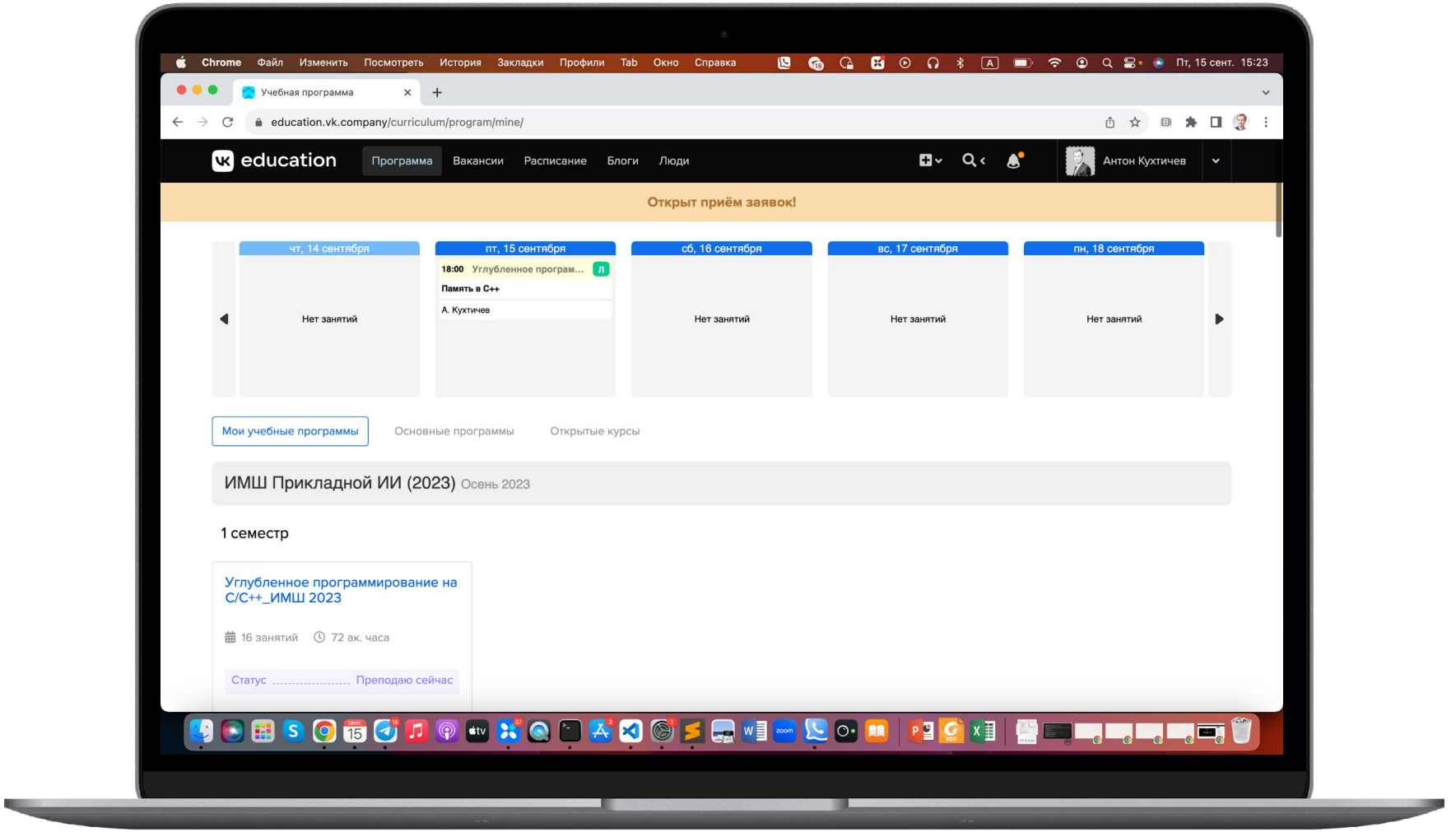
Исключения

Антон Кухтичев

Напоминание отметиться на портале

и оставить отзыв
после лекции

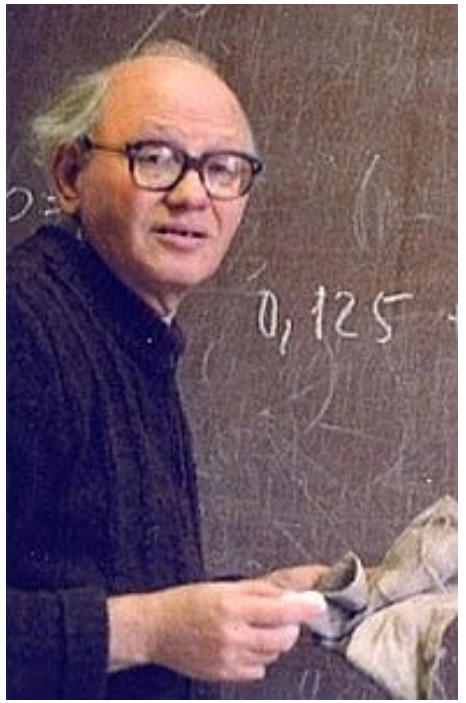


Содержание занятия

- Концепты и требования (Concepts & requires)
- Структурное связывание
- Универсальная ссылка
- Обработка ошибок
- noexcept
- Гарантии безопасности исключений (exception safety)
- Поиск подходящего обработчика
- Исключения в конструкторе/деструкторе
- Точки следования (sequence points)

Угадай программиста по фотографии

Кто это?



