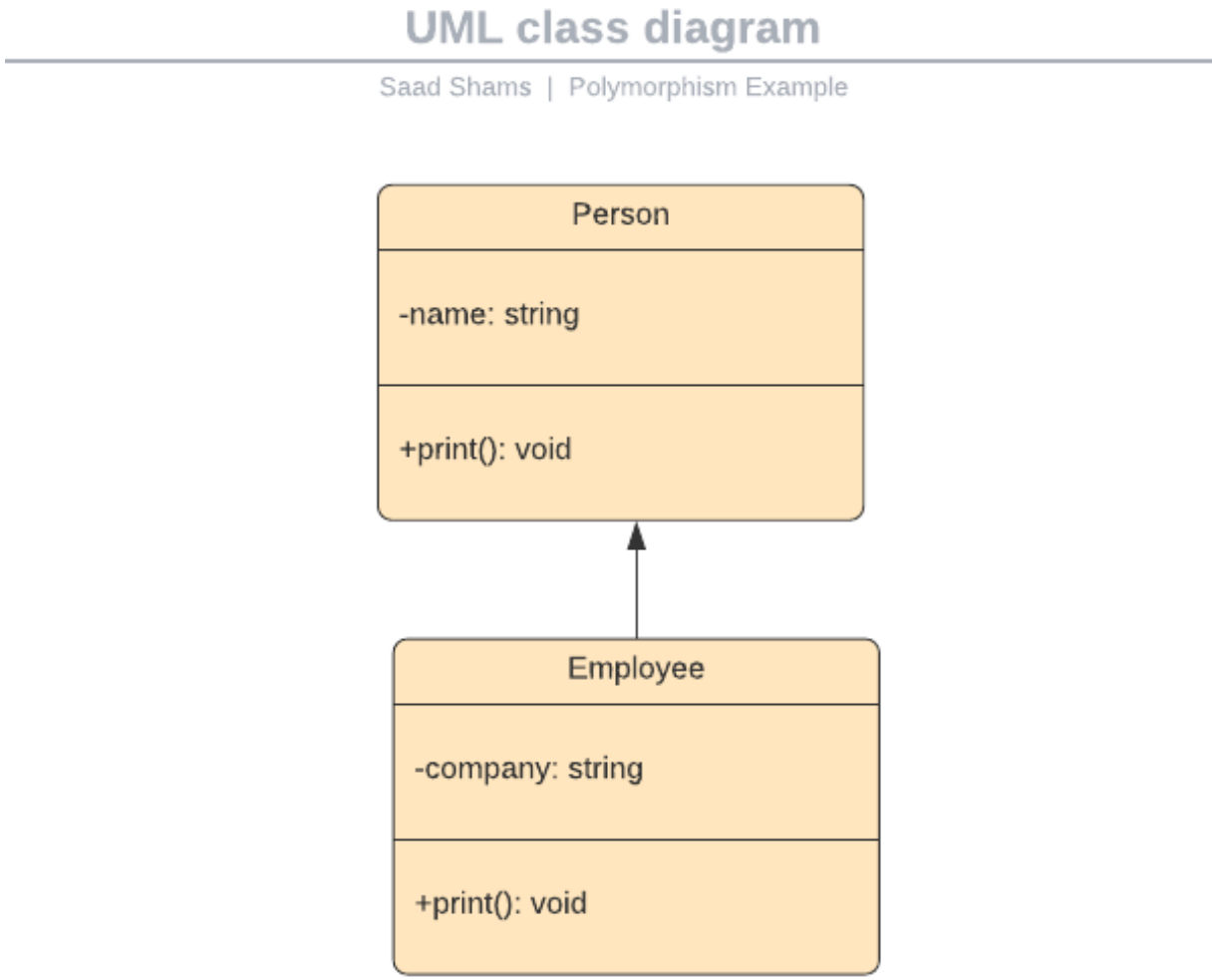
Saad Shams

SOURCE CODE: HTTPS://GITHUB.COM/SAADSHAMS/POLYMORPHISMINC

OOP IN C

Object-oriented programming is a model that organizes software design around data or objects rather than functions and logic. This document simplifies Encapsulation, Inheritance or Composition, and Polymorphism in a procedure-oriented language.

