Pure virtual functions and abstract classes

```
/*

    . Topics:
        . Pure virtual functions
        . Abstract classes
*/
```

Pure virtual functions

```
Shape
  ├──> virtual ~Shape() = default;                    // Virtual destructor
  ├──> virtual double perimeter() const = 0;         // Pure virtual function
  ├──> virtual double surface() const = 0;           // Pure virtual function

Rectangle : public Shape
  ├──> virtual ~Rectangle() override = default;       // Virtual destructor
  ├──> double perimeter() const override;             // Override perimeter
  ├──> double surface() const override;               // Override surface

Circle : public Shape
  ├──> virtual ~Circle() override = default;          // Virtual destructor
  ├──> double perimeter() const override;             // Override perimeter
  ├──> double surface() const override;               // Override surface
```

Pure virtual functions

```
/*
      . Facts about pure virtual functions
        . Pure virtual functions are functions that are declared in a base class but have no implementation.
            . The syntax is to add an =0 at the end of the function declaration.
        . The abstract class defines an interface that derived classes must implement.
        . If a class has at least one pure virtual function, it becomes an abstract class
        . You can’t create objects of an abstract class, if you do that ,
            you’ll get  a hard compiler error
        . Derived classes from an abstract class must explicitly override all the
            pure virtual functions from the abstract parent class, if they don’t
            they themselves become abstract
        . Pure virtual functions don’t have an implementation in the abstract class.
            They are meant to be implemented by deriving classes
        . You can’t call the pure virtual functions from the constructor
            of the abstract class
        . The constructor of the abstract class is used by deriving class to
            build up the base part of the object
*/
```

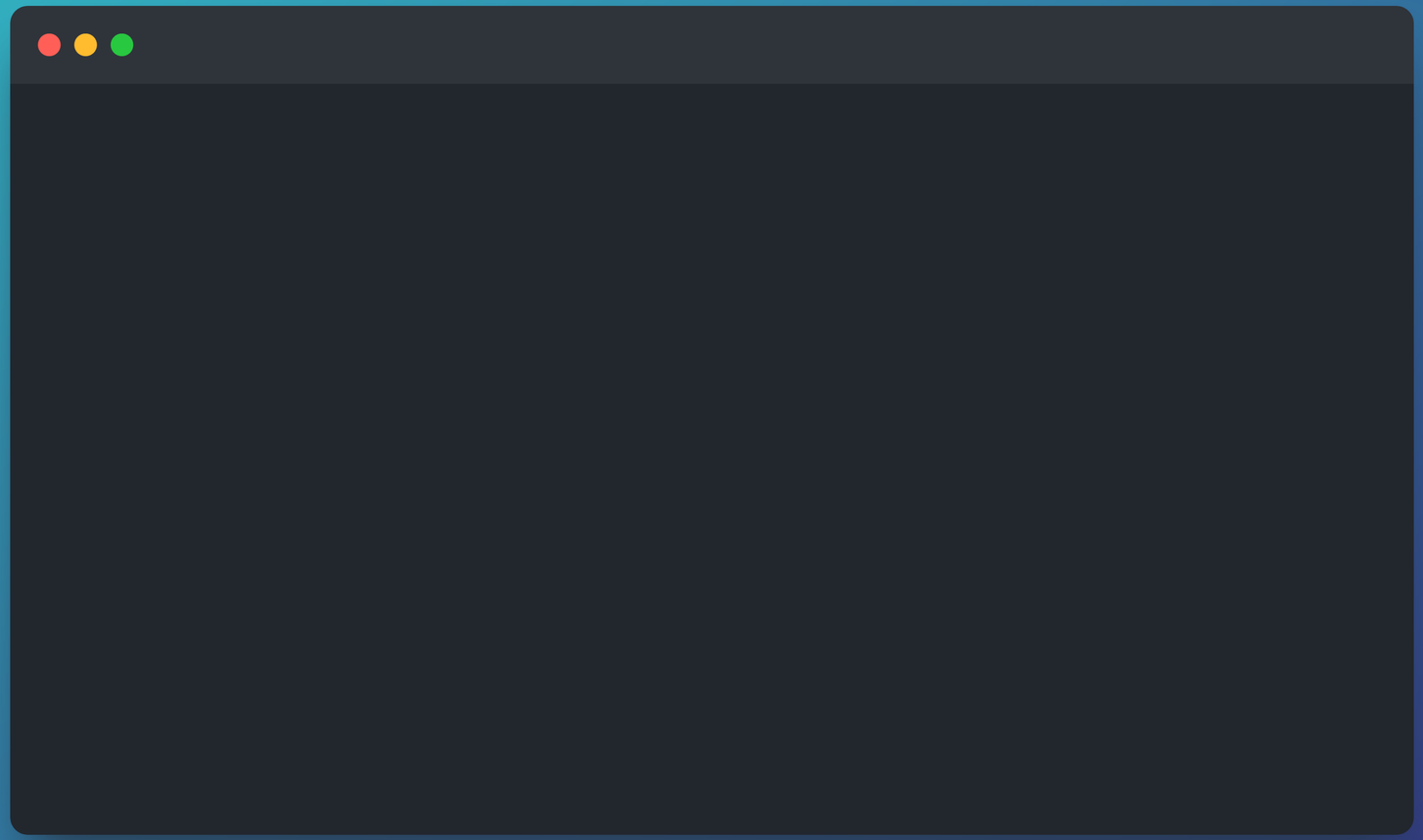
Pure virtual functions

```
//Try it out
//Shape * shape_ptr = new Shape; // Compiler error

const Shape *shape_rect = new Rectangle(10, 10, "rect1");
double surface = shape_rect→surface();
fmt::println("dynamic type of shape_rect : {}", typeid(*shape_rect).name());
fmt::println("The surface of shape rect is : {}", surface);

const Shape *shape_circle =
        new Circle(10, "circle1");
surface = shape_circle→surface();
fmt::println("dynamic type of shape_circle : {}", typeid(*shape_circle).name());
fmt::println("The surface of the circle is : {}", surface);
```

