

Углублённое программирование на C++

Структуры и классы. Часть I

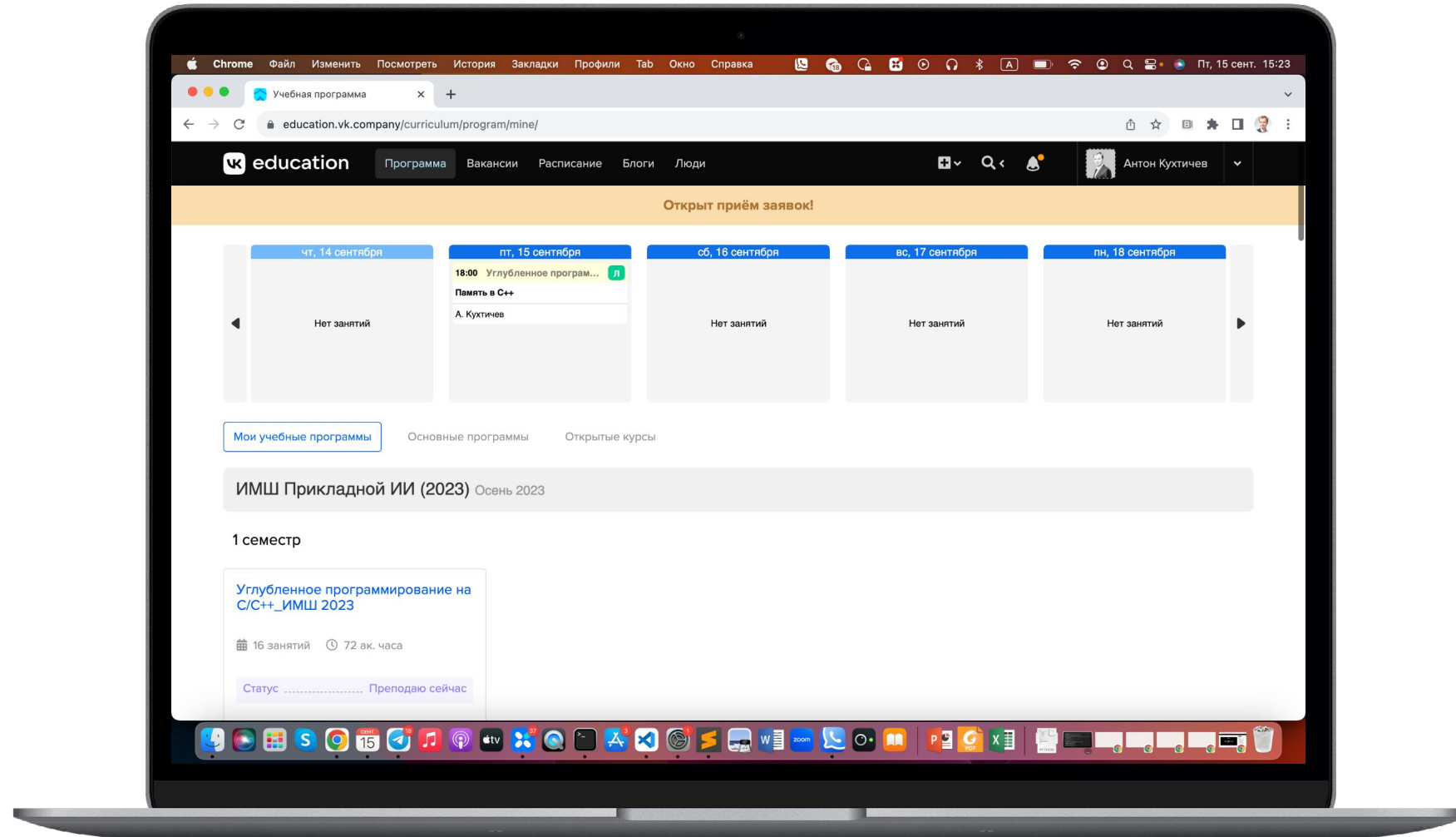
Кухтичев Антон



24 октября 2024 года

Напоминание отметиться на портале

и оставить отзыв
после лекции



Содержание занятия

- Квиз #3
- Пользовательские типы
- Классы и структуры
- Модификаторы доступа
- RAII (Resource Acquire Is Initialization)
- Константные методы
- Наследование
- Перегрузка методов
- Виртуальные функции
- Операторы

Мем
недели

Когда программист на плюсах
впервые увидел код Python:



КВИЗ #3



Минутка юмора



Язык программирования C++. Стандарт C++11.

