Углублённое программирование на C++

Функции в C++

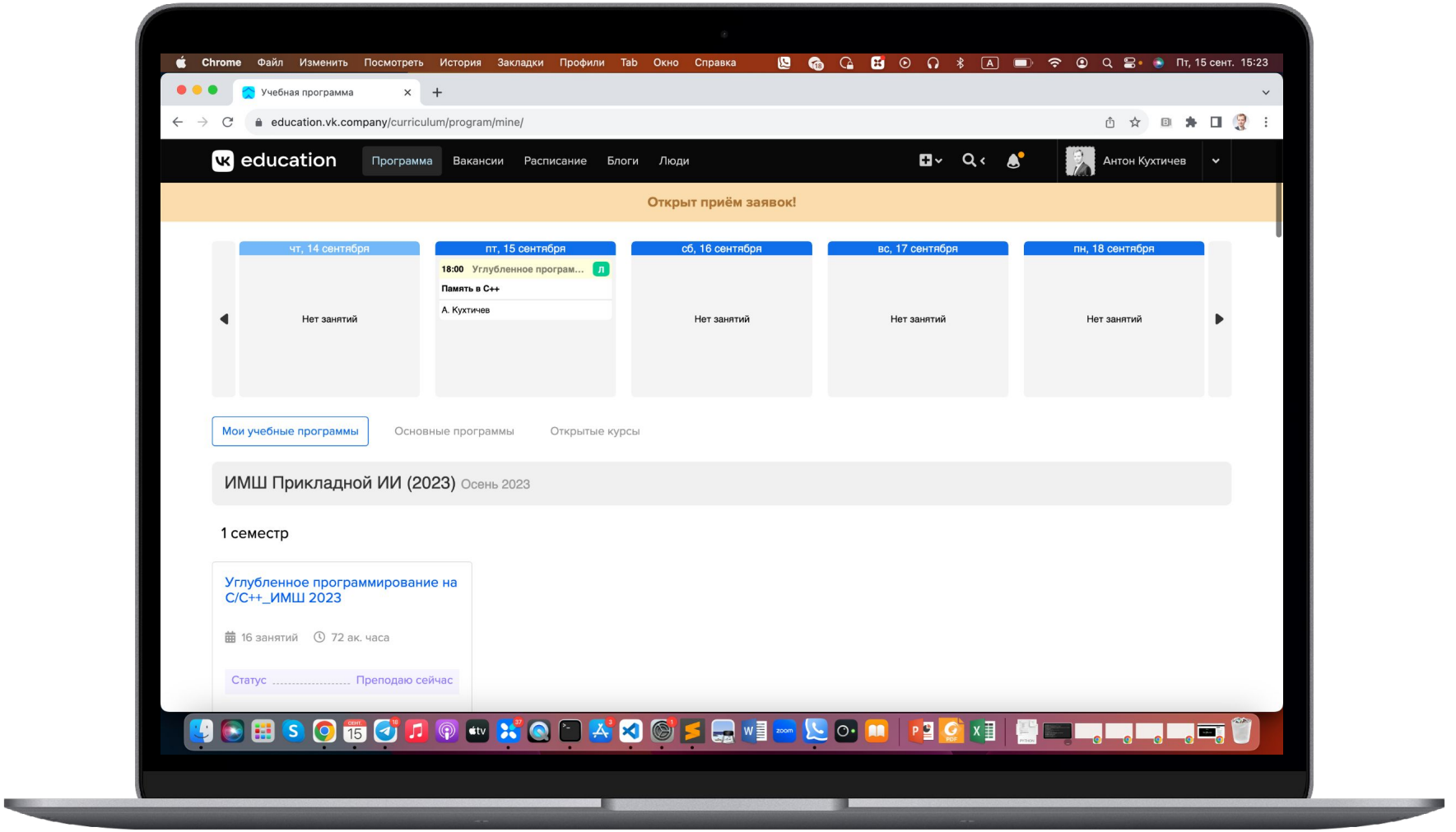
Кухтичев Антон

education

09 октября 2025 года

Напоминание отметиться на портале

и оставить отзыв после лекции



Содержание занятия

- Квиз
- Функции
- Соглашения о вызовах
- Атрибуты
- Встраиваемые функции (inline)
- Функтор
- Ссылки
- λ-выражения
- Функции высшего порядка
- std::function
- std::bind

