[Хранитель]

[Команда]

Хранитель

Напишем шаблон хранителя. Необходимо организовать передачу объектов в холдер. Если передавать объект по ссылке, то будет не торт; если по значению – будет создаваться копия, причем может произойти попытка дважды уничтожить один и тотже объект. Создаётся специальная оболочка для передачи – трансфер-капсула. Реализуем сначала её, потом холдер.

К счастью, не так уж трудно написать небольшой шаблон класса, который инкапсулирует стратегию второго примера. Идея состоит в том, чтобы написать класс, который ведет себя подобно указателю, но при этом уничтожает объект, на который указывает, при уничтожении самого указывающего объекта или при присвоении ему другого указателя. Назовем такой класс *holder* (держатель, хранитель), поскольку он предназначен для безопасного хранения объекта при выполнении различных вычислений. Ниже показано, как это можно сделать.

20.1.3. Holder в качестве члена класса

Избежать утечки ресурсов можно также, используя `Holder` внутри класса. Если член класса имеет этот тип, а не обычный тип указателя, нам зачастую не требуется иметь дело с этим членом в деструкторе, поскольку такой объект будет удален при удалении члена `Holder`. Кроме того, этот шаблон помогает избежать утечки ресурсов в случае генерации исключения при инициализации объекта. Заметим, что деструкторы вызываются только для полностью сконструированных объектов и, если исключение генерируется в конструкторе, то деструкторы вызываются только для тех объектов-членов, конструкторы которых завершили свою работу без ошибок. Если не использовать шаблон `Holder`, это приведет к утечке ресурсов: например, если после первого успешного выделения ресурсов последовало неуспешное.

```
using namespace std;

template<typename T>
class Holder;

template<typename T>
class Trule //TRansfer capsULE //используется только для передачи; чтобы при передачи в метод функции или возврата
  //функции из метода, не возникло утечки
{
private:
  T *ptr;
public:
  Trule(Holder<T> &h)
    { ptr = h.release(); }
  ~Trule() //при передаче параметров также возможна отработка исключений; необходимо позаботиться об уничтожении труля
    { delete ptr;
    cout << "trule delete"; }
```

```
private:
  Trule(Trule<T> &) = delete;
  Trule<T> &operator=(const Trule<T> &) = delete;
  friend class Holder<T>; //для простоты обращения
};

template<typename T>
class Holder {
private:
  T *ptr;
public:
  Holder() : ptr(0) {} //когда создаётся холдер указателя, он не должен принимать указатель откуда-то извне
  //- это может привести к тому, что холдер может держать указатель на уже освобождённую память
  explicit Holder(T *p) : ptr(p) {} //нужен явный вызов конструктора, явное создание объекта
  ~Holder() {
    delete ptr;
    cout << "delete";
  }
  Holder(const Holder<T>&) = delete;
  Holder<T>& operator=(const Holder<T> &) = delete;
  //or
  // Присвоение нового указателя
  /*Holder<T>& operator= (T* p) {
    delete ptr;
    ptr = p;
    return *this;
  }*/
//холдер должен быть прозрачен: черзе него мы должны мочь обратиться ко всему что он держит
  T* operator->() const { return ptr; }
//также нужна и работа со ссылкой на объект
  T& operator*() const { return *ptr; }

//холдер берёт на себя указатель капсулы
  Holder(const Trule<T> &t) {
    ptr = t.ptr; //m.ptr - константа, поэтому приходится извращаться
    const_cast<Trule<T> &>(t).ptr = 0;
  }

//присваивание
  Holder<T> &operator=(Trule<T> const &t) {
    delete ptr;
    ptr = t.ptr;
    const_cast<Trule<T> &>(t).ptr = 0;
    return *this;
  }

  T *release() {
    T *p = ptr;
    ptr = 0;
    return p;
```

```
  }

  // Обмен владением с Другим хранителем
  void exchange_with(Holder<T>& h) {
    swap(ptr,h.ptr);
  }
// Обмен владением с другим указателем
  void exchange_with(T*& p) {
    swap (ptr, p) ;
  }
};
class A {
public:
  void f() {
    cout << "ok";
    throw 1;
  }
};
int main() {
  try {
    Holder<A> obj(new A);
    obj->f();
  }

  catch(int) {
    //res: delete in Holder
  }
}
```

Команда

Паттерн команда.
Основополагающая идея шаблона заключается в использовании единого интерфейса для описания всех типов операций, которые можно производить с системой. Тогда для добавления в систему поддержки новой операции достаточно реализовать предлагаемый интерфейс. Таким образом, каждая операция представляется самостоятельным объектом инкапсулирующим некоторый набор дополнительных свойств. Систем, в свою очередь, приобретает возможно выполнять дополнительный набор действий над запросами (объектами). Это протоколирование, отмена предыдущего действия повторение последующего и т.д.

При событии (нажатии на кнопку, выделении компоненты) можно реализовывать событийную схему или действовать по принципу вызова и передачи команды. Команда может использоваться даже в асинхронном взаимодействии, но в целом появилась именно в синхронном. В первой лабе – имя команды (номер) данные (юнион структур). Можно подойти и более широко. При передаче команды должно выполниться какое-то действие – можно передавать и действие, которое надо выполнить. Команда будет вызываться для нужного нам объекта. Одна команда может разбиваться на целую цепочку действий (повернуть объект – повернуть все его подобъекты). Можно использовать паттерн композицирования.

```
class Command //базовый класс команды
```

```
{
public:
  virtual void execute() = 0;
  virtual ~Command();

protected:
  Command() {}
};

template<typename Reciever>
class SimpleCommand: public Command
{
private:
  typedef void (Reciever::*Action)();
  Reciever* rec; //указатель на объект
  Action act; //указатель на метод
public:
  SimpleCommand(Reciever* r, Action a); //конструктор
  void execute() //экзекутор
  {
    return (rec->*act)();
  }
};
```

При создании команды происходит связывание с объектом, однако оно может происходить и при приеме команды. Экзекут может принимать объект, для которого выполнить данную команду.
Здесь реализовывается простая команда, дальше композируем – выполняется тоже самое но, например, обходя список команд (объект для которого выполняется список команд должен быть один).
Если команда несёт какие-то данные, то в конкоманде (НЕ В ШАБЛОНЕ СИМПЛКОМАНДЕ!) будут дополнительные инты флоаты и так далее.