Углублённое программирование на C++

Семантика копирования и перемещения

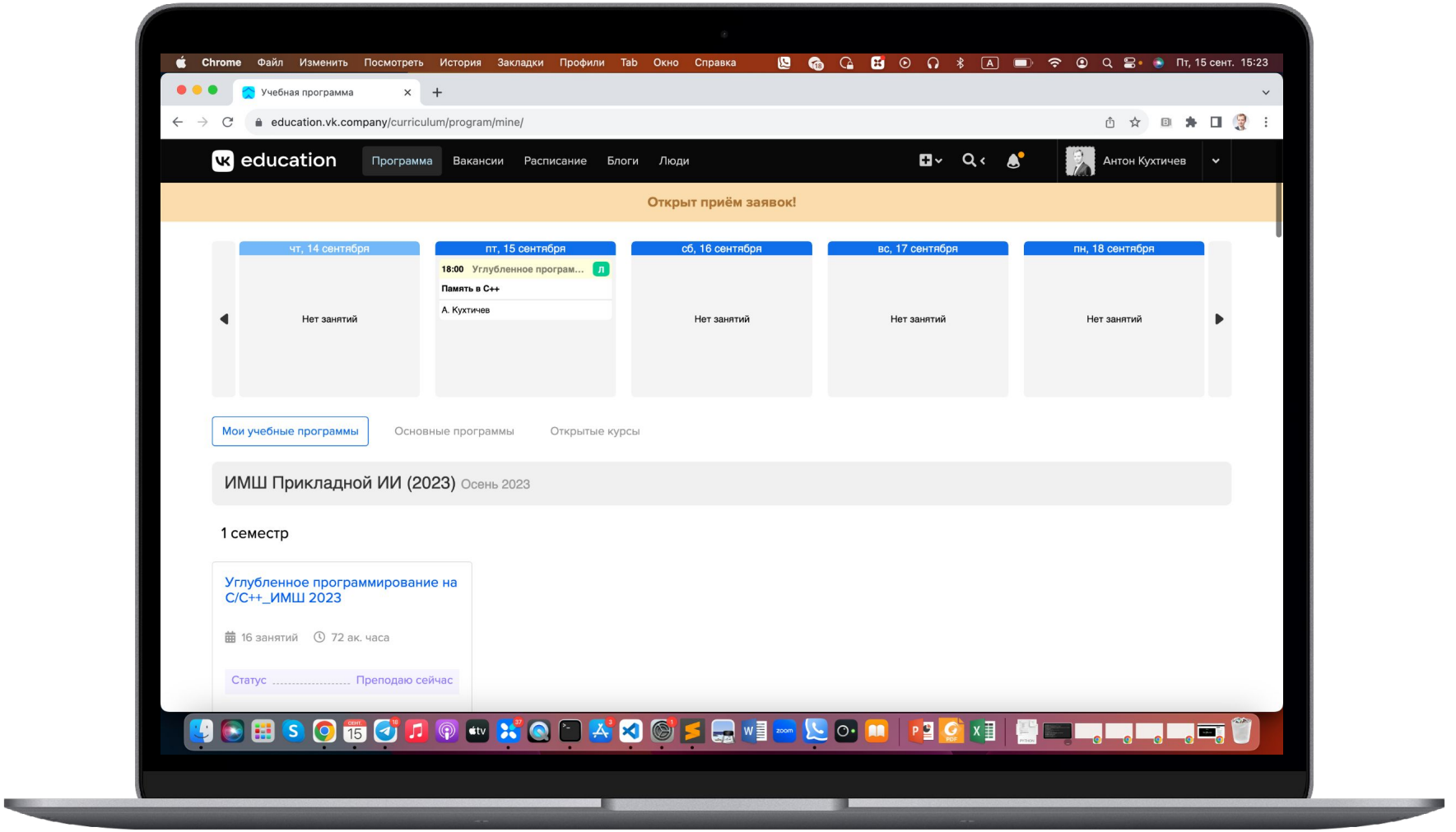
Кухтичев Антон

education

24 апреля 2025 года

Напоминание отметиться на портале

и оставить отзыв после лекции



Содержание занятия

- Квиз
- Правило тройки (пятерки)
- lvalue и rvalue
- Копирование
- Перемещение
- Return value optimization (RVO)
- Copy elision