Мем недели


Питон: просто скажи мне имя переменной, я все сделаю сам


Плюсы: это копия указателя на вектор, это указатель на константный вектор или константный указатель на не константный вектор я тебя спрашиваю???

Квиз #2

Цель занятия

- Сформировать знание о концепции функций в C++, включая перегрузку и механизмы её разрешения
- Освоить использование префиксных и суффиксных модификаторов для написания эффективного и читаемого кода

Функции

«Функции должны делать только что-то одно
и очень хорошо»

— Роберт С. Мартин, «Чистый код»

Функции

Объявление функций имеют следующую форму:

```
префиксные-модификаторы возвращаемый-тип имя-функции(аргументы)
суффиксные-модификаторы
```

```
int square();
int square(double num);
```

Функции

- Это кусок кода, который может выполнить процессор, который находится по определённому адресу;
- Функция может возвращать значение, тип которого в большинстве случаев аналогично типу самой функции;
- Если функция не возвращает никакого значения, то она должна иметь тип `void` (такие функции иногда называют процедурами);
- Функция может принимать параметры (а может и не принимать);
- Функции можно перегружать;

Префиксные модификаторы

- `static` указывает, что функция, не являющаяся членом класса, имеет внутреннюю связь (не будет использоваться за пределами этой единицы трансляции);
- `virtual` указывает, что метод может быть переопределен дочерним классом;
- `constexpr` указывает, что функция должна быть выполнена во время компиляции, если это возможно;
- `[[noreturn]]` указывает, что эта функция не возвращает значения. Этот атрибут помогает компилятору оптимизировать код;
- `inline`, который играет роль в руководстве компилятором для оптимизации кода

Суффиксные модификаторы

- `noexcept` указывает, что функция никогда не вызовет исключение. Это позволяет определенные оптимизации;
- `const` указывает, что метод не будет изменять экземпляр своего класса, что позволяет ссылкам на типы const вызывать метод;
- `volatile`
- `final` указывает, что метод не может быть переопределен дочерним классом;
- `override` указывает, что метод является переопределённым методом базового класса;

Аргументы по умолчанию

-

Стек вызовов

- Поместите аргументы в регистры и в стек вызовов;
- Вставьте адрес возврата в стек вызовов;
- Перейдите к вызываемой функции;
- После завершения функции перейдите к адресу возврата;
- Очистите стек вызовов.

Соглашения о вызовах

cdecl, fastcall, thiscall

cdecl (c-declaration)

Перед вызовом функции вставляется код, называемый прологом (prolog) и выполняющий следующие действия:

- сохранение значений регистров, используемых внутри функции
- запись в стек аргументов функции

После вызова функции вставляется код, называемый эпилогом (epilog) и выполняющий следующие действия:

- восстановление значений регистров, сохранённых кодом пролога
- очистка стека (от локальных переменных функции)

thiscall

Соглашение о вызовах, используемое компиляторами для языка C++ при вызове методов классов.

Отличается от **cdecl**-соглашения только тем, что указатель на объект, для которого вызывается метод (указатель this), записывается в регистр ecx.

fastcall

Передача параметров через регистры: если для сохранения всех параметров и промежуточных результатов регистров недостаточно, используется стек (в gcc через регистры ecx и edx передаются первые 2 параметра).

System V AMD64 ABi (Linux, MacOS, FreeBSD)

- 6 регистров (RDI, RSI, RDX, RCX, R8, R9) для передачи integer-like аргументов
- 8 регистров (XMM0-XMM7) для передачи double/float
- если аргументов больше, они передаются через стек
- для возврата integer-like значений используются RAX и RDX (64 бит + 64 бит)

Code time!

