Advanced C++

#4 Классы.
Введение в ООП.

Menu

- Конструкторы и деструкторы
- Операторы классов
- Методы
- Перегрузка операторов
- Наследования

Конструкторы
Деструкторы

Классы

```
int square(int x)
{
    int tmp = x * x;
    return tmp;
}
```

Структуры

```
struct User
{
    std::string name;
    std::string email;
};

const User user =
    { "Bob", "bob@mail.ru" };

std::cout << user.name;
```

Структуры

Нужен поиск среди имен пользователей. Что эффективнее?

```
User users[N];
```

или

```
struct Users
{
    std::string name[N];
    std::string email[N];
};
```

Модификаторы доступа

```
struct A
{
public:
    int x; // Доступно всем
protected:
    int y; // Наследникам и объектам структуры (класса)
private:
    int z; // Только объектам структуры (класса)
};

A a;
a.x = 1; // ок
a.y = 1; // ошибка
a.z = 1; // ошибка
```

Модификаторы доступа

В C++ struct от class отличаются только модификатором доступа по умолчанию. По умолчанию содержимое struct доступно извне (public), а содержимое class - нет (private).

```
class A
{
    int x; // private
};

struct B
{
    int x; // public
}
```

Инкапсуляция

С помощью модификаторов доступа, можно добиться инкапсуляции -- сокрытия от конечного пользователя нашего класса всего внутреннего функционала методов и полей нашего класса.

Это делается с целью разграничить доступ извне, предоставляя внешней области видимости только работы с классом в рамках доступных методов и видимых полей.

Конструкторы

Из каких типов мы можем сконструировать наш класс?

```
class C {

    int x;
    C(int x_) { x=x_;};

};
```

(btw, В чем проблема?)

Конструкторы

Из каких типов мы можем сконструировать наш класс?

```
class C {
private:
    int x;
public:
    C(int x_) { x=x_;};
};
```

Конструкторы

```
class C {
private:
    int x;
public:
    C(){};
};

int main() {
    C x;
}
```

Что, если мы не хотим ничего определять?
Тогда компилятор сделает это за нас.

Конструкторы

```
class C {
private:
    int x;
};

int main() {
    C x;
}
```

Конструктор по умолчанию сгенерирует сам компилятор.
Но!

Конструкторы

```
class C {
private:
    const int x;
};

int main() {
    C x;
}
```

Если есть константные поля, то их нельзя инициализировать по умолчанию. Тогда будет ошибка компиляции.

Конструкторы

```
class C {
private:
    const int x = 10;
};

int main() {
    C x;
}
```

Конструкторы

```
class String {
public:
    String (const char* str, int size) {
        str_ = new char[size];
        ...
    }

private:
    char* str_;
    int size_;
}
```

Конструкторы

```
class String {
public:
    String() = default;
    String (const char* str, int size) {
        str_ = new char[size];
        ...
    }

private:
    char* str_;
    int size_;
}
```

Конструкторы

```
class String {
public:
    String() = delete;
    String (const char* str, int size) {
        str_ = new char[size];
        ...
    }

private:
    char* str_;
    int size_;
}
```

Деструкторы

```
class String {
public:
    String() = delete;
    String (const char* str, int size) {
        str_ = new char[size];
        ...
    }

    ~String () {delete [] str_;}

private:
    char* str_;
    int size_;
}
```

Деструкторы

```
class String {
public:
    String() = delete;
    String (const char* str, int size) {
        str_ = new char[size];
        ...
    }

    ~String () {delete [] str_;}

private:
    char* str_;
    int size_;
}
```

Конструктор копирования

```
...
String s1 = s;
...
```

Конструктор копирования

```
...
String s1 = s;
...

String (String s_) ?
```

Конструктор копирования

```
...
String s1 = s;
...

String (String& s_) ?
```

Конструктор копирования

```
...
String s1 = s;
...

String (const String& s_);
```

Можно также явно запретить:

```
String (const String& s_) = delete;
```

Конструктор копирования

```
...
String s1 = s;
...

String (const String& s_);
```

Можно также явно запретить:

```
String (const String& s_) = delete;
```

Конструктор копирования

```
class String {
public:
    String (const char * str, int size) {
        str_ = new char[size];
    }

    ~String () {delete [] str_;}
private:
    char * str_;
    int size_;
}
```

Конструктор копирования

```
int main() {
    String s = String(“abc”, 3);
    String s1 = s;
}
```

Будет крах. Почему?

