Homework 2:
Virtual Realities

Due March 3 by the start of lecture.

Overview

In this homework, you will explore the dynamic dispatch mechanism of virtual functions in object-oriented languages. We will hand-build a vtable for a group of "classes" in the C language, and show how the compiler uses a vtable to select a derived function to call at run time based on the type of the value it is called on.

Function Pointers

C and C++ allow programmers to declare variables that point to *functions* rather than *values*. When such a pointer is dereferenced, the actual function that it points to is called, using arguments supplied by the programmer. C and C++ do type checking of function pointers to make sure the right number and types of arguments are provided at compile time. Example:

```
int max(int a, int b) { return a >= b ?  a :  b; } // return the larger integer

// in main():
int (*pFunc)(int, int);     // pFunc is a pointer to a function that takes 2 int arguments
                            // and returns int
pFunc = max;                // pFunc now points to the real function "max"
printf("%d", pFunc(5, 10)); // dereference pFunc, use its return value
```

In F#, we say that a function taking two integers and returning integer has type `int->int->int`. In C, we say it has type `int (*)(int, int)`. (As if we replaced the name of the function with `(*)`, removed the parameter names, and left everything else the same.)

Like all pointers, we do not necessarily know at compile time what a function pointer actually points to, so the actual function to be executed will not be known until run-time. We can use this with some trickery to implement function calls that execute different function bodies depending on run-time values.

Faking Objects and Polymorphism

We will implement this lab in C, which means we do not have access to objects... but we can fake it with structs. After all, a struct is a class without member functions, inheritance, or polymorphism... but it *can* store a pointer to a vtable, and through this we will implement polymorphism and dynamic dispatch.

Consider the following **C++ code** showing the polymorphism feature we want to emulate:

```
class Employee {
   int age;
public:
   int GetAge() { return age; }
   virtual void Speak() = 0;
   virtual double GetPay() = 0;
};

class HourlyEmployee :  public Employee {
   double hourly_rate;
   double hours;
```

```
public:
   virtual void Speak() { cout << "I work for " << hourly_rate << " dollars per hour :(";
};


class CommissionEmployee :  public Employee {
   double sales_amount;
public:
   virtual void Speak() { cout << "I make commision on " << sales_amount << " dollars in sales!";
};

// in main()
Employee *e = ...  // suppose e points to either a CommissionEmployee or HourlyEmployee.
cout << e->GetPay(); // which function gets called?  Who knows!
```

We can emulate this in C using:

1. A `struct Employee`, with two member fields: a **pointer** to vtable (as a `void**`); and an integer field `age`.
2. A `struct HourlyEmployee`, with four member fields: the same fields as `Employee`, and doubles `hourly_rate` and `hours`.
3. A `struct CommissionEmployee`, likewise, but with a double `sales_amount`.
4. Global functions to emulate each of the member functions of the objects:
   (a) `Speak_Hourly` which takes an `Employee` pointer, casts it to a `HourlyEmployee` pointer, and prints the employee's message;
   (b) `GetPay_Hourly` which also takes an `Employee` pointer and returns the employee's total pay (see below);
   (c) `Construct_Hourly` which takes a `HourlyEmployee` pointer and initializes its fields to their default values, **most importantly** initializing the objects's vtable pointer (more on this below).

and then following some tedious steps to create variables of our types and call the appropriate functions:

1. To make a `HourlyEmployee`, declare a `HourlyEmployee` variable (either on the stack or with `malloc`) and then pass it by pointer to `Construct_Hourly`.
2. To use "subtype polymorphism" to point an `Employee` at a `HourlyEmployee`, declare an `Employee` pointer and initialize it by casting your `HourlyEmployee`'s address to an `Employee` pointer.
3. To use "dynamic dispatch", dereference the `Employee` pointer's vtable pointer, index the table to the appropriate method, cast that pointer to the correct function pointer type, and invoke the method by passing the `Employee` pointer and any other necessary parameters. Easy!

Vtables

A **vtable** (short for **virtual table**) is a table of function pointers, with one entry in the table for each virtual function in a class (or its ancestors). Since the `Employee` class has two virtual functions, the vtable for any `Employee`-derived object will have two pointers in it for `Speak` and `GetCost`, plus additional entries for any more virtual functions introduced by the derived class. C++ handles the creation and use of vtables automatically; in this lab, we will simulate the work that a C++ compiler performs to transform virtual method calls into vtable lookups.