Demo time!



Abstract class Interfaces

```
/*
        . An abstract class with only pure virtual functions and no member variable can be used
            to model what is called an interface in Object Oriented Programming.

        . An interface is a specification of something that will be fully implemented in a derived
            class, but the specification itself resides in the abstract class
*/
```

Abstract class Interfaces

```cpp
// StreamInsertable interface
class StreamInsertable {
    friend std::ostream& operator<<(std::ostream& out, const StreamInsertable& operand) {
        operand.stream_insert(out);
        return out;
    }

public:
    virtual void stream_insert(std::ostream& out) const = 0; // Pure virtual function
};

// Animal class
export class Animal : public StreamInsertable {
public:
    Animal() = default;
    Animal(const std::string& description) : m_description(description) {}
    virtual void breathe() const {
        std::cout << "Animal::breathe called for: " << m_description << std::endl;
    }

    // Stream insertable interface
    virtual void stream_insert(std::ostream& out) const override {
        out << "Animal [description: " << m_description << "]";
    }

protected:
    std::string m_description;
};
```

Abstract class Interfaces

```
//Each class directly or indirectly inheriting from an abstract class has to
//implement all the pure virtal functions upstream. Otherwise, it becomes abstract.
StreamInsertable
 ├──> virtual void stream_insert(std::ostream& out) const = 0; // Pure virtual function

Animal : public StreamInsertable
 ├──> virtual void breathe() const;                           // Non-virtual method

Feline : public Animal
 ├──> virtual void run() const;                               // Non-virtual method

Dog : public Feline
 ├──> virtual void bark() const;                              // Non-virtual method

Cat : public Feline
 ├──> virtual void miaw() const;                              // Non-virtual method

Bird : public Animal
 ├──> virtual void fly() const;                               // Non-virtual method

Crow : public Bird
 ├──> virtual void cow() const;                               // Non-virtual method

Pigeon : public Bird
 ├──> virtual void coo() const;                               // Non-virtual method

Point : public StreamInsertable
```

Abstract class Interfaces

```
//Try it out
  Point p1(10,20);
  std::cout << "p1 : " << p1 << "\n";
  //operator<<(std::cout,p1);

  //std::cout << "------------" << std::endl;
  std::unique_ptr<Animal> animal0 =
       std::make_unique<Dog>("stripes","dog1");
  std::cout << *animal0 << std::endl;

  std::unique_ptr<Animal> animal1 =
       std::make_unique<Bird>("white","bird1");
  std::cout << *animal1 << std::endl;


  //Can even put animals in an array and print them polymorphically.
  std::shared_ptr<Animal> animals[] {
    std::make_shared<Dog>("stripes","dog2"),
    std::make_shared<Cat>("black stripes","cat2"),
    std::make_shared<Crow>("black wings","crow2"),
    std::make_shared<Pigeon>("white wings","pigeon2")
  };
  std::cout << std::endl;
  std::cout << "Printing out animals array : " << std::endl;
  for(const auto& a : animals){
    std::cout << *a << std::endl;
  }
```

Demo time!

```
//Try it out
  Point p1(10,20);
  std::cout << "p1 : " << p1 << "\n";
  //operator<<(std::cout,p1);

  //std::cout << "------------" << std::endl;
  std::unique_ptr<Animal> animal0 =
       std::make_unique<Dog>("stripes","dog1");
  std::cout << *animal0 << std::endl;

  std::unique_ptr<Animal> animal1 =
       std::make_unique<Bird>("white","bird1");
  std::cout << *animal1 << std::endl;


  //Can even put animals in an array and print them polymorphically.
  std::shared_ptr<Animal> animals[] {
    std::make_shared<Dog>("stripes","dog2"),
    std::make_shared<Cat>("black stripes","cat2"),
    std::make_shared<Crow>("black wings","crow2"),
    std::make_shared<Pigeon>("white wings","pigeon2")
  };
  std::cout << std::endl;
  std::cout << "Printing out animals array : " << std::endl;
  for(const auto& a : animals){
    std::cout << *a << std::endl;
  }
```