Design Patterns

Strategy Pattern

- If a class relies on complex external functionality, use strategy pattern.
- Allows to add/switch functionality of the class without changing its implementation.
- All strategies must conform to one strategy interface.

```cpp
class Strategy {
 public:
  virtual void Print() const = 0;
};
```

```cpp
class StrategyA : public Strategy {
 public:
  void Print() const override { cout << "A" << endl; }
};

class StrategyB : public Strategy {
 public:
  void Print() const override { cout << "B" << endl; }
};
```

So far, nothing is new with this source code. We just defined an `interface` and then we derived 2 classes from this `interface` and implemented the virtual methods.

```cpp
class MyClass {
 public:
  explicit MyClass(const Strategy& s) : strategy_(s) {}
  void Print() const { strategy_.Print(); }

 private:
  const Strategy& strategy_;
};
```

- `MyClass` holds a `const reference` to an object of type `Strategy`.
- The strategy will be "picked" when we create an object of the class `MyClass`.
- We don't need to hold a reference to all the `types` of available strategies.
- The `Print` method has nothing to do with the one we've defined in `Strategy`.

- Create two different `strategies` objects

```cpp
StrategyA strategy_a = StrategyA();
StrategyB strategy_b = StrategyB();
```

- Create 2 objects that will use the `Strategy` pattern. We pick which `Print` strategy to use when we construct these objects.

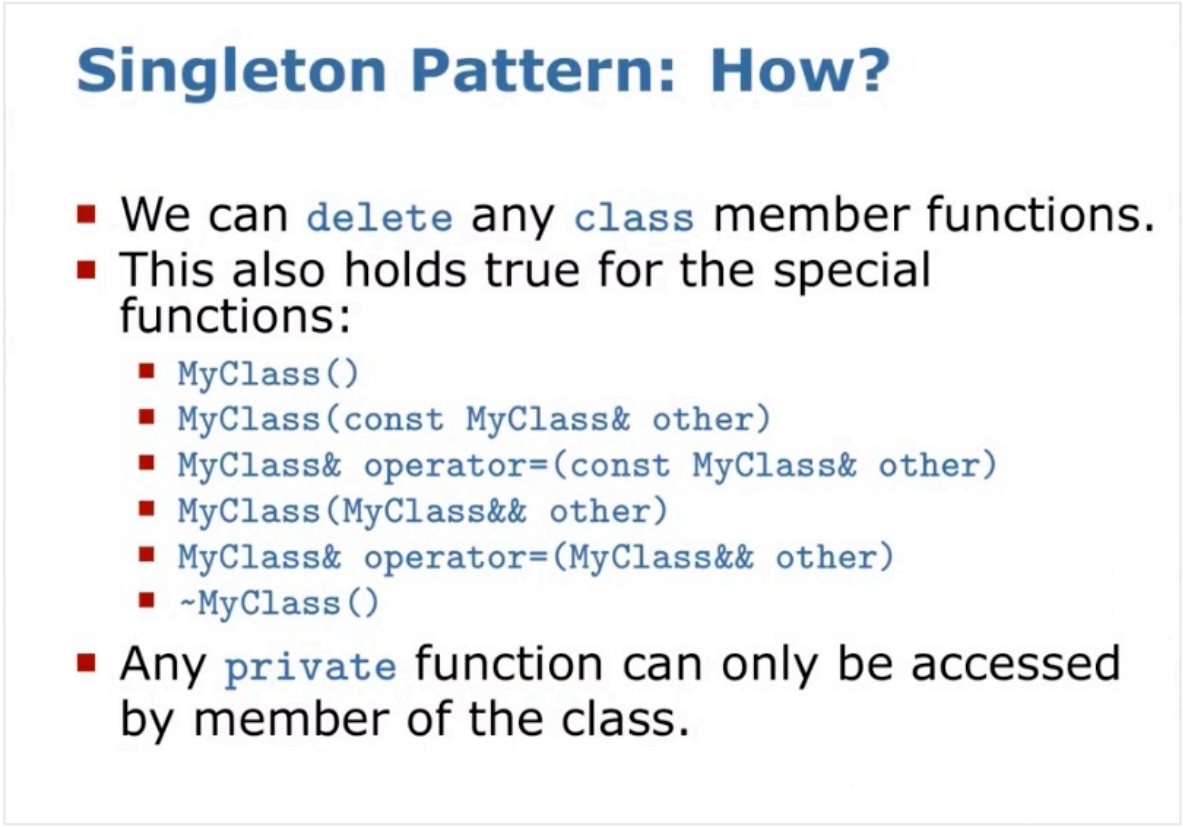
```cpp
MyClass obj_1(strategy_a);
MyClass obj_2(strategy_b);
```

- Use the objects in a " polymorphic" fashion. Both objects will have a `Print` method but they will call different functions according to the `Strategy` we picked when we build the objects.

```cpp
obj_1.Print();
obj_2.Print();
```

Singleton Pattern

- We want only one instance of a given class.
- Without C++ this would be a if/else mess.
- We can make sure that nobody creates more than 1 instance of a given class, at compile time.
- This doesn't use any object.



Singleton Pattern: How?

- Let's **hide** the default `Constructor` and also the `destructor`.

```
1 class Singleton {
2  private:
3   Singleton() = default;
4   ~Singleton() = default;
5 };
```

- This completely **disable** the possibility to create a `Singleton` object or destroy it.

Singleton Pattern: How?

- And now let's delete any copy capability:
  - `Copy Constructor`.
  - `Copy Assigment Operator`.

```
1 class Singleton {
2  public:
3   Singleton(const Singleton&) = delete;
4   void operator=(const Singleton&) = delete;
5 };
```

- This completely **disable** the possibility to copy any existing `Singleton` object.

Singleton Pattern: What now?

- Now we need to create at least **one** instance of the `Singleton` class.
- **How?** Compiler to the rescue:
  - We can create **one unique** instance of the class.
  - At compile time …
  - Using `static` !.

```
1 class Singleton {
2  public:
3   static Singleton& GetInstance() {
4     static Singleton instance;
5     return instance;
6   }
7 };
```

Singleton Pattern: Completed

```
1  class Singleton {
2   private:
3    Singleton() = default;
4    ~Singleton() = default;
5
6   public:
7    Singleton(const Singleton&) = delete;
8    void operator=(const Singleton&) = delete;
9    static Singleton& GetInstance() {
10     static Singleton instance;
11     return instance;
12   }
13 };
```

Singleton Pattern: Usage

```cpp
#include "Singleton.hpp"

int main() {
  auto& singleton = Singleton::GetInstance();
  // ...
  // do stuff with singleton, the only instance.
  // ...

  Singleton s1;                // Compiler Error!
  Singleton s2(singleton);     // Compiler Error!
  Singleton s3 = singleton;    // Compiler Error!

  return 0;
}
```

CRPT Pattern

CRPT Pattern

```cpp
#include <boost/core/demangle.hpp>
using boost::core::demangle;

template <typename T>
class Printable {
 public:
  explicit Printable() {
    // Always print its type when created
    cout << demangle(typeid(T).name()) << " created\n";
  }
};

class Example1 : public Printable<Example1> {};
class Example2 : public Printable<Example2> {};
class Example3 : public Printable<Example3> {};
```

CRPT Pattern

Usage:

```
int main() {
  const Example1 obj1;
  const Example2 obj2;
  const Example3 obj3;
  return 0;
}
```

Output:

```
Example1 Created
Example2 Created
Example3 Created
```