# Лекция 5 – Классы в языке C++

Кафедра прикладной математики и информатики

13 апреля 2020 г.

# Определение класса

Данные, объявленные отдельно от операций над ними, предоставляют больше возможностей по использованию таких данных. Тем не менее более тесная связь между представлением данных и операцями над ними необходима для того, чтобы определенный пользователем тип имел все свойства, ожидаемые от типа, представляющего объект реального мира.

Часто нам надо сохранить представление недоступным для пользователей, чтобы упростить использование, гарантировать последовательное использование данных, сохраняя при этом возможноть улучшить представление. Для этого мы должны различать публично (**`public`**) доступные данные (который будет использоваться всеми) и скрытую (**`private`**) реализацию пользовательского типа (которая имеет доступ к недоступным в противном случае данным).

Языковой механизм для этого называется *классом*.

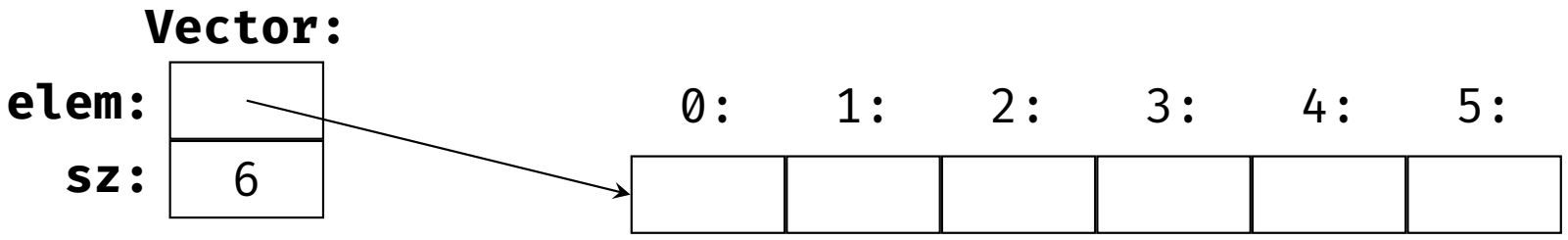
Например:

```
class Vector {
  public:
    // конструктор класса Vector
    Vector(int s) :elem{new double[s]}, sz{s} { }

    // деструктор класса Vector
    ~Vector(){ delete[] elem; }

    // доступ к элементу массива
    double& operator[](int i) { return elem[i]; }

    // размер вектора
    int size() { return sz; }

  private:
    double* elem; // указатель на элементы вектора
    int sz; // количество элементов
};
```

Имея такое определение, мы можем объявить переменную типа `Vector`:

```
Vector v(6);
```

# Представление объектов в памяти




Объект вектора представляет собой обертку, содержащую указатель на элементы (`elem`) и количество элементов (`sz`). Количество элементов (6 в нашем случае) может варьироватся от одного объекта типа `Vector` к другому объекту типа `Vector`.

Однако, сам объект типа `Vector` имеет фиксированный размер.
Это базовая техника для обработки переменного объема информации в C++: обработчик фиксированного размера ссылается на изменяющееся количество информации ”где-то еще” (например аллоцированную через оператор **`new`**).

# Использование публичного интерфейса

Ниже представлено использование приватных полей класса `Vector` через публичный члены класса `Vector()`, **`operator`**`[]()` и `size()`.

```
double read_and_sum(int s)
{
    Vector v(s); // создаем вектор s элементов
    for (int i=0; i!=v.size(); ++i)
        cin>>v[i]; // читаем содержимое из консоли
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=v[i]; // считаем сумму элементов
    return sum;
}
```

”Функция” член с тем же самым именем что и класс называется *конструктором*, которая используется для создания объектов класса.

`Vector(`**`int`**`)` определяет, каким образом конструируется объект типа `Vector`. В частности требуется аргумент типа int для того, чтобы создать объект. Это целое число используется как количество элементов вектора.

Конструктор инициализиурет члены класса `Vector` при помощи списка инициализации: `:elem{new double[s]}, sz{s}`

Таким образом мы инициализируем переменную член `elem` указателем на `s` элементов типа **`double`** полученных из свободного пулла памяти операционной системы. Затем мы инициализируем `sz` числом `s`.

Доступ к элементам предоставляется через квадратные скобки, аналогично массиву встроенному в язык C++. Определение такого доступа соответствует **`operator`**`[]`, которое предоставлет доступ как на чтение, так и на запись значений вектора.

Функция `size()` возвращает количество элементов в векторе.

”Функция” `~Vector()` называется *деструктором* класса. Она предназначена для возвращения аллоцированной памяти обратно в пулл операционной системы. Деструктор вызывается неявно при вытеснении переменной `v` со стека.

# Характеристика базового класса

Основная идея базового класса состоит в том, что его поведение ”такое же как и у встроенных типов данных”. Например тип комплексных чисел ведет себя так же как и встроенный тип `int`, за исключением, конечно, их собственной семантики и набора операций. Также `vector` или `string` похожы на встроенные массивы, только с лучшим поведением. Основная характеристика базового класса состоит в том, что представление данных (representation) класса есть часть его определения. Во множестве случаев, таких как `vector` представление данных в виде одного или нескольких указателей на данные хранящиеся ”где-то еще”. Но это представление данных существует в каждом объекте базового класса. Это позволяет создавать реализации эффективные по скорости и используемой памяти. В частности это дает:

- размещать объекты базового класса на стеке, в статической памяти и других объектах;
- обращаться к объету напрямую (не просто через указатели и ссылки);
- инициализировать объекты сразу и полностью (например используя конструкторы);
- копировать и перемещать объекты.