Мем недели

Квиз

Именованные функции преобразования

Именованные функции преобразования

```
named-conversion<desired-type>(object-to-cast);
```

- const_cast
- static_cast
- reinterpret_cast
- dynamic_cast

const_cast

Удаляет модификатор `const`, позволяя модифицировать значение `const`.

```
void foo(const int& arg)
{
    int &alias = const_cast<int&>( arg );
}
```

static_cast

Отменяет чётко определенное неявное преобразование, такое как приведение целочисленного типа к другому целочисленному типу.

```
int i1 = 11;
int i2 = 3;
float x = static_cast<float>(i1) / i2;

int i = 90;
i = static_cast<int>(i / 3.6);
```

reinterpret_cast

```
int foo()
{
    auto timer = reinterpret_cast<const unsigned long *>(0x1000);
    printf("Timer is %lu.", *timer);
}
```

dynamic_cast

```
void foo(A* a)
{
    B* b = dynamic_cast<B*>(a);

    if (b)
        b -> methodSpecificToB();
    else
        std::cerr << "Этот объект не является объектом типа B" << std::endl;
}
```

ADL и другие штучки

Argument-dependent name lookup (ADL), или Поиск, зависящий от аргументов

Известен также, как поиск Кёнига (Koenig lookup).

Компилятор ищет функцию в текущем пространстве имен и если не находит, то в пространствах имен аргументов. Если находит подходящую функцию в двух местах, то возникает ошибка.

Argument-dependent name lookup (ADL), или Поиск, зависящий от аргументов

```
namespace X
{
    struct A { ... };

    std::ostream& operator<<(std::ostream& out, const A& value) { ...
}

    void foo(const A& value) { ... }
}

X::A a;
std::cout << a;
foo(a);
```

friend-функции

```
class Y {};

class A
{
    int data; // private data member
    class B {}; // private nested type
    enum { a = 100 }; // private enumerator
    friend class X; // friend class forward declaration (elaborated class
specifier)
    friend Y; // friend class declaration (simple type specifier) (since
C++11)
    // the two friend declarations above can be merged since C++26:
    // friend class X, Y;
};

class X : A::B // OK: A::B accessible to friend
{
    A::B mx; // OK: A::B accessible to member of friend
    class Y {
        A::B my; // OK: A::B accessible to nested member of friend
```

Методы генерируемые компилятором неявно

```
struct A
{
    X x;
    Y y;
    // Конструктор
    A() : x(X()), y(Y()) {}
    // Деструктор
    ~A() {}
   ...
};
```

Методы генерируемые компилятором неявно

```
struct A
{
    ...
    // Копирующий конструктор
    // A a1;
    // A a2 = a1;
    A(const A& copied)
        : x(copied.x)
        , y(copied.y)
    {}
   ...

};
```

Методы генерируемые компилятором неявно

```
struct A
{
    ...
    // Оператор копирования
    // A a1;
    // A a2;
    // a2 = a1;
    A& operator=(const A& copied)
    {
        x = copied.x;
        y = copied.y;
        return *this;
    }
};
```

Методы генерируемые компилятором неявно

```
struct A
{
    ...
    // Перемещающий конструктор
    // A a1;
    // A a2 = std::move(a1);
    A(A&& moved)
        : x(std::move(moved.x))
        , y(std::move(moved.y))
    {}
    ...
};
```

Методы генерируемые компилятором неявно

```
struct A
{
    // Оператор перемещения
    // A a1;
    // A a2;
    // a2 = std::move(a1);
    A& operator=(A&& moved)
    {
        x = std::move(moved.x);
        y = std::move(moved.y);
        return *this;
    }
    ...
};
```

rvalue и lvalue

rvalue и lvalue

До стандарта C++11 было два типа значений:

