Курс с++

Царьков Олег

Описание тикета OTSN-6

**Объекты, обертки.**

В языке кроме функций есть объекты. С функциями мы уже знакомы. Типичными примерами объектов являются переменные типа $int, double$, объект $std :: cout$ типа $std :: ostream$ и объект $std :: cin$ типа $std :: istream$.

Можно создавать не только свои объекты, но, оказывается, можно создавать и типы объектов. То есть можно описать собственный тип, который будет говорить, как устроен объект.

Для описания типа есть ключевое слово $class$.

Пример:

```
class MyClass {
private :
    int a;
public :
    int GetNumber() const {
        return a;
    }
};
```

Чем объяснять все подряд, легче просто разобрать этот пример.

Здесь определяется тип(класс) $MyClass$. В определении типа есть области видимости:

• $private$ : объекты будут доступны только коду внутри определения класса.

• $public$ : объекты будут доступны извне.

• $protected$ : объекты будут доступны оберткам вокруг этого класса и его наследникам(??забить пока что).

Обычно в $public$ пихают функции, в $private$ переменные.

Доступ к объектам класса производится через точку. Пример:

```
int main() {
    MyClass var;
    std :: cout << var.a;
    std :: cout << var.GetNumber();
```

}

Тут мы создали переменную *var* типа $MyClass$. Программа не скомпилируется, потому что мы хотим получить доступ к $MyClass$ :: $a$ извне, что запрещено, так как подпеременная $a$ находится в области *private*.

Однако, если стереть эту ошибочную строчку, то ошибок больше не будет. Функция $GetNumber$ в области *public* и она доступна, в ней написано *return a*;, но при этом действии переменная $a$ копируется, поэтому отдаваемое возвращаемое значение является лишь копией, которая не находится в области *private* у класса $MyClass$, поэтому никакой ошибки вообще не происходит.

**Задание 1.**

Сформулировать вопросы о том, что из вышеизложенного непонятно. Написать их в формате .doc с помощью Word, и добавить в svn.

**Задание 2.**

Написать класс комплексных чисел(комплексное число – это пара чисел $(a, b)$, где $a$ называется действительной частью, $b$ — мнимой). У него есть две скрытые переменные, и добавить в него функции $Im$ и $Re$, первая из которых возвращает мнимую часть, вторая — действительную.

Какого типа переменные $a$ и $b$? *int*, а может и *double*, а может и еще что-то. Попробовать сделать шаблонный класс!

**Конструктор и деструктор объекта.**

Кроме того, чтобы задать структуру объекта(то есть указать какие у его функций общие переменные в области *private* и какие тела у этих функций в области *public*), нужно так же указать, какие изначальные значения принимают эти общие переменные.

В задании 2 в итоге выводились два каких-то непонятных числа как раз по тому, что мы не задали их изначальные значения. Чтобы этого избежать, надо использовать конструктор.

Конструктор пишется как функция в области *public* с названием таким же, как название класса. Кроме всех остальных функций, существующих а языке, у конструктора есть список инициализации, который выглядит так

: $a_1(b_1), a_2(b_2), ..., a_n(b_n)$, где $a_i$ — переменные класса из области $private$(вообще говоря, неважно, из какой они области, просто все переменные договорились в области $private$ писать).

Пример:

```
class MyClass {
private :
    int a;
public :
    MyClass(const int value)
     : a(value)
    {}

    int GetNumber() const {
        return a;
    }
};
```

Использование конструктора выглядит так:

$MyClass\ var = MyClass(5);$

или, более правильная запись(чтобы не было копирования при написании знака =), такая

$MyClass\ var(5);$

Когда пишется $int\ a\ = 5$, на самом деле компилятор читает это так: $int\ a(5)$; и так более правильно писать.

В данном случае, конструктор принимает одну переменную, потому что одной переменной хватает для задания переменных класса. После этого с помощью списка инициализации он заставляет быть свой внутренней переменной $a$ равной этому значению $value$. Тело его пустое, потому что в данном случае нам хватило списка инициализации, чтобы задать все переменные класса. Могут быть такие случаи, когда списка инициализации недостаточно. Например, нужно выделить в конструкторе память с помощью команды $malloc$, ну тогда придется писать ее в теле конструктора.

Деструктор — это нечто, что вызывается всегда, когда объект разрушается. Объект разрушается, когда достигнута закрывающаяся фигурная скобка тела, в котором он был объявлен(например, если объект объявлен в начале функции main, то он разрушится в ее конце).

Деструктор пишется как функция, название которой начинается с символа ~ (тильда), а дальше продолжается как название класса.

