Лекция 6. Инициализация и const

Александр Смаль

CS центр
3 октября 2017
Санкт-Петербург

Модификаторы доступа

Модификаторы доступа позволяют ограничивать доступ к методам и полям класса.

```
struct IntArray
{
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i]; }
    size_t size()        { return size_; }

private:
    size_t size_;
    int *  data_;
};
```

Ключевое слово `class`

Ключевое слово `struct` можно заменить на `class`, тогда поля и методы по умолчанию будут private.

```
class IntArray
{
public:
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i]; }
    size_t size()        { return size_; }
private:
    size_t size_;
    int *  data_;
};
```

Инварианты класса

- Выделение *публичного интерфейса* позволяет поддерживать *инварианты класса*
  (сохранять данные объекта в согласованном состоянии).

```
struct IntArray {
    ...
    size_t size_;
    int *  data_; // массив размера size_
};
```

- Для сохранения инвариантов класса:
  1. все поля должны быть закрытыми,
  2. публичные методы должны сохранять инварианты класса.
- Закрытие полей класса позволяет абстрагироваться от способа хранения данных объекта.

Публичный интерфейс

```
struct IntArray
{
    void resize(size_t nsize)
    {
        int * ndata = new int[nsize];
        size_t n = nsize > size_ ? size_ : nsize;
        for (size_t i = 0; i != n; ++i)
            ndata[i] = data_[i];
        delete [] data_;
        data_ = ndata;
        size_ = nsize;
    }
private:
    size_t size_;
    int *  data_;
};
```

Абстракция

```
struct IntArray
{
public:
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i]; }
    size_t size()        { return size_; }

private:
    size_t size_;
    int *  data_;
};
```

Абстракция

```
struct IntArray
{
public:
    explicit IntArray(size_t size)
        : data_(new int[size + 1])
    {
        data_[0] = size;
    }
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i + 1]; }
    size_t size()        { return data_[0]; }

private:
    int *  data_;
};
```

Определение констант

- Ключевое слово `const` позволяет определять типизированные константы.

```
double const pi = 3.1415926535;
int const day_seconds = 24 * 60 * 60;
// массив констант
int const days[12] = {31, 28, 31,
                      30, 31, 30,
                      31, 31, 30,
                      31, 30, 31};
```

- Попытка изменить константные данные приводит к неопределённому поведению.

```
    int * may = (int *) &days[4];
    *may = 30;
```

Указатели и const

В C++ можно определить как константный указатель, так и указатель на константу:

```
int a = 10;
const int * p1 = &a; // указатель на константу
int const * p2 = &a; // указатель на константу
*p1 = 20; // ошибка
p2  = 0;  // OK

int * const p3 = &a; // константный указатель
*p3 = 30; // OK
p3  = 0;  // ошибка

// константный указатель на константу
int const * const p4 = &a;
*p4 = 30; // ошибка
p4  = 0;  // ошибка
```

Указатели и const

Можно использовать следующее правило:

“слово const делает неизменяемым тип слева от него”.

```
int a = 10;
int * p = &a;

// указатель на указатель на const int
int const **  p1 = &p;

// указатель на константный указатель на int
int * const * p2 = &p;

// константный указатель на указатель на int
int ** const  p3 = &p;
```

Ссылки и const

- Ссылка сама по себе является неизменяемой.

```
int a = 10;
int & const b = a; // ошибка
int const & c = a; // ссылка на константу
```

- Использование константных ссылок позволяет избежать копирования объектов при передаче в функцию.

```
Point midpoint(Segment const & s);
```

- По константной ссылке можно передавать rvalue.

```
Point p = midpoint(Segment(Point(0,0),
                           Point(1,1)));
```

Константные методы

- Методы классов могут быть объявлены как `const`.

```
struct IntArray {
    size_t size() const;
};
```

- Такие методы не могут менять поля объекта (тип `this` — указатель на `const`).
- У константных объектов (через указатель или ссылку на константу) можно вызывать только константные методы:

```
IntArray const * p = foo();
p->resize(); // ошибка
```

- Внутри константных методов можно вызывать только константные методы.