1. `lvalue`
2. `rvalue`

"Объект — это некоторая **именованная область памяти**; lvalue — это выражение, обозначающее объект. Термин "lvalue" произошел от записи присваивания E1 = E2, в которой левый (left — левый(англ.), отсюда буква l, value — значение) операнд E1 должен быть выражением lvalue."

rvalue и lvalue

1. Ссылается на объект — `lvalue`
2. Если можно взять адрес — `lvalue`
3. Все что не `lvalue`, то `rvalue`

```
int a = 1;
int b = 2;
int c = (a + b);
int foo() { return 3; }
int d = foo();

1 = a;         // left operand must be l-value
foo() = 2;     // left operand must be l-value
(a + b) = 3; // left operand must be l-value
```

Ещё примерчики

```
int a = 3;
a; // lvalue
int& b = a;
b; // lvalue, ссылается на a
int* c = &a;
*c; // lvalue, ссылается на a
void foo(int val)
{
    val; // lvalue
}
```

```
void foo(int& val)
{
    val; // lvalue, ссылается на val
}
int& bar() { return a; }
bar(); // lvalue, ссылается на a
3; // rvalue
(a + b); // rvalue
int bar() { return 1; }
bar(); // rvalue
```

lvalue-ссылка

Ссылка на lvalue

```
int a = 3;
int& b = a;

int& a = 3;          // ошибка
const int& a = 3; // ок
a; // const lvalue
```

Объект жив до тех пор, пока жива ссылающаяся на него константная ссылка.

rvalue-ссылка

Ссылка на rvalue

```cpp
#include <iostream>

int x = 0;

int val() { return 0; }
int& ref() { return x; }

void test(int&)
{
    std::cout << "lvalue\n";
}
```

std::move приводит lvalue к rvalue

```cpp
void test(int&&)
{
    std::cout << "rvalue\n";
}

int main()
{
    test(0); // rvalue
    test(x); // lvalue
    test(val()); // rvalue
    test(ref()); // lvalue
    test(std::move(x)); // rvalue
    return 0;
}
```

Копирование

Копирование

Семантика: в результате копирования должна появится точная копия объекта.

```
int x = 3;
int y = x;
// x == y


String a;
String b = a;
String c;
c = a;
// a == b == c
```

Code time

- Конструктор/оператор копирования

Копирование и наследование

```
struct A
{
    A() {}
    A(const A& a) {}
    virtual A& operator=(const A&
copied)
    { return *this; }
};
```

```
class B : public A
{
public:
    B() {}
    B(const B& b) : A(b) {}

    A& operator=(const A& copied)
override
    {
        A::operator=(copied);
        return *this;
    }
};
```

Срезка

```
void foo(A a)
{
    // Срезанный до A объект
}

B a;
foo(a);
```

Предпочитайте удалённые функции

```
class Seed {
public:
    ...
    // Не хотим генерации копирующих методов!
    Seed(const Seed&) = delete;
    Seed& operator=(const Seed&) = delete;
};
```

1. Скотт Мейерс. Эффективный и современный C++. Пункт 3.5. Предпочитайте удалённые функции закрытым неопределённым.

Перемещение

Перемещение

Семантика: в результате перемещения в объекте, куда происходит перемещение, должна появиться точная копия перемещаемого объекта, оригинальный объект после этого остается в неопределенном, но корректном состоянии.

