Лекция 6 – Классы в языке C++

Кафедра прикладной математики и информатики

28 апреля 2020 г.

Класс контейнер I

Определение

Контейнером называется объект содержащий коллекцию из нескольких элементов.

Мы называем класс `Vector` контейнером, потому что он предназначен для хранения коллекции нескольких элементов.
Пара функций конструктор и деструктор позволяет создавать коллекции неизвестного на момент исполнения размера.
Техника получения ресурсов в конструкторе и освобождения их в деструкторе, известная как Resource Acquisition Is Initialization или RAII, позволяет нам исключить «голые» операции **`new`**, то есть избежать выделения памяти в общем коде и сохранить такие выделяния скрытыми внутри реализации хороших поведенческих абстракций. Точно так же следует избегать «голой» операции **`delete`**. Скрытие абстракцией **`new`** и **`delete`** делает код намного менее подверженным ошибкам и намного легче избежать утечек ресурсов.

Класс контейнер II

В предыдущих лекция мы уже видели пример класса контейнера:

```
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
};
```

Инициализация данных в контейнере

Контейнер существует для хранения элементов, поэтому, очевидно, нам нужны удобные способы доставки элементов в контейнер. Мы можем создать вектор с соответствующим количеством элементов, а затем назначить их, но обычно другие способы более элегантны.
Например:

- список инициализации
- `push_back()`

```
class Vector {
    public:
    Vector(std::initializer_list<double>); // инициализация
    ↪  списком double
    // ...
    void push_back(double); // добавляет новый элемент в конец
    // ...
};
```

Список инициализации

`std::initalizer_list`, используемый для определения конструктора initializer-list, является типом стандартной библиотеки, известным компилятору: когда мы используем как список, такой как 1,2,3,4, компилятор создаст объект типа `std::initalizer_list` для передачи программе. Итак, мы можем написать:

```
Vector v1 = {1, 2, 3, 4, 5}; // v1 имеет 5 элементов
Vector v2 = {1.2, 3.4, 6, 8}; // v2 имеет 4 элемента
```

Конструктор списка инициализации вектора может быть определен так:

```
Vector::Vector(std::initializer_list<double> lst) //
↪  инициализация списком
:elem{new double[lst.siz e()]},
↪  sz{static_cast<int>(lst.size())}
{
    copy(lst.begin(),lst.end(),elem); // копирование из списка
    ↪  в elem
}
```

push_back()

Функция push_back() полезна для ввода произвольного числа элементов. Например:

```
Vector read(istream& is)
{
    Vector v;
    for (double d; is>>d; ) // читаем числа с двойной
    ↪  точностью во временную переменную d
    v.push_back(d); // add d to v
    return v;
}
```

Цикл ввода завершается из-за конца файла или ошибки форматирования. До того, как это случится, каждое прочитанное число добавляется в Vector, а именно добавляется в конец. Размер вектора v равняется количеству прочитанных элементов.

Абстрактные типы

Такие типы, как `complex` и `Vector`, называются *базовыми* типами, потому что их представление является частью их определения. В этом они похожи на встроенные типы. Напротив, *абстрактный* тип – это тип, который полностью изолирует пользователя от деталей реализации. Для этого мы отделяем интерфейс от представления и оставляем подлинные локальные переменные. Поскольку мы ничего не знаем о представлении абстрактного типа (даже о его размере), мы должны разместить объекты в свободном хранилище и получить к ним доступ через ссылки или указатели.

Абстрактный тип Container I

Сначала мы определим интерфейс класса Container, который мы разработаем как более абстрактную версию нашего класса Vector:

```
class Container {
    public:
    virtual double& operator[](int) = 0; // чистая виртуальная
    ↪  функция
    virtual int size() const = 0; // константная функция член
    virtual ~Container() {} // деструктор
};
```

Этот класс является чистым интерфейсом для определенных контейнеров, определенных позже. Слово vitrual означает «может быть позже переопределено в классе, производном от него». Неудивительно, что функция, объявленная виртуальной, называется виртуальной функцией. Класс, производный от Container, обеспечивает реализацию интерфейса Container.

Синтаксис = 0 говорит, что функция чисто виртуальная; то есть некоторый класс, производный от Container, должен определять функцию.

Абстрактный тип Container II

Тем не менее, невозможно определить объект типа Container. Например

```
Container c; // error : there can be no objects of an abstract
↪  class
Container* p = new Vector_container(10); // OK: Container --
↪  интерфейс
```

Контейнер может служить только интерфейсом для класса, который реализует свои функции **operator**[]() и size() Класс с чисто виртуальной функцией называется абстрактным классом.

Класс контейнер может быть использован следующим образом

```
void use(Container& c)
{
    const int sz = c.size();
    for (int i=0; i!=sz; ++i)
    cout << c[i] << '\n';
}
```

Абстрактный тип `Container` III

Обратите внимание, как `use()` использует интерфейс `Container` при полном незнании деталей реализации. Он использует `size()` и `[]`, не имея представления о том, какой именно тип обеспечивает их реализацию. Класс, который обеспечивает интерфейс для множества других классов, часто называют полиморфным типом.