Две версии одного метода

- Слово `const` является частью сигнатуры метода.

```
size_t IntArray::size() const {return size_;}
```

- Можно определить две версии одного метода:

```
struct IntArray
{
    int get(size_t i) const {
        return data_[i];
    }
    int & get(size_t i) {
        return data_[i];
    }
private:
    size_t size_;
    int *  data_;
};
```

Синтаксическая и логическая константность

- Синтаксическая константность: константные методы не могут менять поля (обеспечивается компилятором).
- Логическая константность — нельзя менять те данные, которые определяют состояние объекта.

```
struct IntArray
{
    void foo() const {
        // нарушение логической константности
         data_[10] = 1;
    }
private:
    size_t size_;
    int *  data_;
};
```

Ключевое слово mutable

Ключевое слово mutable позволяет определять поля, которые можно изменять внутри константных методов:

```
struct IntArray
{
    size_t size() const {
        ++counter_;
        return size_;
    }

private:
    size_t size_;
    int *  data_;

    mutable size_t counter_;
};
```

Копирование объектов

```
struct IntArray
{
    ...
private:
    size_t size_;
    int *  data_;
};

int main() {
    IntArray a1(10);
    IntArray a2(20);
    IntArray a3 = a1; // копирование
    a2 = a1; // присваивание

    return 0;
}
```

Конструктор копирования

Если не определить конструктор копирования, то он сгенерируется компилятором.

```
struct IntArray
{
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_])
    {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Оператор присваивания

Если не определить оператор присваивания, то он тоже сгенерируется компилятором.

```
struct IntArray
{
    IntArray & operator=(IntArray const& a) {
        if (this != &a) {
            delete [] data_;
            size_ = a.size_;
            data_ = new int[size_];
            for (size_t i = 0; i != size_; ++i)
                data_[i] = a.data_[i];
        }
        return *this;
    }
    ...
};
```

Метод swap

```
struct IntArray
{
    void swap(IntArray & a) {
        size_t const t1 = size_;
        size_ = a.size_;
        a.size_ = t1;

        int * const t2 = data_;
        data_ = a.data_;
        a.data_ = t2;
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Метод swap

Можно использовать функцию std::swap и файла algorithm.

```
#include <algorithm>

struct IntArray
{
    void swap(IntArray & a)
    {
        std::swap(size_, a.size_);
        std::swap(data_, a.data_);
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Реализация оператора = при помощи swap

```
struct IntArray
{
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    IntArray & operator=(IntArray const& a) {
        if (this != &a)
            IntArray(a).swap(*this);
        return *this;
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Запрет копирования объектов

Для того, чтобы запретить копирование, нужно объявить конструктор копирования и оператор присваивания как `private` и не определять их.

```
struct IntArray
{
    ...
private:
    IntArray(IntArray const& a);
    IntArray & operator=(IntArray const& a);

    size_t size_;
    int *  data_;
};
```

Методы, генерируемые компилятором

Компилятор генерирует четыре метода:

1. конструктор по умолчанию,
2. конструктор копирования,
3. оператор присваивания,
4. деструктор.

Если потребовалось переопределить конструктор копирования, оператор присваивания или деструктор, то нужно переопределить и остальные методы из этого списка.

Поля и конструкторы

```
struct IntArray
{
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = 0;
    }
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    ...
private:
    size_t size_;
    int *  data_;
};
```

Деструктор, оператор присваивания и swap

```
~IntArray() {
    delete [] data_;
}

IntArray & operator=(IntArray const& a) {
    if (this != &a)
        IntArray(a).swap(*this);
    return *this;
}

void swap(IntArray & a) {
    std::swap(size_, a.size_);
    std::swap(data_, a.data_);
}
```

Методы

```
size_t size() const { return size_; }

int    get(size_t i)  const {
    return data_[i];
}
int &  get(size_t i)        {
    return data_[i];
}

void resize(size_t nsize) {
    IntArray t(nsize);
    size_t n = nsize > size_ ? size_ : nsize;
    for (size_t i = 0; i != n; ++i)
        t.data_[i] = data_[i];
    swap(t);
}
```