Конструктор копирования

```
int main() {
    String s = String(“abc”, 3);
    String s1 = s;
}
```

Будет крах. Почему? Т.к. Произойдет двукратное удаление одного и того же куска памяти.

Конструктор копирования

```
String (const String& other) {
    str_ = new char[other.size_];
    for( int i = 0; i < other.size_; ++i ) {
        ...
    }
    ...
}
```

Конструктор копирования

```
String (const String& other) {
    str_ = new char[other.size_];
    for( int i = 0; i < other.size_; ++i ) {
        ...
    }
    ...
}
```

Можно и перегрузить. Это не запрещено. (const и не const)

Операторы

Операторы . и ->

Обращение к полю или вызов метода делается в объекте класса через .

S.size_

Если есть ссылка на объект класса, то это делается с помощью ->.

(*p).f() ⇔ p->f()

Ключевое слово this

Указатель на тот объект, в котором мы сейчас находимся.

```
class String {
public:
    String (const char * str, int size) {
        this->str = new char[size];
    }
...
```

Оператор присваивания

```
int main() {
    String s = String(“abc”, 3);
    String s1 = s;
    s = s1;
}
```

Оператор присваивания

```
int main() {
    String s = String(“abc”, 3);
    String s1 = s; // создается второй массив
    s = s1; // забывается массив у s
}
```

1) Два указателя в s1 и s
2) После приравнивания, забываем про массив у s
3) В итоге удалим два раза массив s1, но не удалим массив s

Оператор присваивания

Создаем новый объект и в него заполняем те же значения циклом

```
String& operator = (const String& s) {
        str = new char[s.size];
        for () {
            ...
        }
        ...
}
```

В чем проблемы?

Оператор присваивания

Создаем новый объект и в него заполняем те же значения циклом

```
String& operator = (const String& s) {
        delete[] str; // уже была выделена память!
        str = new char[s.size];
        for () {
            ...
        }
        …
}
```

Оператор присваивания

Создаем новый объект и в него заполняем те же значения циклом

```
String& operator = (const String& s) {
        delete[] str; // уже была выделена память!
        str = new char[s.size];
        for () {
            ...
        }
        ...
        return *this; // зачем? (чтобы работали
конструкции вида a=b=c
    }
```

Оператор присваивания

Возврат нужен для того, чтобы работали конструкции вида

```
a = b = c;
```

Оператор присваивания

Присваивание самому себе:

```
String s = String();
s = s;
```

Это тоже будет работать некорректно.

Оператор присваивания

Создаем новый объект и в него заполняем те же значения циклом

```
String& operator = (const String& s) {
        if (&s == *this) {return *this;};
        delete[] str; // уже была выделена память!
        str = new char[s.size];
        for () {
            ...
        }
        ...
        return *this; // зачем? (чтобы работали
конструкции вида a=b=c
    }
```

Правило трех

Иными словами, можно выдвинуть эмпирическое правило:

Если ваш класс требует определения хотя бы одного из конструкторов копирования, или оператора присваивания, или деструктора, то оно на самом деле требует определения сразу всех трех этих сущностей.

Список инициализации

```
class C {
    const int x = 3;
    int& y; // пока что просто определение (не
инициализация объекта C)
}

C(); // CE
```

Список инициализации

```
int z = 10;

class C {
    const int x = 3;
    int& y = z; // пока что просто определение (не
инициализация объекта C)
}

C(const int x, int& y) {
    this->x;
    this->y = y;
};
```

Список инициализации

```
int z = 10;

class C {
    const int x = 3;
    int& y = z; // пока что просто определение (не
инициализация объекта C)
}

C(const int x, int& y) : x(x), y(y) {};
```

Лучше всегда пользоваться списками инициализации, чтобы избежать лишних манипуляций с памятью.

Explicit

```
String s = s1 + 'a';
```

Произойдет неявное преобразование.

В итоге, получим вызов конструктора String(int size);, т.к. компилятор преобразует char к int.