Краткий курс

64 Глава 4

```
class X {  
public:  
    X(Sometype);           // обычный конструктор: создает объект  
    X();                   // умолчательный конструктор  
    X(const X&);           // копирующий конструктор  
    X(X&&);                // перемещающий конструктор  
    X& operator=(const X&); // копирующее присваивание  
    X& operator=(X&&);     // перемещающее присваивание  
    ~X();                 // деструктор  
    // ...  
};
```

Существует пять ситуаций, когда объект должен копироваться или перемещаться:

- Он является источником для присваивания



Пользовательские ТИПЫ



Пользовательские типы

- Перечисления

```
enum class Exception {};
```

- Классы

```
class ClassName {};
```

```
struct ClassName {};
```

- Объединения

```
union VariantName {};
```



Классы и структуры



Понятие класса

- Объектно-ориентированное программирование построено на понятие класса;
- Объявление класса начинается с ключевого слова `class`;
- По умолчанию члены класса являются закрытыми (`private`-членами);
- Классы и структуры это родственные типы;
- Объект — сущность в адресном пространстве компьютера, появляющаяся при создании класса;
- По определению структура есть класс, все члены которого по умолчанию являются открытыми;

Конструктор

- Конструктор — это функция, которая вызывается при создании объекта;
- Конструктор вызывается автоматически при создании объекта при помощи `new` (но не при помощи `malloc`!);
- Если конструктор не написан явно, C++ гарантирует, что будет создан конструктор по умолчанию;
- Не возвращает тип;

Деструктор

- Деструктор — это функция, которая вызывается при разрушении объекта;
- Если деструктор не написан явно, C++ гарантирует, что будет создан деструктор по умолчанию;
- Не возвращает тип;

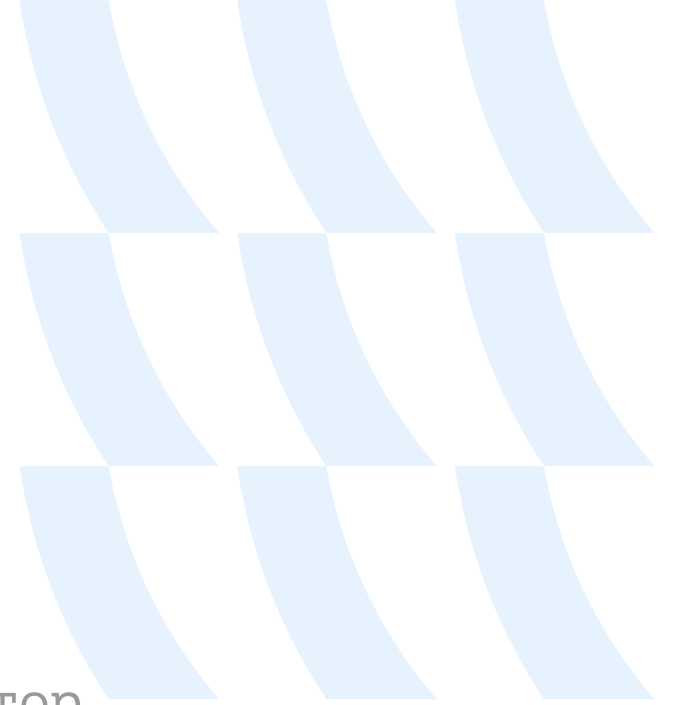
Модификатора доступа

```
class A
{
public:
    int x_; // доступно всем;
protected:
    int y_; // доступно не только лишь всем;
            // только внутри класса и наследникам;
private:
    int z_; // мало кому доступно; доступно только внутри класса;
};
```



Пример класса

```
class Dachshund
{
public:
    Dachshund(uint8_t age) { age_ = age; } // конструктор
    ~Dachshund() {} // деструктор
private:
    uint8_t age_ = 0; // закрытый член класса
};
```



Специальные функции-члены

- В C++98 включает четыре такие функции:
 - конструктор по умолчанию
 - деструктор
 - копирующий конструктор
 - оператор копирующего присваивания
- Эти функции создаются, только если они необходимы, т.е. если некоторый код использует их без их явного объявления в классе;
- Конструктор по умолчанию генерируется только в том случае, если в классе не объявлен ни один конструктор.



1. Скотт Мейерс. Эффективное использование C++. Правило 5. Какие функции C++ создаёт и вызывает молча.

Специальные функции-члены

- В C++11 приняты два новых игрока:
 - перемещающий конструктор;
 - оператор перемещающего присваивания;

Подробнее о них на следующей лекции.



1. Скотт Мейерс. Эффективный и современный C++. Пункт 3.11
Генерация специальных функций-членов

Специальные функции-члены

Если поведение сгенерированных компилятором функций вас устраивает (т.е. почленное копирование нестатических членов-данных), то можно сказать компилятору это:

```
class Seed {  
public:  
    ...  
    ~Seed(); // пользовательский деструктор  
    ...  
    // Поведение копирующего конструктора по умолчанию правильное!  
    Seed(const Seed&) = default ;  
};
```



1. Скотт Мейерс. Эффективный и современный C++. Пункт 3.11
Генерация специальных функций-членов

Какие методы генерирует компилятор при наличии различных входных данных

Если явно объявить

Вы получите результат

	Ничего	Деструктор	Конструктор копирования	Присваивание копии	Конструктор перемещения	Присваивание перемещения
Деструктор ~Foo()	✓	✓	✓	✓	✓	✓
Конструктор копирования Foo(const Foo&)	✓	✓	✓	✓		
Присваивание копии Foo& operator=(const Foo &)	✓	✓	✓	✓		
Конструктор перемещения Foo(Foo &&)	✓		Вместо переноса используется копирование		✓	
Присваивание переноса Foo& operator=(Foo &&)	✓					✓



1. Джош Лоспинозо. С++ для профи. Глава 4. Жизненный цикл объекта. Методы, генерируемые компилятором. (стр. 192).

