



[Home](#) / [Design Patterns](#) / [Creational patterns](#) / [Prototype](#)

Prototype in C++: Before and after



[Back to **Prototype** description](#)

Before

The architect has done an admirable job of decoupling the client from Stoooge concrete derived classes, and, exercising polymorphism. But there remains coupling where instances are actually created.

```
class Stooge
{
    public:
        virtual void slap_stick() = 0;
};

class Larry: public Stooge
{
    public:
        void slap_stick()
        {
            cout << "Larry: poke eyes\n";
        }
};

class Moe: public Stooge
{
    public:
        void slap_stick()
        {
            cout << "Moe: slap head\n";
        }
};

class Curly: public Stooge
{
    public:
        void slap_stick()
        {
            cout << "Curly: suffer abuse\n";
        }
};

int main()
{
    vector roles;
    int choice;

    while (true)
    {
        cout << "Larry(1) Moe(2) Curly(3) Go(0): ";
        cin >> choice;
        if (choice == 0)
            break;
        else if (choice == 1)
            roles.push_back(new Larry);
        else if (choice == 2)
            roles.push_back(new Moe);
        else
            roles.push_back(new Curly);
    }
}
```

```
}  
for (int i = 0; i < roles.size(); i++)  
    roles[i]->slap_stick();  
for (int i = 0; i < roles.size(); i++)  
    delete roles[i];  
}
```

Output

```
Larry(1) Moe(2) Curly(3) Go(0): 2  
Larry(1) Moe(2) Curly(3) Go(0): 1  
Larry(1) Moe(2) Curly(3) Go(0): 3  
Larry(1) Moe(2) Curly(3) Go(0): 0  
Moe: slap head  
Larry: poke eyes  
Curly: suffer abuse
```

After

A `clone()` method has been added to the Stooge hierarchy. Each derived class implements that method by returning an instance of itself. A Factory class has been introduced that maintains a suite of "breeder" objects (aka proto-types), and knows how to delegate to the correct prototype.

```
class Stooge {
public:
    virtual Stooge* clone() = 0;
    virtual void slap_stick() = 0;
};

class Factory {
public:
    static Stooge* make_stooge( int choice );
private:
    static Stooge* s_prototypes[4];
};

int main() {
    vector roles;
    int choice;

    while (true) {
        cout << "Larry(1) Moe(2) Curly(3) Go(0): ";
        cin >> choice;
        if (choice == 0)
            break;
        roles.push_back(
            Factory::make_stooge( choice ) );
    }

    for (int i=0; i < roles.size(); ++i)
        roles[i]->slap_stick();
    for (int i=0; i < roles.size(); ++i)
        delete roles[i];
}

class Larry : public Stooge {
public:
    Stooge* clone() { return new Larry; }
    void slap_stick() {
        cout << "Larry: poke eyes\n"; }
};

class Moe : public Stooge {
public:
    Stooge* clone() { return new Moe; }
    void slap_stick() {
        cout << "Moe: slap head\n"; }
};

class Curly : public Stooge {
public:
    Stooge* clone() { return new Curly; }
    void slap_stick() {
```

```
        cout << "Curly: suffer abuse\n"; }
};

Stooge* Factory::s_prototypes[] = {
    0, new Larry, new Moe, new Curly
};
Stooge* Factory::make_stooge( int choice ) {
    return s_prototypes[choice]->clone();
}
```

Read next

This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.



♥ Learn more

Code examples

Java	Prototype in Java	Prototype in Java
C++	Prototype in C++: Before and after	Prototype in C++
PHP	Prototype in PHP	Prototype in PHP

Design Patterns
AntiPatterns

My account
Forum

[Refactoring](#)

[UML](#)

[Contact us](#)

[About us](#)

© 2007-2018 SourceMaking.com
All rights reserved.

[Terms / Privacy policy](#)