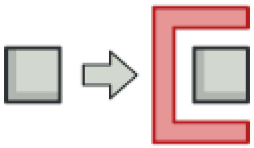

Proxy in C++

Proxy is a structural design pattern that provides an object that acts as a substitute for a real service object used by a client. A proxy receives client requests, does some work (access control, caching, etc.) and then passes the request to a service object.

The proxy object has the same interface as a service, which makes it interchangeable with a real object when passed to a client.

Learn more about Proxy

Complexity:

Popularity:

Usage examples: While the Proxy pattern isn't a frequent guest in most C++ applications, it's still very handy in some special cases. It's irreplaceable when you want to add some additional behaviors to an object of some existing class without changing the client code.

Identification: Proxies delegate all of the real work to some other object. Each proxy method should, in the end, refer to a service object unless the proxy is a subclass of a service.

Conceptual Example

This example illustrates the structure of the **Proxy** 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

```
#include <iostream>
/**
```

```
 * The Subject interface declares common operations for both RealSubject and the
 * Proxy. As long as the client works with RealSubject using this interface,
 * you'll be able to pass it a proxy instead of a real subject.
 */
class Subject {
 public:
  virtual void Request() const = 0;
};
/**
 * The RealSubject contains some core business logic. Usually, RealSubjects are
 * capable of doing some useful work which may also be very slow or sensitive -
 * e.g. correcting input data. A Proxy can solve these issues without any
 * changes to the RealSubject's code.
 */
class RealSubject : public Subject {
 public:
  void Request() const override {
    std::cout << "RealSubject: Handling request.\n";
  }
};
/**
 * The Proxy has an interface identical to the RealSubject.
 */
class Proxy : public Subject {
  /**
   * @var RealSubject
   */
 private:
  RealSubject *real_subject_;

  bool CheckAccess() const {
    // Some real checks should go here.
    std::cout << "Proxy: Checking access prior to firing a real request.\n";
    return true;
  }
  void LogAccess() const {
    std::cout << "Proxy: Logging the time of request.\n";
  }

  /**
   * The Proxy maintains a reference to an object of the RealSubject class. It
   * can be either lazy-loaded or passed to the Proxy by the client.
   */
 public:
  Proxy(RealSubject *real_subject) : real_subject_(new RealSubject(*real_subject)) {
  }

  ~Proxy() {
    delete real_subject_;
  }
  /**
   * The most common applications of the Proxy pattern are lazy loading,
```

```
   * caching, controlling the access, logging, etc. A Proxy can perform one of
   * these things and then, depending on the result, pass the execution to the
   * same method in a linked RealSubject object.
   */
  void Request() const override {
    if (this->CheckAccess()) {
      this->real_subject_->Request();
      this->LogAccess();
    }
  }
};
/**
 * The client code is supposed to work with all objects (both subjects and
 * proxies) via the Subject interface in order to support both real subjects and
 * proxies. In real life, however, clients mostly work with their real subjects
 * directly. In this case, to implement the pattern more easily, you can extend
 * your proxy from the real subject's class.
 */
void ClientCode(const Subject &subject) {
  // ...
  subject.Request();
  // ...
}

int main() {
  std::cout << "Client: Executing the client code with a real subject:\n";
  RealSubject *real_subject = new RealSubject;
  ClientCode(*real_subject);
  std::cout << "\n";
  std::cout << "Client: Executing the same client code with a proxy:\n";
  Proxy *proxy = new Proxy(real_subject);
  ClientCode(*proxy);

  delete real_subject;
  delete proxy;
  return 0;
}
```

Output.txt: Execution result

```
Client: Executing the client code with a real subject:
RealSubject: Handling request.

Client: Executing the same client code with a proxy:
Proxy: Checking access prior to firing a real request.
```

```
RealSubject: Handling request.
Proxy: Logging the time of request.
```