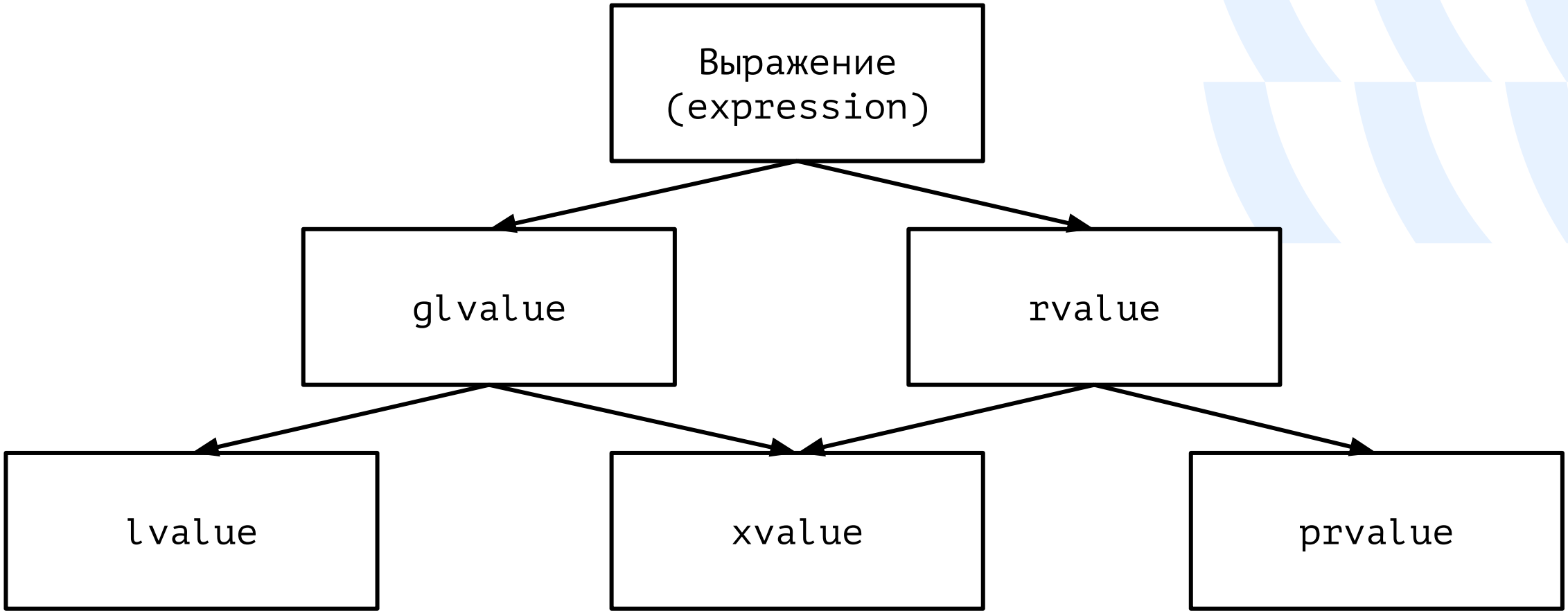
Передача владения

```
class unique_ptr
{
    T* data_;
};
```

Производительность

```
class Buffer
{
    char* data_;
    size_t size_;
};
```

lvalue и rvalue начиная с C++11



lvalue и rvalue начиная с C++11

- `glvalue` ("generalized" lvalue)

  Выражение, чьё вычисление определяет сущность объекта.
- `prvalue` ("pure" rvalue)

  Выражение, чьё вычисление инициализирует объект или вычисляет значение операнда оператора, с соответствии с контекстом использования.
- `xvalue` ("eXpiring" value)

  Это glvalue, которое обозначает объект, чьи ресурсы могут быть повторно использованы (обычно потому, что они находятся около конца своего времени жизни).

lvalue и rvalue начиная с C++11

- `lvalue`

  Это `glvalue`, которое не является `xvalue`.