Consider a basic demo based on a UML diagram around two entities with Employee extended by Person or, technically speaking, composed by it. (See Composition Over Inheritance: https://medium.com/geekculture/composition-over-inheritance-7faed1628595)



POLYMORPHISM

To begin, observe (Fig 1.1) how the base pointer has references to both People and Employee objects (Line 7 & 12) can call print on both (Line 8 & Line 13) using the base pointer.

The Employee overrides its function (print company) while reusing Person (print name) implementation.

See console output (Fig 1.1)

- Person prints the name.
- Employee prints name and company.

```
main.c
1   #include ...
4
5   int main() {
6
7       Person *person = NewPerson( name: "John Doe");
8       person->print(person); // print name
9
10      puts("------");
11
12      person = (Person *) NewEmployee( name: "Jane Doe",  company: "Acme"); // Assign subclass instance to parent pointer
13      person->print(person); // print name and company
14
15      return 0;
16  }
```

```
Run:  TestPoly
/Users/sshams/Documents/PureMVC/Polymorphism_solved/cmake-build-debug/TestPoly
Name: John Doe
------
Name: Jane Doe
Company: Acme

Process finished with exit code 0
```

Fig 1.1

PERSON IMPLEMENTATION

```c
#ifndef _PERSON_H
#define _PERSON_H

typedef struct Person Person;

struct Person {
    char *name;
    void (*print)(const Person *self); // function pointer to print person
};

Person* InitPerson(Person *self, const char *name);

Person* NewPerson(const char *name);

#endif
```

```c
#include ...

static void print(const Person *self) {
    printf("Name: %s\n", self->name);
}

Person* InitPerson(Person *self, const char *name) {
    self->name = strdup( s1: name);
    self->print = print; // providing default implementation
    return self;
}

Person* NewPerson(const char *name) {
    return InitPerson( self: malloc( size: sizeof(Person)), name);
}
```

Fig 1.2

Encapsulation: Data and behavior are encapsulated in a struct, and the implementation file initializes both.

EMPLOYEE IMPLEMENTATION

```c
#include "person.h"

typedef struct Employee Employee;

struct Employee {
    Person person;  // composition approach (vs. inheritance)
    char *company;  // introducing additional field
};

Employee* InitEmployee(Employee *self, const char *name, const char *company);

Employee* NewEmployee(const char *name, const char *company);
```

```c
1  #include "employee.h"
2  #include <string.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  static void (*_print)(const Person *self); // to retain function if class wishes to reuse base functionality
7
8  static void print(const Employee *self) {
9      _print(&self->person); // call base/super function - print name
10     printf("Company: %s\n", self->company); // print company
11 }
12
13 Employee* InitEmployee(Employee *self, const char *name, const char *company) {
14     InitPerson( self: &self->person, name); // base initializer
15     self->company = strdup( s1: company);
16
17     _print = self->person.print; // retain base/super function before overriding
18     self->person.print = (void (*)(const Person *)) print; // override
19     return self;
20 }
21
22 Employee* NewEmployee(const char *name, const char *company) {
23     return InitEmployee( self: malloc( size: sizeof(Employee)), name, company);
24 }
```

Fig 1.3

Composition, overriding, and calling base/super implementation:

The steps below made Polymorphism possible while overriding and/or reusing base implementations:

1. Compose Person in the Employee (.h: Line 6)
2. InitEmployee initializing data while providing overridden print function (.c: Line 8, 18)
3. Before overriding, it caches the base function within a private variable (.c: Line 6, 17)
4. Call the base function via cache pointer followed by own implementation (.c Line 9, 10)

These insights were gained while implementing PureMVC Framework for GoLang (a modern language by Founders of C from AT&T Bell Labs) and paved C's PureMVC Framework.
PureMVC GoLang: https://github.com/PureMVC/puremvc-go-multicore-framework/wiki
PureMVC C: https://github.com/saadshams/puremvc-c-multicore-framework