, 4th Edition, Pearson Education, Inc., 2013.
[12] B. H. Liskov, J. M. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (6) (1994) 1811–1841. `doi:10.1145/197320.197383`.
[13] A. T. Schreiner, Object-Oriented Programming With ANSI-C, Lulu, 2011.
URL `https://www.cs.rit.edu/~ats/books/ooc.pdf`
[14] D. Saks, Alternative idioms for inheritance in C (2013).
URL `https://www.embedded.com/electronics-blogs/programming-pointers/4411013/Alternative-idioms-for-inheritance-in-C`
[15] Quantum Leaps, LLC, Application note: Object-oriented programming in C (2018).
URL `https://www.state-machine.com/doc/AN_OOP_in_C.pdf`
[16] Wikipedia. [link].
URL `https://en.wikipedia.org/wiki/Type_punning`
[17] StackExchange, Implement inheritance in C (2016).
URL `https://softwareengineering.stackexchange.com/questions/338926/implement-inheritance-in-c`
[18] M. Gallagher, Type punning isn't funny: Using pointers to recast in C is bad (2008).
URL `https://www.cocoawithlove.com/2008/04/using-pointers-to-recast-in-c-is-bad.html`
[19] S. Meyers, Effective Modern C++, 1st Edition, O'Reilly Media, Inc., 2017, 11th release.
URL `http://oreilly.com/catalog/errata.csp?isbn=9781491903995`
[20] Wikipedia. [link].
URL `https://en.wikipedia.org/wiki/Virtual_method_table`