- `rvalue`

  Это `prvalue` или `xvalue`.

lvalue

Выражение является `lvalue`, если ссылается на объект уже имеющий имя доступное вне выражения.

```
int a = 3;
a; // lvalue
int& b = a;
b; // lvalue
int* c = &a;
*c; // lvalue

int& foo() { return a; }
foo(); // lvalue
```

xvalue

- Результат вызова функции возвращающей rvalue-ссылку

```
int&& foo() { return 3; }
foo(); // xvalue
```

- Явное приведение к rvalue

```
static_cast<int&&>(5); // xvalue
std::move(5); // эквивалентно static_cast<int&&>
```

xvalue

- Результат доступа к нестатическому члену, объекта xvalue значения
- Объекты, чьи ресурсы могут быть "перемещены". Это rvalue, но с адресом (например, результат std::move).

```
struct A
{
    int i;
};
A&& foo() { return A(); }
foo().i; // xvalue
std::string s1 = "Hello";
std::string s2 = std::move(s1);  // std::move(s1) — xvalue
```

prvalue

- Не принадлежит ни к lvalue, ни к xvalue;
- Временные значения без идентичности. Они инициализируют объекты или участвуют в вычислениях.

```
int foo() { return 3; }
foo(); // prvalue
int func() { return 42; }
int a = func();  // func() — prvalue
int b = 3 + 5;   // (3 + 5) — prvalue
```

rvalue и glvalue

- Объединяет `lvalue` и `xvalue`. Это выражения, имеющие адрес в памяти.
- `rvalue`

  Всё, что принадлежит к xvalue или prvalue.
- `glvalue`

  Всё, что принадлежит к xvalue или lrvalue.

Практическое правило (Скотт Мейерс)

1. Можно взять адрес — `lvalue`
2. Ссылается на lvalue (T&, const T&) — `lvalue`
3. Иначе `rvalue`

```
int arr[3] = {1, 2, 3};
arr[0] = 5;  // arr[0] — lvalue (glvalue)
std::move(x);// xvalue (glvalue)
```

Примеры

```
void foo(int) {}    // 1
void foo(int&) {}   // 2
void foo(int&&) {} // 3
```

1)

```
int x = 1;
foo(x); // lvalue
```

2)

```
int x = 1;
int& y = x;
foo(y); // lvalue
```

3)

```
foo(1); // rvalue
```

4)

```
int bar() { return 1; }
foo(bar()); // rvalue
```

5)

```
foo(1 + 2); // rvalue
```

Code time

- Конструктор/оператор перемещения

Return value optimization (RVO)

Позволяет сконструировать возвращаемый объект в точке вызова.

Чтобы отключить оптимизацию – используйте опцию `-fno-elide-constructors`

```
Server makeServer(uint16_t port)
{
    Server server(port);
    server.setup(...);
    return server;
}
Server s = makeServer(8080);
```

Не мешайте компилятору:

```
Server&& makeServer(uint16_t port)
{
    Server server(port);
    server.setup(...);
    // так не надо, RVO не применяется
    return std::move(server);
}
```

Copy elision

Оптимизация компилятора разрешающая избегать лишнего вызова копирующего конструктора.

```
struct A
{
    explicit A(int) {}
    A(const A&) {}
};
A y = A(5); // Копирующий конструктор вызван не будет
```

В копирующих конструкторах должна быть логика отвечающая только за копирование.

Домашнее задание

Домашнее задание #4 (1)

Написать класс для работы с большими целыми числами. Размер числа ограничен только размером памяти. Нужно поддержать семантику работы с обычным `int32_t`:

```
BigInt a = 1;
BigInt b = a;
BigInt c = a + b + 2;
BigInt d;
d = std::move(c);
std::cout << d << std::endl;
```

Домашнее задание #4 (2)

Реализовать:

- оператор вывода в поток;
- сложение;
- копирующий и перемещающий конструкторы/операторы;
- вычитание;
- умножение (можно Карацубу для особо любопытных);
- унарный минус;
- все операции сравнения.