В данном примере деструктор не нужен, потому что в классе только одна переменная $int\ a$. Деструктор нужен, например, в случаях, когда в конструкторе выделилась память командой $malloc$, а в конце нужно написать команду $free$ для освобождения памяти. Где ее писать? В деструкторе.

Пример:

```
class Array100 {
private:
    int * array;
public:
    Array100() {
        array = static_cast < int* > malloc(sizeof(int) * 100);
    }

    ~Array100() {
        free(array);
    }
};
```

В данном примере у конструктора нет списка инициализации, потому что он все равно не может выделить памяти для $array$ и нет других переменных, которые можно было бы проинициализировать. Инициализацию массива приходится делать в теле конструктора. А в теле деструктора приходится делать особождение выделенной памяти.

### Задание 3.

Найти, где символ тильды ~ расположен на клавиатуре.

### Задание 4.

Дописать в задание 2 конструкторы.

**Задание 5.**

Написать класс *Array*, внутри которого создается массив произвольного размера. Класс должен хранить этот размер. Размер должен передаваться в качестве принимаемой переменной в конструктор. А в деструкторе должна быть очистка памяти. Также в классе должны быть фунции, которые возвращают указатель на первую ячейку памяти, в которой хранится массив, и на ячейку за последней.

Нужно в функции *main* протестировать работу класса. Создать объект этого класса, и проверить, что если к результату функции, возвращающей начало, прибавить размер массива, то это будет равно результату функции, возвращающей ячейку за концом.

**Переопределение операторов.**

Все операторы в языке +, -, *, «, », <, >, <=, интерпретируется как функции, но не однозначным образом.

Строку $a + b$ можно интерпретировать как $operator + (a, b)$, где $operator+$ обычная функция, а можно интерпретировать как $a.opertor+ (b)$, где $operator+$ — функция внутри класса объекта $a$.

Пример:

```
class MyClass {
private:
    int a;
public:
    MyClass& operator+=(const MyClass& other) {
        this->a += other.a;
        return *this;
    }

    MyClass operator+(const MyClass& other) const {
        MyClass tmp(*this);
        tmp += other;
        return tmp;
    }
};
```

• В данном примере $operator+=$ это оператор, меняющий объект самого класса. Поэтому в его определении нет слова *const*. А вот $operator+$ не меняет ничего, он просто возвращает сумму данного объекта с объектом *other*.

• *this* — это обозначение для данного рассматриваемого объекта класса, оно действует только внутри определения класса. Это указатель типа $MyClass*$ на первую ячейку памяти, начиная с которой хранится объект класса $MyClass$(пока мы пишем определение, никакого объекта нету, но когда мы создадим объект $MyClass\ var$, то вместо *this* подставится адрес первой ячейки памяти, начиная с которой хранится *var*).

Когда мы пишем $*this$, мы просто разименовываем указатель.

• В данном коде использован еще один очень важный синтаксический прием. Строка вида $first->second$ интерпретируется компилятором как $(*first).second$, если $first$ — это указатель на объект класса, содержащий *second* в качестве своей внутренней функции или переменной.

То есть $this->a$ означает, что мы взяли переменную $a$ у рассматриваемого в данном контексте объекта класса переменную $a$. Можно было просто написать $a$.

• В операторе $+=$ есть строка $return * this$ специально, чтобы можно было делать следующее.

$var+=a+=b+=c...$

• В строке $MyClass\ tmp(*this)$ происходит копирование объекта $*this$. Это происходит с помощью вызова конструктора копирования, который в качестве аргумента принимает ссылку на объект того же самого класса.

Конструктор копирования сам дописывается компилятором, если его нет.

Копирование здесь нужно, потому что объект *this константный, и, поэтому, к нему нельзя применить оператор $+=$. Однако к его неконстантной копии *tmp* можно. Тогда с *tmp* выполнятся ровно те действия, которые написаны в теле $operator+=$, и все будет как надо. Объект $*this$ при этом, конечно же, не поменяется.

### Задание 6.

Дописать операторы сложения, вычитания, умножения, деления в класс комплексного числа. Дописать туда функции ввода с экрана и вывода на экран с помощью переопределения операторов $operator >>$ и $operator <<$.

**Задание 7.**

Дописать конструктор копирования в класс *Array*. Дело в том, что конструктор копирования, который допишет сам компилятор, будет копировать указатель *array*, в результате копия объекта будет работать с тем же самым участком памяти. При копировании для нового объекта нужно выделить память такого же размера и перенести туда массив поэлементно.

Дописать *operator* >> в этот класс.

Написать *operator*+ = и *operator*+, которые будут складывать поэлементно два массива одинаковой длины. Написать *operator*− = и *operator*− которые будут поэлементно вычитать один массив из другого.