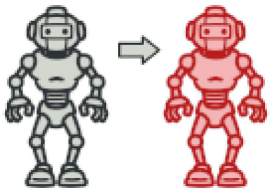

Prototype in C++

Prototype is a creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes.

All prototype classes should have a common interface that makes it possible to copy objects even if their concrete classes are unknown. Prototype objects can produce full copies since objects of the same class can access each other's private fields.

Learn more about Prototype

Complexity:

Popularity:

Identification: The prototype can be easily recognized by a `clone` or `copy` methods, etc.

Conceptual Example

This example illustrates the structure of the **Prototype** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```
using std::string;

// Prototype Design Pattern
//
```

```
// Intent: Lets you copy existing objects without making your code dependent on
// their classes.

enum Type {
  PROTOTYPE_1 = 0,
  PROTOTYPE_2
};

/**
 * The example class that has cloning ability. We'll see how the values of field
 * with different types will be cloned.
 */

class Prototype {
 protected:
  string prototype_name_;
  float prototype_field_;

 public:
  Prototype() {}
  Prototype(string prototype_name)
      : prototype_name_(prototype_name) {
  }
  virtual ~Prototype() {}
  virtual Prototype *Clone() const = 0;
  virtual void Method(float prototype_field) {
    this->prototype_field_ = prototype_field;
    std::cout << "Call Method from " << prototype_name_ << " with field : " << prototype_fiel
  }
};

/**
 * ConcretePrototype1 is a Sub-Class of Prototype and implement the Clone Method
 * In this example all data members of Prototype Class are in the Stack. If you
 * have pointers in your properties for ex: String* name_ ,you will need to
 * implement the Copy-Constructor to make sure you have a deep copy from the
 * clone method
 */

class ConcretePrototype1 : public Prototype {
 private:
  float concrete_prototype_field1_;

 public:
  ConcretePrototype1(string prototype_name, float concrete_prototype_field)
      : Prototype(prototype_name), concrete_prototype_field1_(concrete_prototype_field) {
  }

  /**
   * Notice that Clone method return a Pointer to a new ConcretePrototype1
   * replica. so, the client (who call the clone method) has the responsability
   * to free that memory. If you have smart pointer knowledge you may prefer to
```

```
   * use unique_pointer here.
   */
  Prototype *Clone() const override {
    return new ConcretePrototype1(*this);
  }
};

class ConcretePrototype2 : public Prototype {
 private:
  float concrete_prototype_field2_;

 public:
  ConcretePrototype2(string prototype_name, float concrete_prototype_field)
      : Prototype(prototype_name), concrete_prototype_field2_(concrete_prototype_field) {
  }
  Prototype *Clone() const override {
    return new ConcretePrototype2(*this);
  }
};

/**
 * In PrototypeFactory you have two concrete prototypes, one for each concrete
 * prototype class, so each time you want to create a bullet , you can use the
 * existing ones and clone those.
 */

class PrototypeFactory {
 private:
  std::unordered_map<Type, Prototype *, std::hash<int>> prototypes_;

 public:
  PrototypeFactory() {
    prototypes_[Type::PROTOTYPE_1] = new ConcretePrototype1("PROTOTYPE_1 ", 50.f);
    prototypes_[Type::PROTOTYPE_2] = new ConcretePrototype2("PROTOTYPE_2 ", 60.f);
  }

  /**
   * Be carefull of free all memory allocated. Again, if you have smart pointers
   * knowelege will be better to use it here.
   */

  ~PrototypeFactory() {
    delete prototypes_[Type::PROTOTYPE_1];
    delete prototypes_[Type::PROTOTYPE_2];
  }

  /**
   * Notice here that you just need to specify the type of the prototype you
   * want and the method will create from the object with this type.
   */
  Prototype *CreatePrototype(Type type) {
    return prototypes_[type]->Clone();
```

```
  }
};

void Client(PrototypeFactory &prototype_factory) {
  std::cout << "Let's create a Prototype 1\n";

  Prototype *prototype = prototype_factory.CreatePrototype(Type::PROTOTYPE_1);
  prototype->Method(90);
  delete prototype;

  std::cout << "\n";

  std::cout << "Let's create a Prototype 2 \n";

  prototype = prototype_factory.CreatePrototype(Type::PROTOTYPE_2);
  prototype->Method(10);

  delete prototype;
}

int main() {
  PrototypeFactory *prototype_factory = new PrototypeFactory();
  Client(*prototype_factory);
  delete prototype_factory;

  return 0;
}
```

Output.txt: Execution result

```
Let's create a Prototype 1
Call Method from PROTOTYPE_1  with field : 90

Let's create a Prototype 2
Call Method from PROTOTYPE_2  with field : 10
```