Паттерны проектирования C++

Александр Смаль

CS центр
11 мая 2016
Санкт-Петербург

Уже обсудили

1. Singleton
2. Adapter (`stack, queue`)
3. Type-erasure (`function, any`)
4. Tag-dispatching (алгоритмы STL)
5. Traits (`iterator_traits`)
6. Proxy access (`vector<bool>`)
7. Small Object Optimization
8. RAII
9. Smart pointers
10. CRTP

Класс Singleton

В C++ 11 данный синглтон не требует дополнительной синхронизации в многопоточных приложениях — гарантируется, что конструктор s будет вызван единожды.

```
struct Singleton
{
    static Singleton & instance()
    {
            static Singleton s;
            return s;
    }

    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

private:
    Singleton() {}
};
```

Pointer to implementation

Пусть у на был следующий класс.

```
struct Book {
  void print();
private:
  std::string  contents_;
};
```

В дальнейшем мы его изменили.

```
struct Book {
  void print();
private:
  std::string  contents_;
  std::string  title_;
}
```

Это приведёт к перекомпиляции всего кода, который использует Book.

Pointer to implementation

Пусть у на был следующий класс.

```
/* public.h */
struct Book {
  Book();
  ~Book();
  void print();
private:
  struct BookImpl* p_;
};

/* private.h */
#include "public.h"
#include <iostream>
struct BookImpl {
  void print();
private:
  std::string  contents_;
  std::string  title_;
}
```

Pointer to implementation

Что мы получили:

1. При изменении `BookImpl` нужно перекомпилировать только реализацию класса `Book`.
2. Можно скрыть реализацию `BookImpl`: передать библиотеку с классами `Book` и `BookImpl` и заголовочный файл `public.h`.
3. При раздельной компиляции можно поддерживать бинарную совместимость, если зафиксировать класс `Book`.

Expression Template

```
string a("Computer"), b("Science"), c("Center");

string res = a + " " + b + " " + c; // не стоит использовать auto
```

```
template<class O1, class O2>
struct string_expr {
    size_t size() const;      // суммарная длина всех строк
    operator string() const; // склейка всех строк в одну
private:
    O1 & o1;
    O2 & o2;
};

string_expr<string, string>
    operator+(string const& a, string const& b);

template<class O1, class O2>
string_expr<string, string_expr<O1, O2> >
    operator+(string const& a, string_expr<O1, O2> const& b);
...
```

Visitor

```
struct SizeVisitor {
    void visit(char const * s)   { res_ += strlen(s); }
    void visit(string const & s) { res_ += s.size(); }
    template<class T>
    void visit(T const& t) { t.visit(*this); }

    size_t value() const { return res_; }
private:
    size_t res_ = 0;
};
```

Внутри класса string_expr:

```
    template<class Visitor>
    void visit(Visitor & v) {
        v.visit(o1);
        v.visit(o2);
    }
    size_t size() const {
        SizeVisitor v;
        visit(v);
        return v.value();
    }
```

Named constructor

```
struct Game {
    // named constructor
    static Game createSinglePlayerGame() { return Game(0); }

    // named constructor
    static Game createMultiPlayerGame()  { return Game(1); }

protected:
    Game (int game_type);
};

int main(void)
{
   // Using named constructor
   Game g1 = Game::createSinglePlayerGame();

   // multiplayer game; without named constructor (does not compile)
   Game g2 = Game(1);
}
```

Attorney-Client

```
class Foo {
  void A(int a);
  void B(float b);
  void C(double c);
  friend class Bar;
};
// Needs access to Foo::A and Foo::B only
struct Bar { };
```

```
class Client {
  void A(int a);
  void B(float b);
  void C(double c);
  friend struct Attorney;
};

class Attorney {
  static void callA(Client & c, int a)   { c.A(a); }
  static void callB(Client & c, float b) { c.B(b); }
  friend struct Bar;
};
// Bar now has access to only Client::A and Client::B through the Attorney.
struct Bar { };
```

Strategy

```
template<class T>
struct NewCreator {
    static T* create()          { return new T(); }
    static void destroy(T * t)  { delete t; }
};

template<class T>
struct MallocCreator {
    static T* create() {
        void* buff = malloc(sizeof(T));
        if (buff == 0) return 0;
        return new (buff) T();
    }
    static void destroy(T * t)  { t->~T(); free(t); }
};

template<template <class T> CreationPolicy>
struct WidgetManager : protected CreationPolicy<Widget>
{
    ...
};
```

Non-Virtual Interface

```
struct Base {
    virtual ~Base() {}

    void show() { do_show(); }

    void load(std::string const& filename) {
        // Здесь можно проверить, существует ли файл
        do_load(filename);
        // Здесь можно проверить, валидны ли прочитанные данные
    }

    void save(std::string const& filename) {
        do_save(filename);
        // Здесь можно проверить, успешно ли прошла запись
    }
protected:
    virtual void do_show() = 0;
    virtual void do_load(std::string const& filename) = 0;
    virtual void do_save(std::string const& filename) = 0;
};

int main() {
    Base * b = new Derived();

    b->load("~/Schema.xml");
    b->show();
    b->save("~/Schema.xml");
}
```