Как правило для абстрактных классов, контейнер не имеет конструктора. Ведь у него нет данных для инициализации. С другой стороны, контейнер имеет деструктор и этот деструктор является виртуальным, так что классы, производные от `Container`, могут предоставлять реализации. Опять же, это характерно для абстрактных классов, потому что ими, как правило, манипулируют с помощью ссылок или указателей, а тот, кто уничтожает `Container` с помощью указателя, не знает, какие ресурсы принадлежат его реализации.

Реализация интерфейса класса Container I

Абстрактный класс Container определяет только интерфейс и никакой реализации. Чтобы контейнер был полезен, мы должны реализовать контейнер, который реализует функции, требуемые его интерфейсом. Для этого мы можем использовать конкретный класс Vector:

```
class Vector_container : public Container { //
↪  Vector_container реализует интерфейс Container
  public:
    Vector_container(int s) : v(s) { } // Vector вектор s
    ↪  элементов
    ~Vector_container() {}
    double& operator[](int i) override { return v[i]; }
    int size() const override { return v.size(); }
    private:
    Vector v;
};
```

Реализация интерфейса класса `Container` II

Выражение `:public` может читаться как «является производным от» или «является подтипом». Класс `Vector_container` называется производным от класса `Container`, а класс `Container` считается базовым классом класса `Vector_container`. Альтернативная терминология: подкласс и суперкласс `Vector_container` и `Container` соответственно. Говорят, что производный класс наследует членов от своего базового класса, поэтому использование базовых и производных классов обычно называют наследованием.

Считается, что члены **`operator`**`[]()` и `size()` переопределяют соответствующие члены в базовом классе `Container`. Здесь используется явное переопределение (**`override`**), чтобы явно указать намерение. Использование **`override`** является необязательным, но будучи явным, компилятор может отлавливать ошибки, такие как неправильное написание имен функций или небольшие различия между типом виртуальной функции и ее предполагаемым переопределением. Явное использование переопределения особенно полезно в больших классах, где иначе сложно понять, что должно быть переопределено.

Деструктор (`~Vector_container()`) переопределяет деструктор базового класса (`~Container()`). Обратите внимание, что деструктор-член `~Vector()` неявно вызывается деструктором своего класса (`~Vector_container()`).

Реализация интерфейса класса `Container` III

Для функции `use(Container &)` использующей контейнер в полном незнании деталей реализации, некоторые другие функции должны будут создать объект, с которым она может работать. Например:

```
void g()
{
    Vector_container vc(10); // Vector 10 элементов
    // ... инициализация данных ...
    use(vc);
}
```

Реализация интерфейса класса Container IV

Поскольку функция use() не знает о Vector_containers, а знает только интерфейс Container, она будет работать так же хорошо для другой реализации контейнера. Например:

```
class List_container : public Container { // List_container
↪  implements Container
  public:
    List_container() { } // пустой список
    List_container(initializer_list<double> il) : ld{il} { }
    ~List_container() {}
    double& operator[](int i) override;
    int size() const override { return ld.size(); }
  private:
    std::list<double> ld; // (standard-library) список double
};
double& List_container::operator[](int i)
    {
    for (auto& x : ld) {
        if (i==0)
            return x;
        --i;
    }
    throw out_of_range{"List container"};
}
```

Реализация интерфейса класса `Container` V

Здесь представление данных – список стандартной библиотеки параметрзованный типом `<double>`. Обычно не стоит реализовывать контейнер с операцией произвольного доступа используя список, потому что производительность такого доступа по списку ужасна по сравнению с произвольным доступом у вектора. Однако здесь просто показана реализация, которая радикально отличается от обычной.

Некоторая функция может создать `List_container` и затем использовать его при помощи функции `use()`:

```
void h()
{
    List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    use(lc);
}
```

Реализация интерфейса класса `Container` VI

Идея в том, что `use(Container &)` понятия не имеет, является ли его аргумент `Vector_container` , `List_container` или каким-либо другим видом контейнера; этого не нужно знать. Функция может использовать любой вид контейнера. Она знает только интерфейс, определенный контейнером. Следовательно, `use(Container &)` не нужно перекомпилировать, если используется реализация `List_container` или используется совершенно новый класс, производный от `Container`.

Обратная сторона этой гибкости заключается в том, что объектами нужно манипулировать с помощью указателей или ссылок.