Ссылка на себя

- Каждая (нестатическая) функция-член знает, для какого объекта она вызвана, и **может явно** на него ссылаться при помощи `this`;
- `this` является указателем на объект, для которого вызвана функция;

```
struct A
{
    int x_ = 0;
    void foo([A *this]) {
        this->x_ += 10;
        x_ += 10;
    }
};
```

RAII (Resource Acquire Is Initialization)

- Захват ресурса есть инициализация.
- В конструкторе объект получает доступ к какому либо ресурсу (например, открывается файл), а при вызове деструктора этот ресурс освобождается (закрывается файл).
- Можно использовать не только для управления ресурсами;
- Класс инкапсулирует владение (захват и освобождение) некоторого ресурса;

RAII (Resource Acquire Is Initialization)

```
struct Profiler
{
    Profiler() {
        // получаем текущее время
    }
    ~Profiler() {
        // сохраняем время между вызовами конструктора и деструктора
    }
};

void someFunction()
{
    Profiler prof;
    // ...
}
```



Как узнать имя функции внутри профайлера?

```
struct Profiler
{
    Profiler(const std::string &func_name) {
        func_name_ = func_name;
    }
    std::string func_name_;
};

void someFunction()
{
    Profiler prof(__func__);
    // ...
}
```

Как узнать имя функции внутри профайлера C++20?

```
#include <source_location>
```

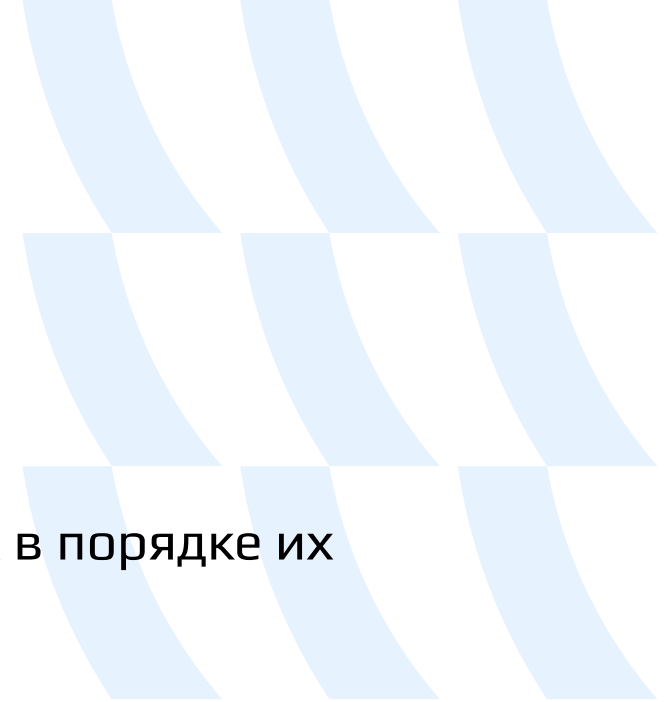
```
struct Profiler
```

```
{  
    Profiler(const std::source_location &location =  
              std::source_location::current()) {  
        location_ = location;  
    }  
    std::source_location location_;  
};
```

```
void someFunction()  
{  
    Profiler prof();  
    // ...  
}
```

