Computer Architecture

Programming in C

OOP-like programming in C

Agenda

- Function pointers
- Simulating a class
- Simulating inheritance and polymorphism

Function pointer

```c
#include <stdio.h>

// Define a simple function
void greet() {
    printf("Hello, World!\n");
}

int main() {
    // Declare a function pointer that points to
    //a function returning void and taking no arguments
    void (*functionPtr)();

    // Assign the address of the function 'greet'
    //to the function pointer
    functionPtr = &greet;

    // Call the function via the function pointer
    // Option 1: Using dereferencing
    (*functionPtr)();
    // Option 2: Direct invocation (same as calling the function normally)
    functionPtr();

    return 0;
}
```

A function pointer is a pointer that points to the address of a function.

Define a function pointer:

`return_type (*pointer name)(parameters)`

A function pointer allows one to

- call the function indirectly
- pass the function as a parameter to another function
- store functions in array or other data structures

Simulating a class

- The `Rectangle` structure contains two data member (`width` and `height`) and two function pointers (`area` and `perimeter`).
- Functions `calculateArea` and `calculatePerimeter` act as **methods** that operate on the `Rectangle` instance.
- The function pointers inside the structure allow us to call these functions like calling methods in OOP.
- The `createRectangle` function acts like a **constructor**, initializing a `Rectangle` object with given width and height.

```c
// Define a structure to represent the "class"
struct Rectangle {
    int width;
    int height;
    // Function pointers for "methods"
    int (*area)(struct Rectangle* r);
    int (*perimeter)(struct Rectangle* r);
};

// Method to calculate area (equivalent to a class method)
int calculateArea(struct Rectangle* r) {
    return r->width * r->height;
}

// Method to calculate perimeter (equivalent to a class method)
int calculatePerimeter(struct Rectangle* r) {
    return 2 * (r->width + r->height);
}

// Constructor function to initialize the "object"
struct Rectangle createRectangle(int width, int height) {
    struct Rectangle r;
    r.width = width;
    r.height = height;

    // Assign functions (methods) to function pointers
    r.area = calculateArea;
    r.perimeter = calculatePerimeter;
    return r;
}
```

Simulating a class

```c
int main() {
    // Create a Rectangle object (instantiate)
    struct Rectangle rect = createRectangle(10, 20);
    // Access attributes and call methods (like OOP)
    printf("Width: %d, Height: %d\n", rect.width, rect.height);
    printf("Area: %d\n", rect.area(&rect));          // Call the area method
    printf("Perimeter: %d\n", rect.perimeter(&rect)); // Call the perimeter method
    return 0;
}
```

Simulating Inheritance

```c
// Define the Animal structure (base class)
struct Animal {
    char name[50];
    // Function pointer for the sound method
    void (*makeSound)(struct Animal* a);
};

// Define the Dog structure (derived class, inherits from Animal)
struct Dog {
    struct Animal base;  // The Dog "inherits" the Animal struct
};

// Animal's makeSound method
void animalSound(struct Animal* a) {
    printf("%s makes a generic animal sound.\n", a->name);
}

// Dog's makeSound method (overrides Animal's method)
void dogSound(struct Animal* a) {
    printf("%s barks: Woof!\n", a->name);
}
```

Inheritance: The Dog structure contains the Animal structure as a member, effectively allowing it to inherit the attributes and methods of Animal.

Simulating Inheritance

```
26  // Constructor for Animal
27  struct Animal createAnimal(const char* name) {
28      struct Animal a;
29      //strncpy(dest, src, n);
30      //copy the n characters from src to dest
31      strncpy(a.name, name, sizeof(a.name) - 1);
32      a.makeSound = animalSound;
33      return a;
34  }
35  // Constructor for Dog
36  struct Dog createDog(const char* name) {
37      struct Dog d;
38      d.base = createAnimal(name);  // Initialize the Animal part
39      d.base.makeSound = dogSound;  // Override the makeSound function
40      return d;
41  }
42
43  int main() {
44      // Create an Animal object
45      struct Animal a = createAnimal("GenericAnimal");
46      a.makeSound(&a);  // GenericAnimal makes a generic animal sound.
47      // Create a Dog object
48      struct Dog d = createDog("Buddy");
49      d.base.makeSound(&d.base);  // Buddy barks: Woof!
50      return 0;
51  }
```

- `strncpy(dest, src, n)` is a function from `<string.h>`
  - It copies n characters in src to dest.
- In `createDog`, it initializes the `Dog` by calling the `createAnimal`. Then, overwriting the `makeSound`, which is essentially a function pointer.

Simulating Polymorphism

```
// Derived class: Dog
struct Dog {
    struct Animal base;  // Inherit from Animal
};
// Derived class: Cat
struct Cat {
    struct Animal base;  // Inherit from Animal
};
// Dog-specific sound
void dogSound(struct Animal* a) {
    printf("Dog '%s' says: Woof!\n", a->name);
}
// Cat-specific sound
void catSound(struct Animal* a) {
    printf("Cat '%s' says: Meow!\n", a->name);
}
// Constructor for Dog
struct Dog createDog(const char* name) {
    struct Dog d;
    strncpy(d.base.name, name, sizeof(d.base.name) - 1);
    d.base.makeSound = dogSound;  // Assign dog-specific sound function
    return d;
}
// Constructor for Cat
struct Cat createCat(const char* name) {
    struct Cat c;
    strncpy(c.base.name, name, sizeof(c.base.name) - 1);
    c.base.makeSound = catSound;  // Assign cat-specific sound function
    return c;
}
```

- `Dog` and `Cat` inherit from `Animal`.
- In the constructors, the `makeSound` in each "class" overwrites by `dogSound` and `catSound`, respectively. So, when the `makeSound` of different objects is called, it will behave differently.

```
int main() {
    // Create instances of Dog and Cat
    struct Dog dog1 = createDog("Buddy");
    struct Cat cat1 = createCat("Whiskers");
    struct Dog dog2 = createDog("Rex");
    // Array of Animal pointers to
    //simulate polymorphism
    struct Animal* animals[3];
    animals[0] = (struct Animal*)&dog1;
    animals[1] = (struct Animal*)&cat1;
    animals[2] = (struct Animal*)&dog2;
    // Iterate over the array and
    //call makeSound polymorphically
    for (int i = 0; i < 3; ++i) {
        animals[i]->makeSound(animals[i]);
    }
    return 0;
}
```

Exercise

Please try the exercise in Recitation Week 3 on Gradescope