- Смотрим на ассемблерный код для cdecl, fastcall

Встраиваемые функции (inline)

Атрибуты

- Атрибуты позволяют программистам накладывать дополнительные ограничения на исходный код и давать компилятору новые возможности для оптимизации.
- Можно использовать атрибуты для типов, переменных, функций, имен и блоков кода.
- `[[noreturn]]`$^{C++11}$: обозначает, что функция не возвращает значений;
- `[[deprecated]]`$^{C++14}$: означает, что вы не должны использовать это имя;
- `[[maybe_unused]]`$^{C++17}$: подавляет предупреждение компилятора о неиспользуемом имени
- `[[nodiscard("reason")]]`$^{C++20}$: выводит предупреждение, если возвращаемое значение отбрасывается

inline

- Указывает компилятору, что он должен пытаться каждый раз генерировать в месте вызова код, соответствующий функции;
- Компилятор умный и он может не встроить код;
- Тело inline-функции должно быть определено в той же единице трансляции, где она вызывается (*One definition rule (ODR)* игнорируется).

```
inline void foo()
{
}
```

Мейерс С. “Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов”. Правило 33. Тщательно обдумывайте использование встраиваемых функций.

Просим ещё настойчивее

- `__attribute__((always_inline))`
- `__forceinline`
- `[[gnu::always_inline]]`

Компилятор пытается встроить функцию вне зависимости от характеристик функции.

В некоторых случаях компилятор может игнорировать встраивание:

- Рекурсивная функция никогда не встраивается в себя;
- Если в функции используется функция `alloca()`;
- Всё равно гарантий нет!

Примерчик

```
__attribute__((always_inline)) void foo() { }

// или

[[gnu::always_inline]] void foo() { }


#ifdef __GNUC__

#define __forceinline __attribute__((always_inline))

#endif
```

Функтор (функциональный объект)

Функтор (функциональный объект)

```
template <class T>
class Less
{
    const T& x_;
public:
    Less(const T& x) : x_(x) {}

    bool operator()(const T& y) const
    {
        return y < x_;
    }
};

Less lessThen3(3);

bool result = lessThen3(5); // false
```

Передача аргументов

По ссылке и по значению

Передача аргументов по значению

- В функции окажется копия объекта, её изменение не отразится на оригинальном объекте;
- Копировать большие объекты может оказаться накладно;

```
void foo(int a) { a += 1; }

int a = 1;
foo(a); // всё ещё 1.
```

Передача аргументов по ссылке

- Можно передавать аргументы по ссылке
  - Копирования не происходит, все изменения объекта внутри функции отражаются на объекте;
  - Следует использовать, если надо изменить объект внутри функции.

```
void foo(int &a);
```

Передача аргументов по константной ссылке

- Копирования не происходит, при попытке изменения объекта будет ошибка
- Большие объекты выгоднее передавать по ссылке, маленькие - наоборот

```
void foo(const int &a);
```

Передача аргументов по указателю

- Копирования не происходит;
- Если указатель на константный объект, то при попытке изменения объекта будет ошибка;
- Есть дополнительный уровень косвенности, возможно придется что-то подгрузить в кеш из дальнего участка памяти;
- Реализуется optional-концепция.

```
void foo(int *a);
```

Перегрузка функций

Перегрузка функций

- Использование одного имени для операции, выполняемой с различными типами, называется **перегрузкой;**
- Процесс поиска подходящей функции из множества перегруженных заключается в нахождении наилучшего соответствия типов формальных и фактических аргументов;
- Функции, объявленные в различных областях видимости (не пространствах имён), не являются перегруженными;
- Объявление небольшого количества перегруженных вариантов функции может привести к неоднозначности;
- Перегруженная функция - декорированная функция;