# Пример арифметического типа

```
class complex {
    double re, im; // representation: два числа double
    public:
    complex(double r, double i) :re{r}, im{i} {} // конструктор двух
    ↪  аргументов
    complex(double r) :re{r}, im{0} {} // конструктор одного аргумента
    complex() :re{0}, im{0} {} // конструктор по умолчанию complex:
    ↪  {0,0}
    double real() const { return re; }
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }
    complex& operator+=(complex z)
    {
        re+=z.re; // сумма по определению комплексных чисел
        im+=z.im;
        return *this; // возвращение результата
    }
    complex& operator-=(complex z)
    {
        re-=z.re;
        im-=z.im;
        return *this;
    }
    complex& operator*=(complex); // определено вне класса
    complex& operator/=(complex); // определено вне класса
};
```

Приведенная ранее реализация комплексных чисел представляет собой упрощенную версию класса `complex` из стандартной библиотеки.

Определение класса содержит только те операции, которым необходим доступ к представлению данных класса. Представление скрыто за приватным модификатором доступа. Само по себе представление данных простое и понятное: два числа типа **`double`** представляющие действительную и мнимую часть комплексного числа.

По умолчанию функции члены, определенные внутри класса, имеют модификатор **`inline`**, который объявлен неявно. Этот модификатор позволяет компилятору избавиться от функционального вызова и напрямую генерировать байт код для соответствующей операции.

Конструктор, который может быть вызван без аргументов называется *конструктором по умолчанию*. Таким образом `complex()` – конструктор по умолчанию класса комплексных чисел. При объявлении конструктора по умолчанию избегается ситуация создания не инициализированных переменных данного типа.

# Const

Квалификатор **const**, примененный к функциям возвращающим действительную и мнимую часть комплексного числа, сообщает о том, что функция не изменяет состояние объекта, для которого она вызвана.

Константная функция член может быть вызвана для константных и неконстантных объектов, но неконстантная функция член может быть вызвана только для неконстантных объектов. Например:

```
complex z = {1,0};
const complex cz {1,3};
z =cz; //OK: assigning to a non-const var iable
cz = z; // error : complex::operator=() is a non-const
 ↪  member function
double x = z.real(); // OK: complex::real() is a const
 ↪  member function
```

# Операторы

Множество полезных операторов не требуют прямого доступа к представлению данных класса `complex`, таким образом их можно определить вне класса:

```
complex operator+(complex a, complex b)
{ return a+=b; }
complex operator-(complex a, complex b)
{ return a-=b; }
complex operator-(complex a) // унарный минус
{ return {-a.real(), -a.imag()}; }
complex operator*(complex a, complex b)
{ return a*=b; }
complex operator/(complex a, complex b)
{ return a/=b; }
```

Здесь использован факт, что аргументы передаваемые по значению копируются, таким образом изменение локальной переменной a не влечет изменения оригинальной переменной, с которой вызвана функция.

Определение операторов == и != весьма прямолинейно:

```
bool operator==(complex a, complex b) // equal
{
    return a.real()==b.real() && a.imag()==b.imag();
}
bool operator!=(complex a, complex b) // not equal
{
    return !(a==b);
}
```

Использование класса может быть таким:

```
void f(complex z)
{
    complex a {2.3}; // конструируем {2.3,0.0} из 2.3
    complex b {1/a};
    complex c {a+z*complex{1,2.3}};
    // ...
    if (c != b)
        c = -(b/a)+2*b;
    // ...
}
```

# Перегрузка операторов

Компилятор преобразует операторы с участием комплексных чисел в корректные функциональные вызовы. Например, `c!=b` означает `operator!=(c,b)` и `1/a` означает `operator/(complex{1},a)`

Определенные пользователем операторы (”перегруженные операторы”) должны быть реализованы с осторожностью и умом. Синтакс данных операторов фиксирован языком, так что программист не может определить унарный оператор `/`. Также, невозможно переопределить значение оператора для встроенного типа, тоесть нельзя переопределить `+` для вычитания двух значений типа `int`.

# Советы I

Изучай:
https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

1. Выражай идеи непосредственно в коде.
2. Базовый класс – простейшая форма класса. Когда возможно используй базовый класс вместо более сложных классов и вместо простых структур данных.
3. Используй базовые классы для представления простейших концептов.
4. Используй базовые классы вместо иерархий классов для критических компонентов требующих высокой производительности.
5. Определяй конструкторы для инициализации объекта.
6. Создавай функцию член только тогда, когда ей требуется непосредственный доступ к представлению данных класса.
7. Определяй операторы преимущественно для имитации их стандартного использования.
8. Используй свободные функции для определения симметричных операторов.

# Советы II

9. Объявляй функции члены константными, если они не изменяют состояния объекта.
10. Если конструктор получает ресурс, этот класс нуждается в деструкторе, который этот ресурс освобождает.
11. Избегай ”голых” операторов **`new`** и **`delete`**.
12. Используй обертки для того чтобы управлять ресурсом.