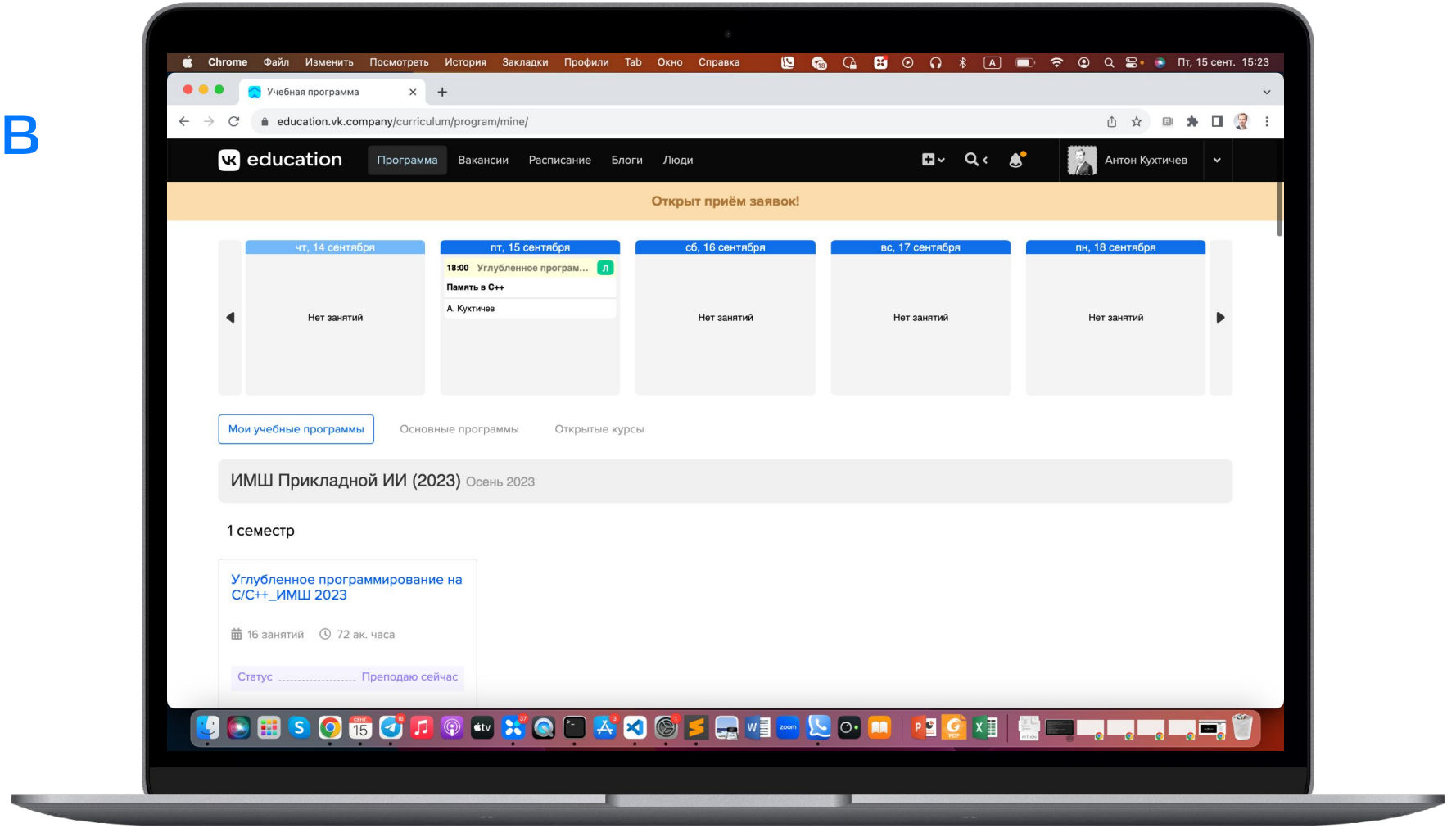
`std::vector` и другие контейнеры использовать нельзя - управляйте памятью самостоятельно.

Полезная литература в помощь

- Скотт Мейерс “Эффективный и современный C++”
- Бьерн Страуструп “Языка программирования C++”
- Статья Страуструпа про выражения

Напоминание оставить отзыв

Это правда важно



Спасибо
за внимание!