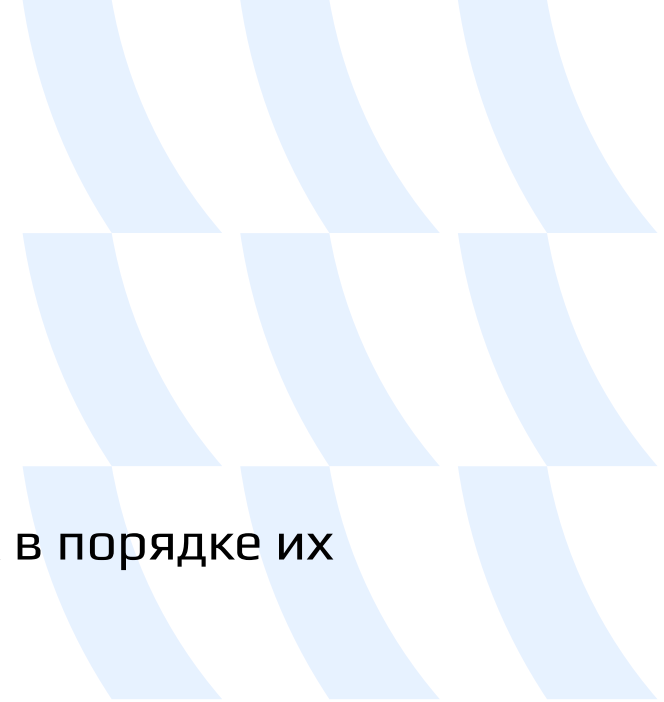

Конструирование объекта

1. Выделяется память под объект;
2. Если есть базовые классы, то конструирование начинается с них в порядке их очередности в списке наследования;
3. Инициализируются поля класса в том порядке, в котором они объявлены в классе;
4. Происходит вызов конструктора.



Уничтожение объекта

1. Выделяется память под объект;
2. Если есть базовые классы, то конструирование начинается с них в порядке их очередности в списке наследования;
3. Инициализируются поля класса в том порядке, в котором они объявлены в классе;
4. Происходит вызов конструктора.



Списки инициализаторов членов

1. Все инициализации члена выполняются перед телом конструктора;
 - а. Обеспечивает правильность всех элементов перед выполнением конструктора;
2. Нужно упорядочить инициализаторы членов в том же порядке, в котором они указаны в определении класса, поскольку их конструкторы будут вызываться в этом же порядке;

Константные методы



Константные методы

- Любые методы кроме конструктора и деструктора могут быть константными.
- Метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект).
- Константный метод можно вызывать как для константного, так и для неконстантного объекта, в то время как неконстантный метод можно вызвать только для объекта, не являющегося константой;
- **Рекомендация:** делайте все ваши методы, которые не изменяют данные объекта класса, константными.

mutable

- Позволяет изменять члены класса внутри `const`-методов;



Перегрузка методов

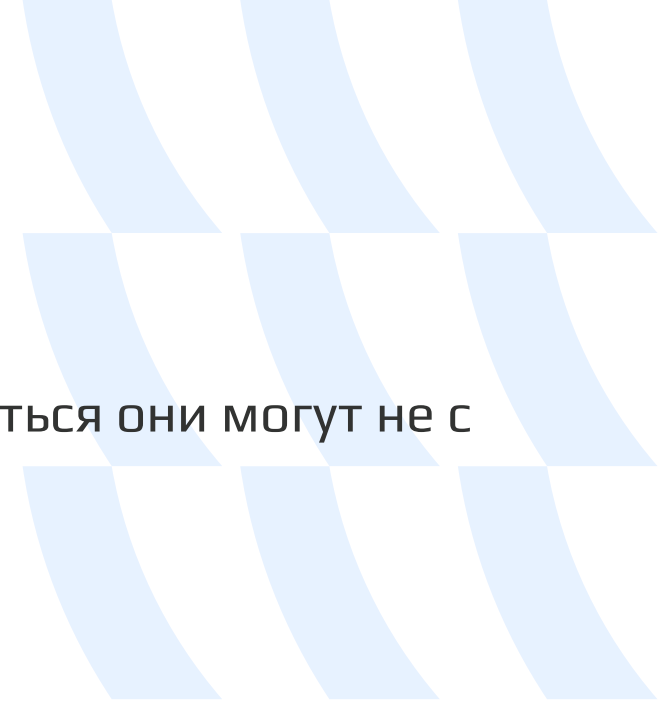


Перегрузка методов

- Методы классов - это просто функции, в которые неявно передается указатель на сам класс;
- Конструкторы - это тоже функции и их тоже можно перегружать.
- Деструкторы - тоже функции, но перегружать нельзя.

Параметры по умолчанию

- Пропусков в параметрах по умолчанию быть не должно, начинаться они могут не с первого аргумента, но заканчиваться должны на последнем.



Наследование



Наследование

- Возможность порождать класс на основе другого с сохранением всех свойств класса-предка.
- Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс – потомком, наследником, дочерним или производным классом.
- Наследование моделирует отношение «является».
- Требуется для создания иерархичности – свойства реального мира.

Приведение вверх и вниз по иерархии

- Приведение вверх (к базовому классу) всегда безопасно;
- Приведение вниз может быть опасным;

```
struct A {};  
struct B : public A {};  
struct C : public A {};
```

```
B* b = new B();  
A* a = b;  
C* c = a; // Ошибка компиляции  
C* c = static_cast<C*>(b); // Ошибка компиляции  
C* c = static_cast<C*>(a); // !!!
```

Приведение вверх и вниз по иерархии

- Приведение вверх (к базовому классу) всегда безопасно;
- Приведение вниз может быть опасным;

```
struct A {};  
struct B : public A {};  
struct C : public A {};
```

```
B* b = new B();  
A* a = b;  
C* c = a; // Ошибка компиляции  
C* c = static_cast<C*>(b); // Ошибка компиляции  
C* c = static_cast<C*>(a); // !!!
```

