Objektorientert programmering i C og Rust

Introduksjon

Vi skal se på hvordan vi programmerer objektorientert programmering i C og Rust. Vi dekker her klasser, datamedlemmer og medlemsfunksjoner. Polymorfi vil bli vist senere. Se først på eksempelet programmert i C++:

```cpp
#include <iostream>

using namespace std;

class Surface {
public:
  Surface(double length_, double width_) : length(length_), width(width_) {}

  double length;
  double width;

  double area() {
    return length * width;
  }
};

class ColoredSurface : public Surface {
public:
  ColoredSurface(double length, double width, double red_, double green_, double blue_)
      : Surface(length, width), red(red_), green(green_), blue(blue_) {}

  double red;
  double green;
  double blue;
};

int main() {
  Surface surface(2.0, 3.0);
  cout << surface.width << endl;  // Outputs 3
  cout << surface.area() << endl; // Outputs 6

  ColoredSurface colored_surface(2.0, 3.0, 0.0, 0.0, 1.0);
  cout << colored_surface.width << endl;  // Outputs 3
  cout << colored_surface.blue << endl;   // Outputs 1
  cout << colored_surface.area() << endl; // Outputs 6
}
```

Her utvides `Surface` -klassen i `ColoredSurface` , og instanser av `ColoredSurface` kan brukes som instanser av `Surface` -klassen.

C

For å opprette instanser med datamedlemmer må vi i C bruke nøkkelordet `struct` i stedet for `class` . Alle datamedlemmene er synlige som ved `public` i C++.

Medlemsfunksjonene må i tillegg legges utenfor `struct` 'ene. Vi må derfor ha argumenter i disse medlemsfunksjonene slik at vi kan legge til instansen som funksjonen skal kjøres mot (se `surface_area()` -funksjonen med argumentet `Surface *surface`).

I tillegg, om vi skal utvide en datastruktur, må vi opprette en instans av basestrukturen i den utvidede strukturen. Dette fører til at bruken av en instanse av en utvidet struktur blir litt annerledes enn bruken av en instanse av basestrukturen. For eksempel må en skrive `colored_surface.surface.width` i stedet for `colored_surface.width` .

```c
#include <stdio.h>

typedef struct {
  double length;
  double width;
} Surface;

double surface_area(Surface *surface) {
  return surface->length * surface->width;
}

typedef struct {
  Surface surface; // Extend Surface struct by adding a Surface object
  double red;
  double green;
  double blue;
} ColoredSurface;

int main() {
  Surface surface = {2.0, 3.0};           // Create surface instance with length = 2.0 and width = 3.0
  printf("%f\n", surface.width);          // Outputs 3
  printf("%f\n", surface_area(&surface)); // Outputs 6

  ColoredSurface colored_surface = {{2.0, 3.0}, 0.0, 0.0, 1.0}; // Create colored surface instance with
                                                                // length = 2, width = 3, red = 0, green = 0, blue
  printf("%f\n", colored_surface.surface.width);                // Outputs 3
  printf("%f\n", colored_surface.blue);                         // Outputs 1
  printf("%f\n", surface_area((Surface *)&colored_surface));    // Outputs 6. Colored surface is downcast to Surfac
}
```

Merk at vi her kan også bruke `surface_area()` -funksjonen med et `ColoredSurface` objekt. Dette gjøres ved å gjøre om en peker til en `ColoredSurface` -instans til en peker av typen `Surface*` . Dette fungerer fordi `Surface surface` er plassert øverst i `ColoredSurface` -strukturen, og `length` og `width` finnes da på de samme minneområdene som de ville gjort om de var plassert i en `Surface` instanse.

Rust

For å oppnå det samme med Rust må en implementere `area()` medlemsfunksjonene for både `Surface` og `ColoredSurface` datastrukturene. Merk at `struct` i Rust fungerer langt på vei som `struct` nøkkelordet i C.

```rust
struct Surface {
    width: f64, // f64 corresponds to double in Rust
    length: f64,
}

impl Surface {
    fn area(self: &Self) -> f64 { // self is a reference to the calling Surface instance
        return self.width * self.length;
    }
}

struct ColoredSurface {
    surface: Surface, // Extend Surface struct by adding a Surface object
    red: f64,
    green: f64,
    blue: f64,
}

impl ColoredSurface {
    // Need to implement area() for ColoredSurface as well
    fn area(self: &Self) -> f64 {
        return self.surface.area();
    }
}

fn main() {
    let surface = Surface {
        width: 2.0,
        length: 3.0,
    };
    println!("{}", surface.width); // Outputs 2
    println!("{}", surface.area()); // Outputs 6

    let colored_surface = ColoredSurface {
        surface: Surface {
            width: 2.0,
            length: 3.0,
        },
        red: 0.0,
        green: 0.0,
        blue: 1.0,
    };
    println!("{}", colored_surface.surface.width); // Outputs 2
    println!("{}", colored_surface.blue); // Outputs 1
    println!("{}", colored_surface.area()); // Outputs 6
}
```

Siden en må implementere samme funksjonen flere ganger kan det være nyttig å lage en *interface* (*trait*) for denne, og eventuelt andre lignende funksjoner, slik at vi slipper tilfeller der en funksjon for eksempel heter `area` en plass og `area1` en annen plass:

```rust
struct Surface {
    width: f64, // f64 corresponds to double in Rust
    length: f64,
}

trait GeometricQuantities {
    fn area(self: &Self) -> f64; // self is a reference to the calling instance
}

impl GeometricQuantities for Surface {
    // All functions in the trait GeometricQuantities must be implemented here,
    // and only functions in GeometricQuantities can be implemented.
    fn area(self: &Self) -> f64 {
        return self.width * self.length;
    }
}

struct ColoredSurface {
    surface: Surface, // Extend Surface struct by adding a Surface object
    red: f64,
    green: f64,
    blue: f64,
}

impl GeometricQuantities for ColoredSurface {
    // All functions in the trait GeometricQuantities must be implemented here,
    // and only functions in GeometricQuantities can be implemented.
    fn area(self: &Self) -> f64 {
        return self.surface.area();
    }
}

fn main() {
    let surface = Surface {
        width: 2.0,
        length: 3.0,
    };
    println!("{}", surface.width); // Outputs 2
    println!("{}", surface.area()); // Outputs 6

    let colored_surface = ColoredSurface {
        surface: Surface {
            width: 2.0,
            length: 3.0,
        },
        red: 0.0,
        green: 0.0,
        blue: 1.0,
    };
    println!("{}", colored_surface.surface.width); // Outputs 2
    println!("{}", colored_surface.blue); // Outputs 1
    println!("{}", colored_surface.area()); // Outputs 6
}
```

I *trait*'en `GeometricQuantities` kan vi legge til flere funksjoner, for eksempel, `circumference()` som må implementeres når en velger å implementere denne *trait*'en for en datastruktur.