РАЗБОР ДОРЕШКИ 5

Операторы присваивания
SharedBuffer и LazyString

Если мы делаем корректный буфер, то LazyString - это просто надстройка над ним. И тогда не надо писать присваивания. (Конструктор LazyString писать всё же придётся).

Возвращение ссылки на себя:

- позволяет писать цепочки присваиваний

```
x = y = z = src;
x = (y = (z = src));
```

- позволяет присваивать и проверять

```
if ((x = src).is_okay()) .....
```

Две стратегии присваивания:

- очистка и перезапись
- copy and swap

Очистка и перезапись

```
T& operator = (const T& src) {  // найдите проблему
    dec_ref();
    data_ = src.data_;
    inc_ref();
    return *this;
}

T& operator = (T&& src) {
    dec_ref();
    data_ = src.data_;
    src.data_ = nullptr;
    return *this;
}
```

Не надо использовать в роли dec_ref() деструктор!

```
    this->~T();
```

copy and swap

```
T& operator = (const T& src) {
    T(src).swap(*this);
    return *this;
}
T& operator = (T&& src) noexcept {
    src.swap(*this);
    return *this;
}
void swap(T& other) noexcept {
    std::swap(data_, other.data_);
}
```

Инкремент/декремент ссылок теперь только в конструкторе копирования / деструкторе.

Проверка на самоприсваивание

```
T& operator = (const T& src) {
    if (&src != this) {
        .....
    }
    return *this;
}
```

В конструкторе такая проверка не нужна. (Разве что мы боремся против хаков)

```
T tmp(tmp);
```

Конструкторы пустых объектов

Пустые объекты встречаются чаще всего. Особенно, у типов с семантикой значения. Поэтому крайне желательно делать их легковесными.

- поддержка нулевого содержимого
- паттерн Null Object

Нулевое содержимое

```
class SharedBuffer {
    size_t* ref_count_ = nullptr;
    char* buffer_ = nullptr;
    size_t size_ = 0;
public:
    SharedBuffer() = default;
    SharedBuffer(size_t size) { if (size > 0) {.....} }
    .....
};
```

- плюсы: очень дешёвый
- минусы: требуются проверки на nullptr

Null Object

Это полноценный объект, возможно, синглетон, которым все пользуются

```
static SharedBuffer empty_string_buf(1);

static SharedBuffer makeBuf(const char* s) {
    size_t len = strlen(s);
    if (len == 0) return empty_string_buf;
    SharedBuffer buf(len + 1); strcpy(buf.getData(), s);
    return buf;
}

class LazyString {
    SharedBuffer buf_;
public:
    LazyString(const char* s) : buf_(makeBuf(s)) {}
};
```

- плюсы: однородный код
- минусы: в общем случае нужно гарантировать, что мы не перепишем сам синглетон

Операции с пустыми операндами

```
LazyString empty("");

empty + right == right
left + empty == left

left += empty;
empty += right;
```

Сравнение строк

Что из перечисленного нужно, а что нет?

- `this == &other`
- `getData() == other.getData()`
- `getSize() == other.getSize()`
- `*getData() == *other.getData()`
- `strcmp(getData(), other.getData())` - ничего здесь не забыли?

Прокси-ссылка на символ

```
class LazyString {
    .....
    char operator[](size_t i) const;
    CharWrapper operator[](size_t i) { return CharWrappe
    .....
};

class CharWrapper {
public:
    operator char() const;
    char& operator = (char v) const;
    char& operator ++ () const;
    char operator ++ (int) const;
};

void ensureUnique(LazyString& s); // постусловие - s.use
```

- где лучше разместить функцию ensureUnique?

read / write

- си-строка: массив символов, в котором ровно один раз встречается ОСОБОЕ значение (`'\0'`)
- вектор чисел: произвольный массив, особых значений нет

Два способа записывать массивы произвольных символов/чисел/чего угодно

- префикс длины
- escape-последовательности ({0, 0} - конец, {0, 1} - просто 0)
- часто используется со строками: `"\n"` / `"\\"`

Описание задач

В задачах 1-3 будем реализовывать свою библиотеку полиморфных стримов. В исследовательской задаче 4 поизучаем как ведет себя выравнивание и размер структуры в зависимости от её состава.

Имена файлов (sic!), в которых следует реализовывать решение, определите, исходя из CMakeLists и прилагаемых тестов.

Также поймите, в каком пространстве имён должна быть реализована функциональность.

Есть базовый класс `my_ostream` с виртуальными методами. Есть несколько наследников `my_ostream`. Стримы умеют выводить в поток три типа: `int`, `double`, `string`.

Пример кода

```
my_ostream_con cout;                          // console outpu
my_ostream_file cerr("errors.txt");           // file output
my_ostream_db dbout(...);                     // db output

int main() {
  std::vector<my_ostream*> stream_ptrs = {&cout, &cerr,
  for (auto stream_ptr : stream_ptrs)
    (*stream_ptr) << "Hello world on console!" << "\n"
                  << 1 << 2 << 3 << 4 << "\n"
                  << 777.777 <<  "\n";

  return 0;
}
```

Задача #1 (1 балл)

Реализовать базовый класс `my_ostream`, одного наследника `my_ostream_con` для вывода данных в консоль.

Должен поддерживаться вывод: `int`, `double`, `std::string`.

Задача #2 (1 балл)

Реализовать наследника `my_ostream_file`, который выводит данные в файл. Попробуйте использовать стримы полиморфно, как на слайде предисловия (положить указатели в вектор и записать что-нибудь одновременно в два стрима).

Работа с файлом должна быть корректной (файл должен быть закрыт после того, как объект `my_ostream_file` уничтожен)

Задача #3 (1 балл)

Реализовать наследника `my_ostream_combo`, который принимает в конструкторе два стрима и комбинирует их. Т.е. при записи в такой стрим, информация записывается во оба переданных ему в конструктор стрима.

Задача #4* (2 балла)

Пишем sizeof!

Хотим реализовать функцию `size_t GetSize(const std::string& struct)` возвращающую то же, что вернет `sizeof()` описанной структуры в c++.

- поля типов char, short, int, float, double (1 балл)
- возможно спецификаторы полей `alignas(n)`, где n - степень двойки (+ 1 балл)

В данном случае мы делаем следующие допущения

- Решение будет не портабельно, ибо паддинги и выравнивания не строго специфицированы по стандарту. Мы миримся с этим: на стаданртных свежих компиляторах gcc/clang ожидаем одинаковое поведение.
- Полагаем в данной задаче, что у нас архитектура x64, размеры char, short, int, long соответственно 1, 2, 4, 8 байт

Пример строки с описанием структуры

```
struct A {
  int x;
  char y;
  alignas(16) char z;
};
```

Парсинг строки рекомендуется делать с помощью <regex>

Про выравнивание и паддинг можно почитать тут

Пояснения.

Гарантируется, что

- структура всегда имеет имя `A`
- каждое поле определяется на отдельной строке, без лишних пробелов
- каждое определение начинается с двух пробелов и заканчивается точкой с запятой (см. пример)
- поля в структуре не инициализируются
- имеется всегда только одна пара фигурных скобок - на первой и последней строке определения структуры