Множественное наследование

- Сначала выделяется память под базовый класс
- Затем – под наследуемый
- Подобъекты являются полноценными объектами!

```
class Wolf
{
};
class Dog
{
};
class Husky : public Wolf, public Dog
{
};
```



Представление в памяти при наследовании



Инструменты для исследования

- В целях повышения быстродействия данные в памяти должны быть выровнены, то есть размещены определенным образом;

- Предпочтительное выравнивание можно узнать:

```
std::cout << alignof(char) << std::endl;    // 1
```

```
std::cout << alignof(double) << std::endl; // 8
```

- `sizeof(T)` - размер типа в байтах
- `offsetof(T, M)` - смещение поля M от начала типа T

Инструменты для исследования

```
struct S
{
    char m1;
    double m2;
};
```

```
sizeof(char) == 1
sizeof(double) == 8
sizeof(S) == 16
offsetof(S, m1) == 0
offsetof(S, m2) == 8
```

```
[          char          ][          double          ]
[c][.][.][.][.][.][.][.][d][d][d][d][d][d][d][d]
```

Инструменты для исследования

```
#pragma pack(push, 1)
class S
{
public:
    char m1;
    double m2;
};
#pragma pack(pop)
```

```
offsetof(S, m1) == 0
offsetof(S, m2) == 1
sizeof(S) == 9
```

Простые типы (POD, Plain old data)

1. Скалярные типы (`bool`, числа, указатели, перечисления (`enum`), `nullptr_t`)
2. `class` или `struct` которые:
 - a. Имеют только **тривиальные** (сгенерированные компилятором) конструктор, деструктор, конструктор копирования;
 - b. Нет виртуальных функций и базового класса;
 - c. Все нестатические поля с модификатором доступа `public`;
 - d. Не содержит статических полей не POD типа.
 - e. **В C++20 это понятие изменилось**



1. Разбор понятий: trivial type, standard layout, POD
<https://habr.com/ru/articles/532972/>

Простые типы (POD, Plain old data)^{C++20}

1. Класс со стандартным выравниваем (standard layout) это класс, который
 - a. не имеет нестатических элементов данных типа класса с нестандартным выравниванием,
 - b. не имеет виртуальных функций и виртуальных базовых классов,
 - c. имеет одинаковый контроль доступа для всех нестатических элементов данных,
 - d. не имеет базовых классов с нестандартным выравниваем,
 - e. ещё кое-что (см. ссылку)
2. То есть то, что C может понять.



1. https://en.cppreference.com/w/cpp/language/classes#Standard-layout_class

Простые типы (POD, Plain old data)

```
class NotPOD
{
public:
    NotPOD(int x)
    {
    }
};
```

```
class NotPOD
    : public Base
{
};
```

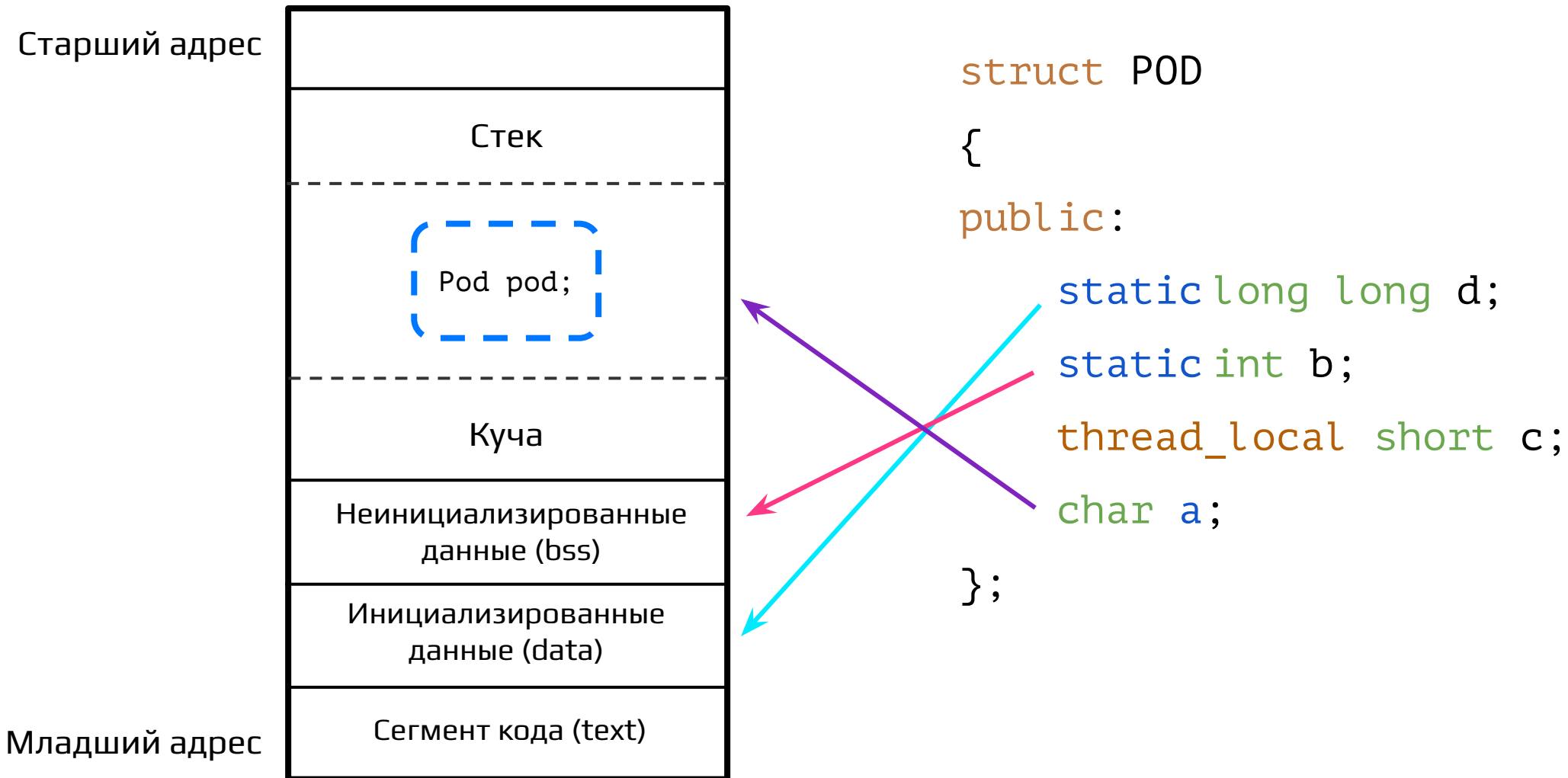
```
class NotPOD
{
    virtual void f()
    {
    }
};
```

```
class NotPOD
{
    int x;
};
```

