Table of Contents

About

factory method : simple example code

build

```
g++ *.cpp -o run
```

different games

- BombedMazeGame:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

- MazeGame:

```
class MazeGame {
public:
    Maze* CreateMaze();

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
};
```

main

- main will create maze of differnt games (MazeGame/BombedMazeGame)
- but they will be Maze !!!

```
int main()
{
    MazeGame myGame;            //(1)
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!//
    Maze* m1 = myGame.CreateMaze();    //(1.1)
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!//
    std::cout << m1 << std::endl;
    BombedMazeGame myBombedGame;      //(2)
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!//
    Maze* m2 = myBombedGame.CreateMaze(); //(2.1)
    //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!//
    std::cout << m2 << std::endl;
```

Maze.h

```
class Maze {

public:
  Maze();
  Room* RoomNo(int n) const;
  void AddRoom(Room *room);
  virtual Maze* Clone() const;
```

- (1.1) CreateMaze for myGame which is MazeGame for Maze class !!!
- (2.1) CreateMaze for myBombedGame also is Maze class !!!
- will call the CreateMaze of Maze !!!

MazeGame.cpp

```
Maze* MazeGame::CreateMaze() {

    std::cout << "MazeGame::CreateMaze()" << std::endl;


    Maze* aMaze = MakeMaze();        //(3)

    Room* r1 = MakeRoom(1);        //(4)
    aMaze->AddRoom(r1);
```

case#1 MazeGame

- (3) MakeMaze is (virtual Maze* MakeMaze()) member of at MazeGame that return new Maze;
- (4) makeRoom is member of MazeGame

case#2 myBombedGame

- (3) myBombedGame has no member MakeMaze but it inherit (class BombedMazeGame : public MazeGame) MakeMaze
- (3) MakeMaze will run form MazeGame
- (4) makeRoom is override from myBombedGame and will run its version

run

```
case#1
MazeGame.cpp:CreateMaze()
MazeGame.cpp:CreateMaze() call MakeMaze
MazeGame.h::MakeMaze        //(3)
MazeGame.h::MakeRoom        //(4)

case#2
BombedMazeGame::BombedMazeGame()
MazeGame.cpp:CreateMaze()
MazeGame.cpp:CreateMaze() call MakeMaze
MazeGame.h::MakeMaze        //(3)
BombedMazeGame.h::MakeRoom    //(4)
```

Factory Method (2)

- The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.

bad example

```
// A design without factory pattern
#include <iostream>
using namespace std;

// Library classes
class Vehicle {
public:
    virtual void printVehicle() = 0;
};
class TwoWheeler : public Vehicle {
public:
    void printVehicle() {
        cout << "I am two wheeler" << endl;
    }
};
class FourWheeler : public Vehicle {
    public:
    void printVehicle() {
        cout << "I am four wheeler" << endl;
    }
};

// Client (or user) class
class Client {
public:
    Client(int type) {

        // Client explicitly creates classes according to type
        if (type == 1)
            pVehicle = new TwoWheeler();
        else if (type == 2)
            pVehicle = new FourWheeler();
        else
            pVehicle = NULL;
    }

    ~Client() {
        if (pVehicle)
        {
            delete[] pVehicle;
            pVehicle = NULL;
        }
    }

    Vehicle* getVehicle() {
        return pVehicle;
    }
private:
    Vehicle *pVehicle;
};

// Driver program
int main() {
    Client *pClient = new Client(1);
    Vehicle * pVehicle = pClient->getVehicle();
    pVehicle->printVehicle();
    return 0;
}
```

- output:

```
I am two wheeler
```

- bad desing:
  - As you must have observed in the above example, Client creates objects of either TwoWheeler or FourWheeler based on some input during constructing its object.
  - Say, library introduces a new class ThreeWheeler to incorporate three wheeler vehicles also. What would happen? Client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler. Which in turn will need Client to be recompiled. So, each time a new change is made at the library side, Client would need to make some corresponding changes at its end and recompile the code. Sounds bad? This is a very bad practice of design

good design

- create a static (or factory) method

```
// C++ program to demonstrate factory method design pattern
#include <iostream>
using namespace std;

enum VehicleType {
    VT_TwoWheeler, VT_ThreeWheeler, VT_FourWheeler
};

// Library classes
class Vehicle {
```

```
public:
    virtual void printVehicle() = 0;
    static Vehicle* Create(VehicleType type);
};
class TwoWheeler : public Vehicle {
public:
    void printVehicle() {
        cout << "I am two wheeler" << endl;
    }
};
class ThreeWheeler : public Vehicle {
public:
    void printVehicle() {
        cout << "I am three wheeler" << endl;
    }
};
class FourWheeler : public Vehicle {
    public:
    void printVehicle() {
        cout << "I am four wheeler" << endl;
    }
};

// Factory method to create objects of different types.
// Change is required only in this function to create a new object type
Vehicle* Vehicle::Create(VehicleType type) {
    if (type == VT_TwoWheeler)
        return new TwoWheeler();
    else if (type == VT_ThreeWheeler)
        return new ThreeWheeler();
    else if (type == VT_FourWheeler)
        return new FourWheeler();
    else return NULL;
}

// Client class
class Client {
public:

    // Client doesn't explicitly create objects
    // but passes type to factory method "Create()"
    Client()
    {
        VehicleType type = VT_ThreeWheeler;
        pVehicle = Vehicle::Create(type);
    }
    ~Client() {
        if (pVehicle) {
            delete[] pVehicle;
            pVehicle = NULL;
        }
    }
    Vehicle* getVehicle() {
        return pVehicle;
    }

private:
    Vehicle *pVehicle;
};

// Driver program
int main() {
    Client *pClient = new Client();
    Vehicle * pVehicle = pClient->getVehicle();
    pVehicle->printVehicle();
    return 0;
}
```