Перегрузка функций

1. Точное соответствие типов; то есть полное соответствие или соответствие, достигаемое тривиальными преобразованиями типов;
2. Компилятор попытается использовать целые числа и числа с плавающей точкой для получения подходящей перегрузки (например, `int` для `long` или `float` для `double`);
3. Компилятор попытается сопоставить типы, используя стандартные преобразования, такие как преобразование целочисленного типа в тип с плавающей точкой или приведение указателя на потомка к указателю на родителя;
4. Компилятор будет искать пользовательское преобразование;
5. Компилятор будет искать вариативную функцию (многоточие `...`);

Вариативные функции

```
int sum(size_t n, ...)
{
    va_list args;
    va_start(args, n);
    int result{};
    while(n--) {
        auto next_element = va_arg(args, int);
        result += next_element;
    }
    va_end(args);
    return result;
}
```

Вариативные функции

- Вариативные функции являются пережитком языка C;
- Вариативные аргументы не обеспечивают безопасность типов;
- Количество элементов в вариативный аргументах должно отслеживаться отдельно;
- Компилятор не может помочь разработчику ни с одной из этих проблем;
- Лучше использовать вариативные шаблоны (рассмотрим в лекции про шаблоны), обеспечивают более безопасный и эффективный способ реализации вариативных функций.

Пространство имён

Пространства имён

- Проблема

```
// math.h
double cos(double x);

// ваш код
double cos(double x);
```

Пространства имён

- Решение в стиле C

```
// ваш код
double fastCos(double x);
```

Пространства имён

- Решение в стиле C++

```
namespace fast
{
    double cos(double x);
}

fast::cos(x);
cos(x); // вызов из math.h
```

Пространства имён

1. Проверка в текущем namespace;
2. Если имени нет и текущий namespace глобальный - ошибка;
3. Рекурсивно поиск в родительском namespace.

Пространства имён

```
void foo() {} // ::foo

namespace A
{
    void foo() {} // A::foo

    namespace B
    {
        void bar() // A::B::bar
        {
            foo(); // A::foo
            ::foo(); // foo()
        }
    }
}
```

Указатель на функцию

Указатель на функцию

В общем случае объявляется так:

```
возвращаемый-тип (*имя-указателя)(тип-аргумента1, тип-аргумента2,
...);
```

Указатель на функцию

В общем случае объявляется так:

```
возвращаемый-тип (*имя-указателя)(тип-аргумента1, тип-аргумента2,
...);
```

```
void foo(int x) { }

typedef void (*FooPtr)(int);
// или используя using
using FooPtr = void(*)(int);

FooPtr ptr = foo;
ptr(5);
```

Функции высшего порядка

- Функция высшего порядка — функция, принимающая в качестве аргументов другие функции или возвращает другую функцию в качестве результата.

```
void sort(int* data, size_t size, bool (*compare)(int x, int y));

bool less(int x, int y)
{
    return x < y;
}

sort(data, 100, less);
```

-выражения

λ-выражения

```
[захват](параметры) спецификаторы -> возвращаемый тип {тело}
```

- Захват (captures): переменные-члены объекта функции (то есть частично примененные параметры);
- Параметры (parameters): параметры необходимые для вызова объект функции;
- Тело (body): код объекта функции;
- Спецификаторы (specifiers): элементы вроде constexpr, mutable, noexcept и [[noreturn]];
- тип возвращаемого значения (return type): тип, возвращаемый объектом функции.