Простые типы (POD, Plain old data)

```
class POD
{
public:
    POD_ m1;
    int m2;
    static double m3;
private:
    void f() {}
};
```

Data layout: static members

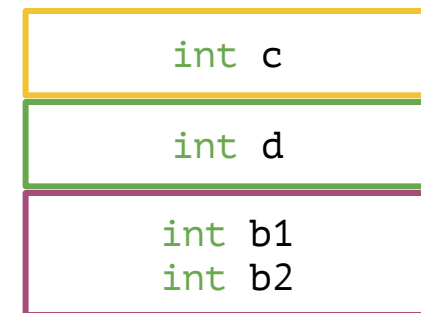


Наследование

- Память классов родителей идёт в начале в последовательности указанной после ":";
- Приведение к базовому классу не требует действий;
- Приведение ко второму базовому классу, уже требует прибавить смещение

Общее правило: последовательность размещения наследуемых классов строится обходом дерева наследования DFS (Depth First, в глубину).

```
struct C
{
    int c;
};
struct D
{
    int d;
}
struct B : C, D
{
    int b1;
    int b2;
};
```



Code time



- Разберём наследование;
- Выравнивание;
- Мотивация для виртуальных функций;

Виртуальные функции



Виртуальные функции

- Решают проблему, связанную с полем типа, предоставляя возможность программисту объявить в базовом классе функции, которые можно заместить в каждом производном классе.
- Производный класс, которые не нуждается в собственной версии виртуальной функции, не обязан её реализовывать;
- Функция из производного класса с тем же именем и с тем же набором типов аргументов, что и виртуальная функция в базовом классе, *замещает* (override) виртуальную функцию из базового класса;
- Тип, имеющий виртуальные функции, называется полиморфным типом.



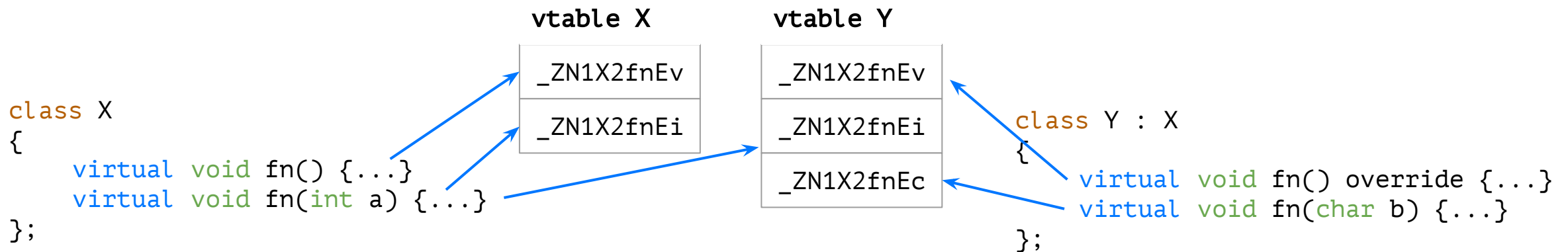
1. Скотт Мейерс. Эффективное использование C++. 55 верных советов улучшить структуру и код ваших программ. Правило 36. Никогда не переопределяйте наследуемые неvirtуальные функции.