Suppose we have two functions: `void Speak_Hourly(struct HourlyEmployee*)` and `double GetPay_Hourly(struct HourlyEmployee*)`. We can build a "table" (really an array) of two pointers to functions as such:

```
void* Vtable_Hourly[2] = {Speak_Hourly, GetPay_Hourly};
```

The `void*` type in C lets us create a pointer to anything, but the compiler won't help us use such a pointer; we will need to cast it to something specific in order to use it. Suppose we have a `struct HourlyEmployee h` variable that we want to call `Speak_Hourly` on, but we don't want to use `Speak_Hourly` directly – instead, we want to use `Vtable_Hourly`. We note that the first entry in `Vtable_Hourly` is a pointer to `Speak_Hourly`, and thus try to use that pointer to call the function it points to:

```
Vtable_Hourly[0]((struct HourlyEmployee *)&h);
```

This makes sense in our head: the element 0 of `Vtable_Hourly`is a pointer to `Speak_Hourly`, which wants a single parameter of type `struct Employee *`. Unfortunately C sees `Vtable_Hourly[0]` as a pointer to `void`, not to a function; we have to tell the compiler that it actually points to a function that returns `void` and takes a single parameter of type `struct Employee *`. A cast will accomplish this task:

```
((void (*)(struct Employee*))Vtable_Hourly[0])((struct Employee *)&h);
```
[1]
which tells the compiler to invoke the function pointed to by `Vtable_Hourly[0]`, passing it the address of `h` as its parameter. Success!

All we need to do now is add a new member variable to our structs: a `void**` pointer to a vtable appropriate to the class. Create one vtable variable globally for all derived `Employee` types, point the vtable pointers to the appropriate global tables in constructor methods, and voila! We can now call a virtual function through a base class pointer by accessing the appropriate index from the vtable associated with the variable and invoking that function by hand.

Program

Implement the following code:

1. Create `Employee`, `HourlyEmployee`, and `CommissionEmployee` structs as described above. The first member of each struct should be a `void** vtable` variable.
2. Implement the `Speak_Hourly` and `Speak_Commission` functions for the `HourlyEmployee` and `CommissionEmployee` structs as global functions.
3. Add `HourlyEmployee` and `CommissionEmployee` implementations of a `GetPay` function, which takes an `Employee` pointer and returns a `double` as such:
   (a) The pay for an hourly employee is the number of hours multiplied by their hourly rate.
   (b) The pay for a commission employee is 10% of their total sales, plus 40,000.
4. Create a `Vtable_XX` global array for each employee derived type, initialized with pointers to the appropriate `Speak` and `GetPay` functions for that type, **in that order**.
5. Implement `Construct_XX` functions for `HouryEmployee` and `CommissionEmployee`, which initialize the member variables to 0 values, and **most importantly**, sets the employee's vtable pointer to the appropriate global `Vtable_XX` variable. (Don't overthink this. It's as easy as writing `parameter->vtable = Vtable_Hourly;`, for example.)
6. Add a new "class" `SeniorSalesman`, which "derives" from `CommissionEmployee` by duplicating all of `CommissionEmployee`'s member variables. `SeniorSalesman` will override the `GetPay` method but will use `CommissionEmployee`'s version of `Speak`. Create a vtable and constructor for the `SeniorSalesman` class, and a `GetPay` method that returns 20% of the salesman's sales, plus 50,000, plus another 5% of sales if the employee is at least 40 years old.

Then write a main program that does the following:

1. Declare an `Employee` pointer.
2. Ask the user to choose either an hourly employee, a commission employee, or a senior salesman.
3. Use `malloc` to create space for the appropriate employee, for example, `HourlyEmployee *h = (HourlyEmployee*)mallo`

[1] We know that we're working with function pointers in C when our code starts to look like line noise.

(a) Ask the user how old the employee is.

(b) If the user selects an hourly, ask them for the employee's pay rate and hours.

(c) If the user selects a commission or a senior salesman, ask for the employee's amount of sales.

(d) Use your `Construct_` function to initialize the memory given back from `malloc`, passing the appropriate parameters for the type selected.

(e) Point your `Employee` to the variable.

(f) **Past this point of the program, you can have no code referring to CommissionEmployee, HourlyEmployee, or SeniorSalesman explicitly – everything must be through Employee pointers.**

4. Tell the `Employee` to speak, then inform the user how much money they make.

   (a) To do this, you will access the `vtable` pointer from the `Employee` pointer in your main, index it to the appropriate position for each function, cast that pointer as described above, and invoke the function it is pointing to.

Deliverables

Turn in the following when the lab is due:

Turn in the following when the lab is due:

1. Your source code file(s).
2. A printed copy of the output of your program, where you choose:

   (a) an hourly employee 25 years old making $9.50 an hour working 90 hours.

   (b) a commission employee 30 years old with $80,000 in sales.

   (c) a senior salesman 50 years old with $100,000 in sales.