Рассмотрим код

```
bool isEven(int i) { return i % 2 == 0; }


void foo()
{
    std::vector<int> arr;
    …
    std::find_if(std::begin(arr), std::end(arr), isEven);
}
```

λ-выражения

```
void foo()
{
    std::vector<int> arr;
    auto comp = [](int i) { return i % 2 == 0 };
    …
    std::find_if(std::begin(arr), std::end(arr), comp);
}
```

Список захвата

- Если не указать &, то будет захват по значению, то есть копирование объекта; если указать, то по ссылке (нет копирования, модификации внутри функции отразяться на оригинальном объекте).

```
// Захват всех переменных в области видимости по значению
auto foo = [=]() {};
// Захват всех переменных в области видимости по ссылке
auto foo = [&]() {};
```

Список захвата

- `[] // без захвата переменных из внешней области видимости`
- `[=] // все переменные захватываются по значению`
- `[&] // все переменные захватываются по ссылке`
- `[x, y] // захват x и y по значению`
- `[&x, &y] // захват x и y по ссылке`
- `[in, &out] // захват in по значению, а out — по ссылке`
- `[=, &out1, &out2] // захват всех переменных по значению,`
  `// кроме out1 и out2, которые захватываются по ссылке`
- `[&, x, &y] // захват всех переменных по ссылке, кроме x`

std::function<>

std::function

```
std::function<возвращаемый-тип (тип-аргумента1, тип-аргумента2, ...)>
```

- Шаблон стандартной библиотеки C++11, который обобщает идею указателя на функцию;
- Может ссылаться на любой вызываемый объект, т.е. на всё, что может быть вызвано как функция;
- Могут хранить, копировать и вызывать произвольные вызываемые объекты — функции, λ-выражения, выражения связывания и другие функциональные объекты;

std::function

```
#include <functional>

using MoveFunction =
    std::function<void (int& x, int& y)>;

void moveLeft(int &x, int &y) {}

MoveFunction getRandomDirection() { ... }

std::vector<MoveFunction> trajectory =
{
   [](int& x, int& y)  { … },
   moveLeft,
   getRandomDirection()
};
```

std::bind

std::bind

Позволяет получить функциональный объект с требуемым интерфейсом.

```
if (std::all_of(v.begin(), v.end(),
    [](int x) { return x < 5; }))
{
    ...
}
```

std::bind

Позволяет получить функциональный объект с требуемым интерфейсом.

```
bool lessThan(int v, int max)
{
    return v < max;
}

auto lessThan3 = std::bind(lessThan, std::placeholders::_1, 3);

if (std::all_of(v.begin(), v.end(), lessThan3))
{
    ...
}
```

Домашнее задание

Домашнее задание #2

Необходимо написать библиотеку-парсер строк состоящих из следующих токенов:

- строка (например, “`hello`”, “`l2go`”)
- число (например, “`91`”, “`001`”)

Число состоит из символов от 0 до 9, строка - все остальные символы. Токены разделены пробелами, символами табуляции и перевода строки.

Пользователь библиотеки должен иметь возможность зарегистрировать callback-функцию вызываемую каждый раз, когда найден токен, функция получает токен. Должно быть возможно зарегистрировать свой обработчик под каждый тип токена. Также должны быть колбеки вызываемые перед началом парсинга и по его окончанию.

Подведём итоги

1. Функции в C++ позволяют **структурировать код,** а их перегрузка дает возможность объявлять несколько функций с одним именем, но разными параметрами

2. **Компилятор разрешает перегрузку функций**, выбирая наилучшее соответствие типов аргументов среди доступных вариантов

3. **Префиксные и суффиксные модификаторы управляют поведением функций**, определяя область видимости, исключения и возможность переопределения

4. **Соглашения о вызовах (cdecl, fastcall, thiscall) влияют на передачу параметров и возврат значений**, что важно для совместимости кода и производительности

5. **Встраиваемые функции (inline) могут ускорять выполнение кода**, но чрезмерное их использование увеличивает размер бинарного файла

6. **Функции можно передавать и хранить с помощью указателей**, функторов и std::function, что позволяет использовать их как аргументы в других функциях

7. **Лямбда-выражения ([]() {}) позволяют создавать компактные анонимные функции**, упрощая работу с алгоритмами стандартной библиотеки