Таблица виртуальных функций (1)

1. Если какая-либо функция класса объявлена как виртуальная, создается `vtable`, которая хранит адреса виртуальных функций этого класса;
2. Для всех таких классов компилятор добавляет скрытую переменную `vptr`, которая указывает на `vtable`;
3. Если виртуальная функция не переопределена в производном классе, `vtable` производного класса хранит адрес функции в родительском классе;

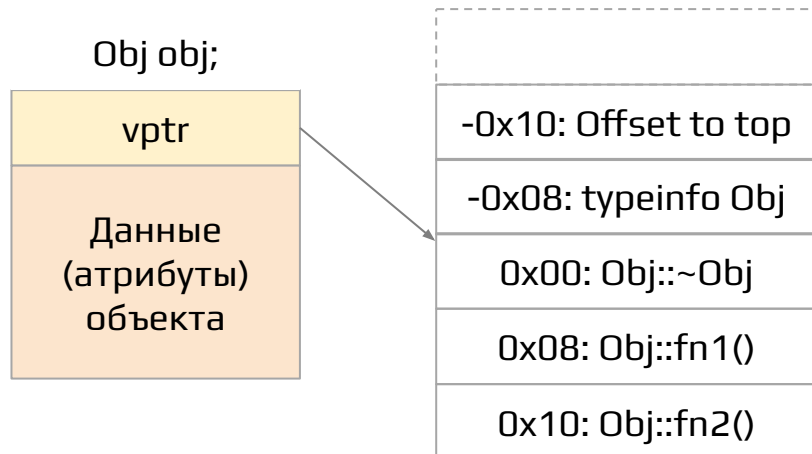
Таблица виртуальных функций (2)

1. Если в унаследованном классе есть функция с такой же сигнатурой, то заменяется соответствующий слот в таблице этого класса;
2. Если такой сигнатуры нет - выделяем новый слот;
3. Вызов виртуальной функции идёт через указатель в этой таблице.



vtable

1. В начале каждого объекта с виртуальными функциями есть указатель `vptr` на `vtable`
2. `vptr` указывает на элемент с индексом 2 (от начала);
3. `vptr[0]` содержит указатель на самую первую виртуальную функцию;
4. Виртуальные функции раскладываются в `vptr[n]` в порядке объявления;
5. `vptr[-1]` указывает на структуру `typeinfo` объекта;
6. `vptr[-2]` содержит смещение до начала объекта;
7. После `vptr` идут данные класса, в порядке объявления;
8. Неявно созданные деструкторы и т.п. располагаются в конце.



Виртуальный деструктор

- Когда объект производного класса уничтожается через указатель на базовый класс с неvirtуальным деструктором, то результат не определен;
- Во время исполнения это обычно приводит к тому, что часть объекта, принадлежащая производному классу, никогда не будет уничтожена
- **Правило:** Объявляйте виртуальный деструктор! При удалении объектов производных классов будет происходить именно то, что нужно.



1. Скотт Мейерс. Эффективный использование C++. Правило 7. Объявляйте деструкторы виртуальными в полиморфном базовом классе.

Абстрактные классы

- Класс с одной или несколькими чисто виртуальными функциями называется абстрактным классом;
- Абстрактный класс можно использовать только как интерфейс и в качестве базы для других классов;

```
class A
{
public:
    virtual void foo() = 0;
}
```


Операторы



Операторы

- `bool operator==(const T& other) const`
- `bool operator!=(const T& other) const`
- `T operator+(const T& other) const`
- `T operator-() const`
- `T& operator++() // ++x`
- `T operator++(int) // x++`
- `const T& operator[](size_t i) const`
- `std::strong_ordering operator<=>(const T&) constC++20`
- Также есть операторы `new`, `delete` и `,` (запятая)
 - `void* operator new (size_t)`
 - `void operator delete (void *)`



1. Скотт Мейерс. Эффективное использование С++. Правило 51: Придерживайтесь принятых соглашений при написании `new` и `delete`.

Оператор космический корабль

- Определяет, какое из соотношений $A < B$, $A == B$ или $A > B$ имеет место;
- Сгенерированный компилятором оператор трёхстороннего сравнения сравнивает указатели, а не объекты, на которые они ссылаются
- Три категории сравнения: строгая (strong ordering), слабая (weak ordering) и частичная упорядоченная (partial ordering)



1. Райнер Гримм. С++20 в деталях. 4.3 Оператор трёхстороннего сравнения

Code time



- Дорабатываем пример для вывода на устройство;
- Напишем класс длинной арифметики с некоторыми операторами;

Домашнее задание



Домашнее задание #3 (1)

Нужно написать класс-матрицу, тип элементов `int32_t`. В конструкторе задается количество столбцов и количество строк.

