RAII

Введение в объектно-ориентированное программирование

24.04.2025

Владение ресурсом

- Памятью владеет тот, кто её выделяет и освобождает.

```
S *p = new S;
foo(p); // foo(S*);
delete p;
```

- Что может пойти не так в этом коде?

Владение ресурсом

- Памятью владеет тот, кто её выделяет и освобождает.

```
S *p = new S;
foo(p); // foo(S *p) { delete p; }
delete p;
```

- Что может пойти не так в этом коде?
- В общем случае память это только один из возможных ресурсов.

Забавный пример

```
template <typename S> int foo(int n) {
  S *p = new S{n};
// .... some code ....
  if (condition) {
    delete p;
    return FAILURE;
}
// .... some code ....
  delete p;
  return SUCCESS;
}
```

• Хотелось бы иметь одну точку освобождения чтобы избежать проблем.

Страшное goto

```
template <typename S> int foo(int n) {
S *p = new S{n}; int result = SUCCESS;
// .... some code ....
if (condition) {
result = FAILURE;
goto cleanup;
}
// .... some code ....
cleanup:
delete p;
return result;
}
```

Социально-приемлимое goto

```
template <typename S> int foo(int n) {
S *p = new S{n}; int result = SUCCESS;
do {
// .... some code ....
if (condition) {
result = FAILURE;
break;
}
// .... some code ....
} while(0);
delete p;
return result;
}
```

Отступление: goto considered harmful

- Что вы думаете о вот таком коде?

```
struct X {
  int smth = 42;
};
int foo(int cond) {
  switch(cond) {
    case 0: X x;
    case 1: return x.smth; // 42?
  }
}
```

Отступление: goto considered harmful

- Что вы думаете о вот таком коде?

```
struct X {
  int smth = 42;
};
int foo(int cond) {
  switch(cond) {
    case 0: X x;
    case 1: return x.smth; // FAIL
  }
}
```

Обсуждение

- Какие мы знаем goto-маскирующие конструкции?

switch-case, break, continue, return, ещё?

- Будьте со всеми ними крайне осторожны при работе с конструкторами и деструкторами. Ваш выбор – явные блоки.

```
int foo(int cond) {
switch(cond) {
case 0: { X x; }
case 1: return x.smth; // очевидная ошибка, x не виден
}
}
```

RAII: resource acquisition is initialization

• Чтобы не писать goto можно спроектировать класс, в котором конструктор захватывает владение, а деструктор освобождает ресурс.

```
template <typename S> int foo (int n) {
ScopedPointer<S> p{new S(n)}; // ownership passed
// .... some code ....
if (condition)
return FAILURE; // dtor called: delete
// .... some code ....
return SUCCESS; // dtor called: delete
}
```

• Как этот класс мог бы выглядеть?

RAII обёртка

• Как мог бы выглядеть упомянутый ScopedPointer?

```
template <typename T> class ScopedPointer {
T *ptr_;
public:
ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
~ScopedPointer() { delete ptr_; }
```

• И у нас есть две проблемы. Первая: как написать копирование/присваивание.

• Вторая: как сделать с ним что-то полезное, не дав утечь указателю?

Глубокое копирование

• Начнем с копирования.

```
template <typename T> class ScopedPointer {
T *ptr_;
public:
ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
~ScopedPointer() { delete ptr_; }
ScopedPointer(const ScopedPointer& rhs) : ptr_(new T*rhs.ptr_) {}
// как бы вы реализовали присваивание?
ScopedPointer& operator= (const ScopedPointer& rhs);
```

Доступ к состоянию

• Можно сделать просто функцию вроде access.

```
template <typename T> class ScopedPointer {
T *ptr_;
public:
ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
~ScopedPointer() { delete ptr_; }
T& access() { return *ptr_; }
const T& access() const { return *ptr_; }
```

• Итог немного многословен.

```
ScopedPointer<S> p{new S(n)}; int x = p.access().x; // (*p).x
```

Перегрузка разыменования

• Разыменование указателя это оператор и он перегружается.

```
template <typename T> class ScopedPointer {
T *ptr_;
public:
ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
~ScopedPointer() { delete ptr;}
T& operator*() { return *ptr_; }
const T& operator*() const { return *ptr_; }
```

• Уже сейчас стало гораздо лучше, но хотелось бы, конечно, стрелочку.

```
ScopedPointer<S> p{new S(n)}; int x = (*p).x; // p->x
```

Проблема со стрелочкой

```
template <typename T> class ScopedPointer {
T *ptr_;
public:
T& operator*() { return *ptr_; }
const T& operator*() const { return *ptr_; }
??? operator->() { return ???; } // например p->x, для T::x
```

- А что собственно возвращать?

Решение: drill down

```
template <typename T> class ScopedPointer {
T *ptr_;
public:
T& operator*() { return *ptr_; }
const T& operator*() const { return *ptr_; }
T* operator->() { return ptr_; } // например p->x, для T::x
const T* operator->() const { return ptr_; }
```

- Вызов p->x эквивалентен (p.operator->())->x и так сколько угодно раз.
- Стрелочка как бы "зарывается"в глубину на столько уровней на сколько может.