Виртуальные функции I

Рассмотрим снова использование контейнера:

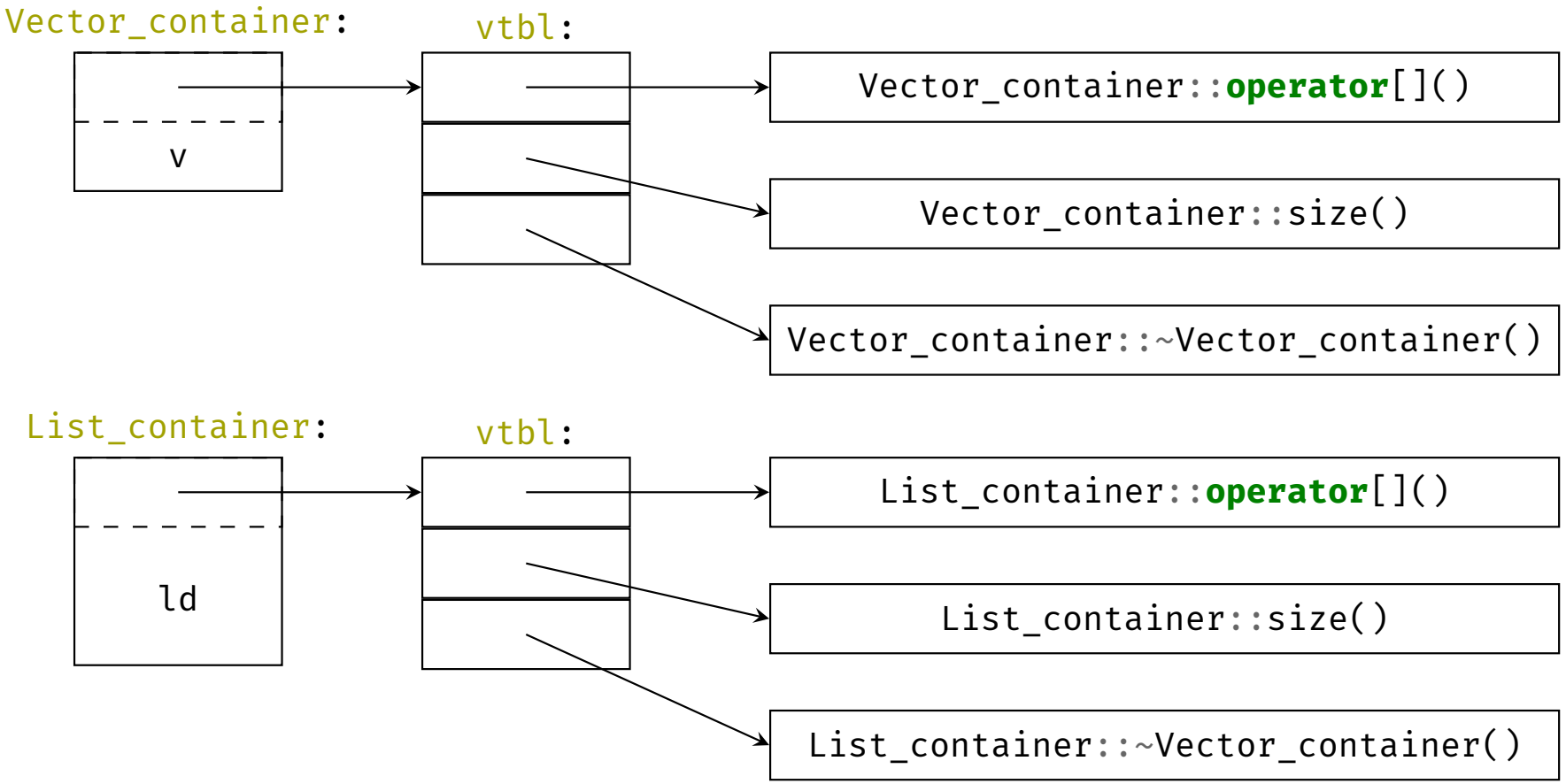
```
void use(Container& c)
{
    const int sz = c.size();
    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

Как разрешается вызов `c[i]` в функции `use(Container &)` до правильного оператора **`operator`**`[]()`? Когда `h()` вызывает `use()`, должен быть вызван оператор **`operator`**`[]()`, который реализован в `List_container`. Когда `g()` вызывает `use()`, должен быть вызван оператор **`operator`**`[]()`, который реализован в `Vector_container`.

Для достижения этого разрешения объект-контейнер должен содержать информацию, позволяющую ему выбрать правильную функцию для вызова во время выполнения. Обычный метод реализации заключается в том, что компилятор преобразует имя виртуальной функции в индекс таблицы указателей на функции. Эта таблица обычно называется таблицей виртуальных функций или просто `vtbl`.

Виртуальные функции II

Каждый класс с виртуальными функциями имеет свою собственную `vtbl`, регистрирующую его виртуальные функции. Это может быть представлено графически следующим образом:



Виртуальные функции III

Функции в `vtbl` позволяют правильно использовать объект, даже если размер объекта и расположение его данных неизвестны вызывающей стороне. Реализация вызывающей стороны должна знать только местоположение указателя на `vtbl` в `Container` и индекс, используемый для каждой виртуальной функции. Этот механизм виртуального вызова можно сделать почти таким же эффективным, как механизм «обычного вызова функции» (в пределах 25%). Его служебная память занимает один указатель в каждом объекте класса с виртуальными функциями плюс одна `vtbl` для каждого такого класса.