Explicit

```
String s = s1 + 'a';
```

Произойдет неявное преобразование.

В итоге, получим вызов конструктора String(int size);, т.к. компилятор преобразует char к int.

Ключевое слово explicit решает эту проблему:

```
explicit String(int size_) {};
```

Const методы

Определим метод

```
int length() {return size;};

Const String s = “abc”;
s.length(); // CE
```

Почему CE?

Const методы

Определим метод

```
int length() {return size;};

Const String s = “abc”;
s.length(); // CE
```

Почему CE?

По дефолту каждый метод класса может менять поля класса. Чтобы метод класса можно было вызывать над константным инстансом вашего класса, нужно явно помечать, что этот метод не меняет полей класса.

Const методы

Определим метод

```
int length() const {return size;};

Const String s = “abc”;
s.length(); // CE
```

Почему CE?

По дефолту каждый метод класса может менять поля класса. Чтобы метод класса можно было вызывать над константным инстансом вашего класса, нужно явно помечать, что этот метод не меняет полей класса.

Mutable

```
mutable int length_call_counter = 0;

int length() const {
    length_call_counter++;
    return size;
};

Const String s = “abc”;
s.length(); // length_call_counter = 1
```

Friend

Если есть функция или оператор, которые не являются членом класса (т.е. Его левый аргумент не this).

```
friend f(...);
```

Друзьями можно объявлять целые классы, но дружба не транзитивна.

Статические поля

```
static int instances_counter;
```

Это поле не является полем какого-то конкретного класса. Оно статично существует для этого класса. Эти переменные лежат в статической памяти.

Указатель на член класса

```
class C{
public:
    int a;
    char b;
}

int main() {
    int C::* p = C::a; // указатель на член a
//                              класса C
}
```

Указатель на член класса

```
class C{
public:
    int a;
    char b;
}
int main() {
    int C::* p = C::a;
    C c;
    c.*p = 5; // оператор .*
}
```

Указатель на член класса

```
class C{
public:
    int a;
    char b;
}
int main() {
    int C::* p = &C::a;
    C* c;
    c->*p = 5; // оператор ->*
}
```

Перегрузка операторов

Перегрузка операторов

Рассмотрим на примере BigInteger.

```
BigInt operator + (const BigInt x) const {
    ...
}
```

Перегрузка операторов

Рассмотрим на примере BigInteger.

```
BigInt operator + (const BigInt& x) const {
    ...
}

BigInt& operator += (const BigInt& x) {
    return *this = *this + x;
}
```

Перегрузка операторов

Рассмотрим на примере BigInteger.

```
BigInt operator += (const BigInt& x) const {
    ...
    return *this;
}

BigInt& operator + (const BigInt& x) {
    BigInt y = *this;
    return y += x;
}
```

Перегрузка операторов

```
BigInt c; // correct

c + 5; // correct, если есть конструктор

5 + c; // нельзя
```

Все операторы внутри класса левым аргументом принимают this

Перегрузка операторов

За пределами класса:

```
BigInt operator + (const BigInt& x,
                   const BigInt& y) const {...}
```

Компилятор найдет оператор +, в котором int приводится к левому, а правый типа BigInt.

Тогда не нужен оператор внутри класса, определенный ранее.

Перегрузка операторов

BTW: Сработает ли такое?

```
BigInt c1 = 3;
BigInt c2 = 4;

c1+c2 = 5;
```

lvalue (locator value) представляет собой объект, который занимает идентифицируемое место в памяти (например, имеет адрес).

rvalue — это выражение, которое *не представляет* собой объект, который занимает идентифицируемое место в памяти.

Перегрузка операторов

BTW: Сработает ли такое?

```
BigInt c1 = 3;
BigInt c2 = 4;

c1+c2 = 5;
```

Можно просто возвращать const оператору +. Тогда мы избежим эту “проблему”.

Перегрузка операторов

BTW: Сработает ли такое?

```
BigInt c1 = 3;
BigInt c2 = 4;

c1+c2 = 5;
```

Можно просто возвращать const оператору +. Тогда мы избежим эту “проблему”.