Поддерживаются операции:

- получить количество строк(`rows`)/столбцов(`columns`);
- получить конкретный элемент;
- умножить на число (`*=`);
- сравнение на равенство/неравенство.

В случае ошибки выхода за границы бросать исключение:

```
throw std::out_of_range("")
```

Домашнее задание #3 (2)

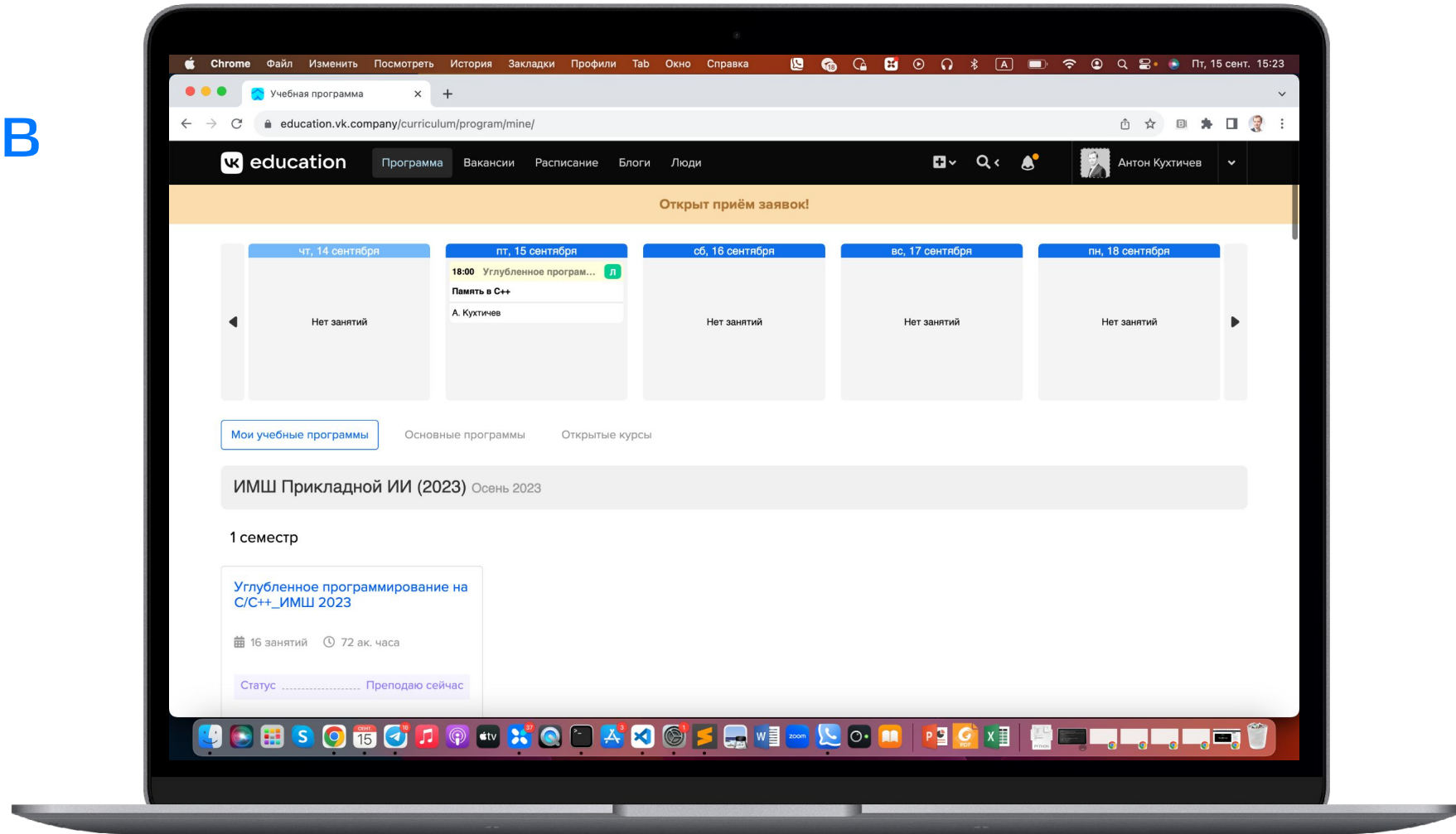
Чтобы реализовать семантику `[] []` понадобится прокси-класс. Оператор матрицы возвращает другой класс, в котором тоже используется оператор `[]` и уже этот класс возвращает значение.

Полезная литература в помощь

- Джош Ласпинозо «С++ для профи»
- Скотт Мейерс «Эффективный и современный С++»
- Бьерн Страуструп «Язык программирования С++»

Напоминание оставить отзыв

Это правда важно





Спасибо
за внимание!