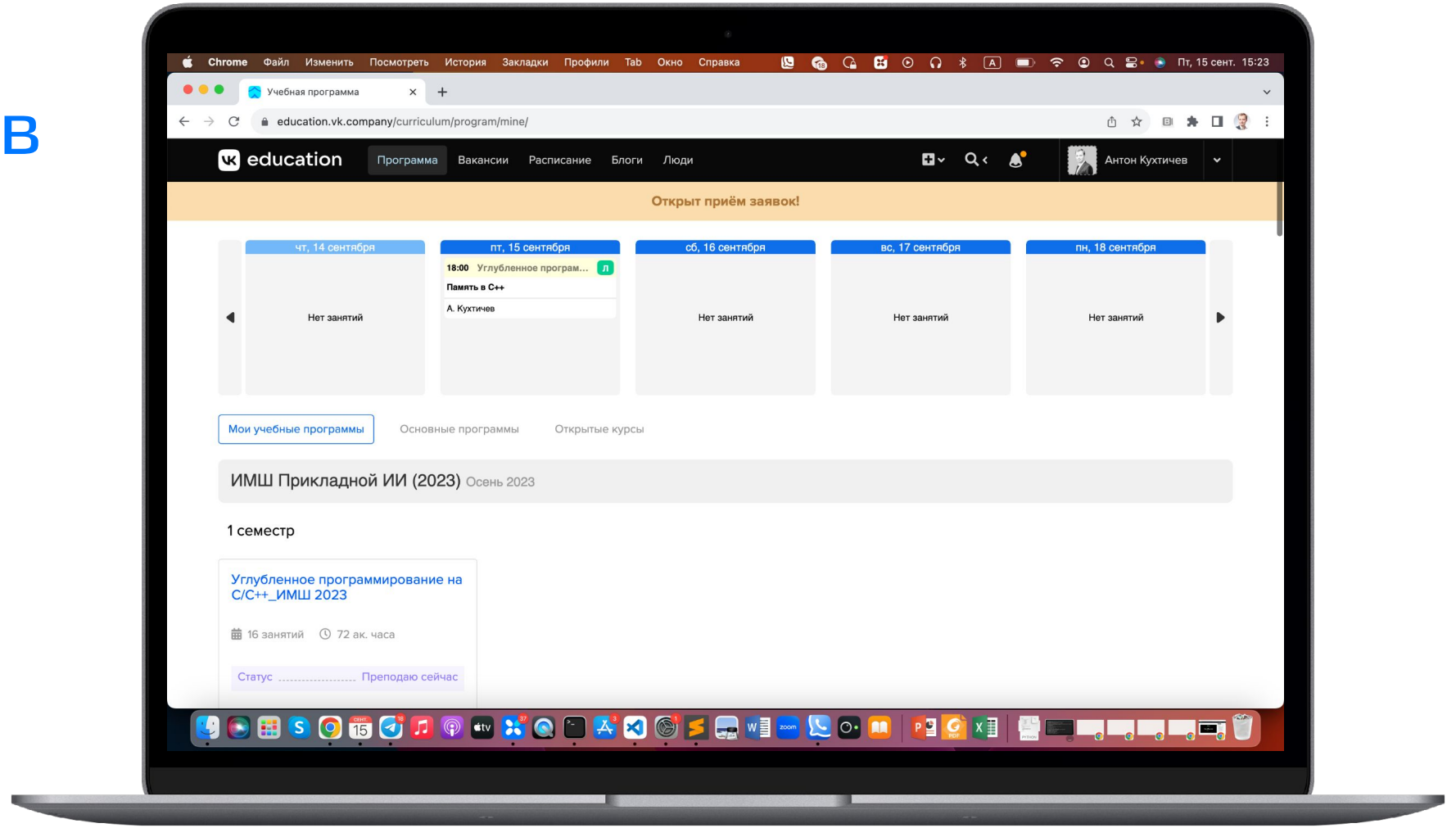
8. **std::bind позволяет частично применять параметры к функциям**, создавая новые функции с предустановленными аргументами

Полезная литература в помощь

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

Напоминание оставить отзыв

Это правда важно



Спасибо
за внимание!