Перегрузка операторов

Перегружаются следующие операторы:

+, -, *, &, |, ^, <<, >>, %, /

Можно переопределить вывод/ввод в/из потока.

Хотим:

```
BigInt c(5);
cout << c;
```

Перегрузка операторов

```
std::ostream & operator << (std::ostream& out,
                            C& c) {
...
}

std::istream & operator >> (std::istream& out,
                            C& c) {
...
}
```

Перегрузка операторов

Перегружаются следующие операторы:

Меняющие объекты:

`+, -, *, &, |, ^, <<, >>, %, /`

Сравнения:

`>, <, >=, <=, ==, !=`

Перегрузка операторов

Посмотрим оператор <

```
bool operator < (const BigInt& c) {...}
```

Как выразить остальные операторы?

И что не так с кодом?

Перегрузка операторов

Посмотрим оператор <

```
bool operator > (const BigInt& c) const {
    return c < *this;
}
```

Перегрузка операторов

Посмотрим оператор <

```
bool operator > (const BigInt& c) const {
    return c < *this;
}

bool operator == (const BigInt& c) const {
    return !(*this < c || *this > c);
}
```

Перегрузка операторов

Посмотрим оператор <

```
bool operator >= (const BigInt& c) const {
    return !(c < *this);
}

bool operator <= (const BigInt& c) const {
    return !(c > *this);
}
```

Перегрузка операторов

Перегружаются следующие операторы:

Меняющие объекты:

+, -, *, &, |, ^, <<, >>, %, /

Сравнения:

>, <, >=, <=, ==, !=

Инкремент/декремент:

++, --

Перегрузка операторов

Префиксный инкремент:

```
BigInt operator++() {
    return *this += 1
}
```

Перегрузка операторов

Префиксный инкремент:

```
BigInt& operator ++() {
    return *this += 1
}
```

Постфиксный инкремент:

```
const BigInt operator ++(int) {
    BigInt x = *this;
    &this += 1; return x;
}
```

Перегрузка операторов

Перегружаются следующие операторы:

Меняющие объекты:

+, -, *, &, |, ^, <<, >>, %, /

Сравнения:

>, <, >=, <=, ==, !=

Инкремент/декремент:

++, --

Скобки:

[], ()

Перегрузка операторов

```
char* arr;
...
char& operator[] (int i) {
    return arr[i];
}
```

Почему не const?

Перегрузка операторов

Можно объектам класса придать “функциональность” посредством перегрузки оператора ()

```
___ operator () ( … ) {

}
```

Нужно для написания функторов.

Перегрузка операторов

Перегружаются следующие операторы:

Меняющие объекты:

+, -, *, &, |, ^, <<, >>, %, /

Сравнения:

>, <, >=, <=, ==, !=

Инкремент/декремент:

++, --

Скобки:

[], ()

Указательные: *, &, ->

Other: &&, ||, ,

Перегрузка операторов

Нельзя перегрузить:

., ::, cast'ы (static, dynamic), ->*, .*, ?:

Нельзя создавать новые операторы и менять порядок вычисления операторов.

Но можно перегрузить c-style cast:

(Операторы преобразования типа)

```
operator int() {

}
```

Наследование

Наследование

```
Base <- Derived
```

1) `class Derived : public(private) Base {`
   `}`

   Приватное наследование уместно, когда надо “реализовать” с помощью класса-родителя.

2) `class Derived : protected Base {`
   `}`

   Protected -- видно мне, моим друзьям и моим наследникам

Наследование

```
Base <- Derived
b,       d,
fb()     fd()

main:
Derived d;

public: b, fb(), d, fd()
```

Вопрос: если в наследнике определена такая же функция, как и в базовом классе?

Наследование

```
Base <- Derived
b,       d,
fb()     fd()

main:
Derived d;

public: b, fb(), d, fd()
```

Ответ: выбираем те, которые мы определили в наследнике

Наследование

```
Base <- Derived
b,       d,
f()    f(int x)

main:
Derived d; d.f();

Что будет?
```

Наследование

```
Base <- Derived
b,       d,
f()    f(int x)

main:
Derived d; d.f();
```

Ошибка компиляции, т.к. Компилятор найдет уже найдет определение функции f в скоупе класса Derived и не пойдет искать выше в Base