Концепты и требования

Концепты

Концепт это шаблон. Это постоянное выражение, включающее аргументы шаблона, вычисляемое во время компиляции. Представьте концепт как один большой предикат: функцию, которая вычисляет true или false.

```
template<typename T1, typename T2, …>
concept ConceptName = …
```

Требования

```
requires (argument-1, argument-2, …) {
   {  expression1 } -> return-type-requirement1;
   {  expression2 } -> return-type-requirement2;
}
```

`return-type-requirement` может быть следующим:

- `std::same_as<T>` - возвращаемый тип T;
- `std::convertible_to<T>` - возвращаемый тип можно привести к T;

Code time

- Перепишем пример с типом, который должен иметь

  void foo();

Структурное связывание

structured binding

Структурное связывание 1

```
std::tuple<std::string, uint8_t> CreatePerson();
{
    return std::make_tuple(“Say my name”, 52);
}

{
    auto items = CreatePerson();
    std::string name = std::get<0>(items);
    uint8_t age = std::get<1>(items);
}
```

Структурное связывание 2

```
std::tuple<std::string, uint8_t> CreatePerson();
{
    return std::make_tuple(“Say my name”, 52);
}

{
    std::string name;
    uint8_t age;
    std::tie(name, age) = CreatePerson();
}
```

Структурное связывание 3

```
std::tuple<std::string, uint8_t> CreatePerson();
{
    return std::make_tuple(“Say my name”, 52);
}

// Начиная с 17-го стандарта.
{
    auto [name, age] = CreatePerson();
}

{
    std::set<std::string> myset{"hello"};
    if (auto [iter, success] = myset.insert("Hello"); success)
    //...
}
```

1. Adding C++17 structured bindings support to your classes
2. STRUCTURED BINDINGS in C++
3. Восемь возможностей C++17, которые должен применять каждый разработчик

Универсальная ссылка

Универсальная ссылка

```
void f(Widget&& param); // rvalue-ссылка

Widget&& var1 = Widget(); / rvalue-ссылка

auto &&var2 = var1; // Не rvalue-ссылка

template<typename T>
void f(std::vector<T>&& param); // rvalue-ссылка

template<typename T>
void f(T&& param); // Не rvalue-ссылка
```

1. Скотт Мейерс. Эффективный и современный C++. 5.2 Отличие универсальных ссылок от rvalue-ссылок.

Универсальная ссылка

```
template<typename T>
void foo(T&& param); // param является универсальной ссылкой.

// ...
Widget w;

// В f передаётся lvalue, тип param -- Widget &.
foo(w);

// В f передаётся rvalue, тип param -- Widget &&.
foo(std::move(w));
```

Обработка ошибок

Обработка ошибок

1. Возврат кода ошибки
2. Исключения

Возврат кода ошибки

```
enum class Error
{
    Success,
    Failure
};

Error doSomething()
{
    return Error::Success;
}

if (doSomething() != Error::Success)
{
    showError();
}
```

Возврат кода ошибки

+ Простота
- Ошибку можно проигнорировать
- Делает код громоздким

Code time

- Напишем обработку ошибок

Исключения

```
struct Error
{
    std::string message_;
    const char* fileName_;
    int line_;
    Error(const std::string& message,
        const char* fileName, int line)
        : message_(message)
        , fileName_(fileName)
        , line_(line)
    {
    }
};
```

Исключения

```
void doSomething()
{
    throw Error(
        "doSomething error", __FILE__, __LINE__);
}

try
{
    doSomething();
}
catch (const Error& error)
{
    showError();
}
```

Исключения

- Вопросы производительности
- При неправильном использовании могут усложнить программу
+ Нельзя проигнорировать

Что такое исключительная ситуация?

Ошибка которую нельзя обработать на данном уровне и игнорирование которой делает дальнейшую работу программы бессмысленной.

noexcept

noexcept

```
void foo() noexcept
{
}
```

`noexcept` говорит компилятору, что функция не выбрасывает исключений — это позволяет компилятору генерировать более компактный код, но если фактически исключение было выброшено, то будет вызвана функция `terminate`.

noexcept

- `noexcept` является частью интерфейса функции, а это означает, что вызывающий код может зависеть от наличия данного модификатора.
- Функции, объявленные как `noexcept`, предоставляют большие возможности оптимизации, чем функции без такой спецификации
- Спецификация `noexcept` имеет особое значение для операции перемещения, обмена, функций освобождения памяти и деструкторов

1. Скотт Мейерс. Эффективный и современный C++. Пункт 3.8: Если функции не генерируют исключений, объявляйте их как noexcept.

Гарантии безопасности исключений (exception safety)

Гарантировано исключений нет (No-throw guarantee)

Операции всегда завершаются успешно, если исключительная ситуация возникла она обрабатывается внутри операции.

Строгая гарантия (Strong exception safety)

Также известна как коммит ролбек семантика (commit/rollback semantics). Операции могут завершиться неудачей, но неудачные операции гарантированно не имеют побочных эффектов, поэтому все данные сохраняют свои исходные значения.

Строгая гарантия (Strong exception safety)

```
std::vector<int> source = ...;
try
{
    std::vector<int> tmp = source;
    tmp.push_back(getNumber());
    tmp.push_back(getNumber()); // <-- Исключение
    tmp.push_back(getNumber());
    source.swap(tmp);
}
catch (...)
{
    return;
}
```

Базовая гарантия (Basic exception safety)

Выполнение неудачных операций может вызвать побочные эффекты, но все инварианты сохраняются и нет утечек ресурсов (включая утечку памяти). Любые сохраненные данные будут содержать допустимые значения, даже если они отличаются от того, что они были до исключения.

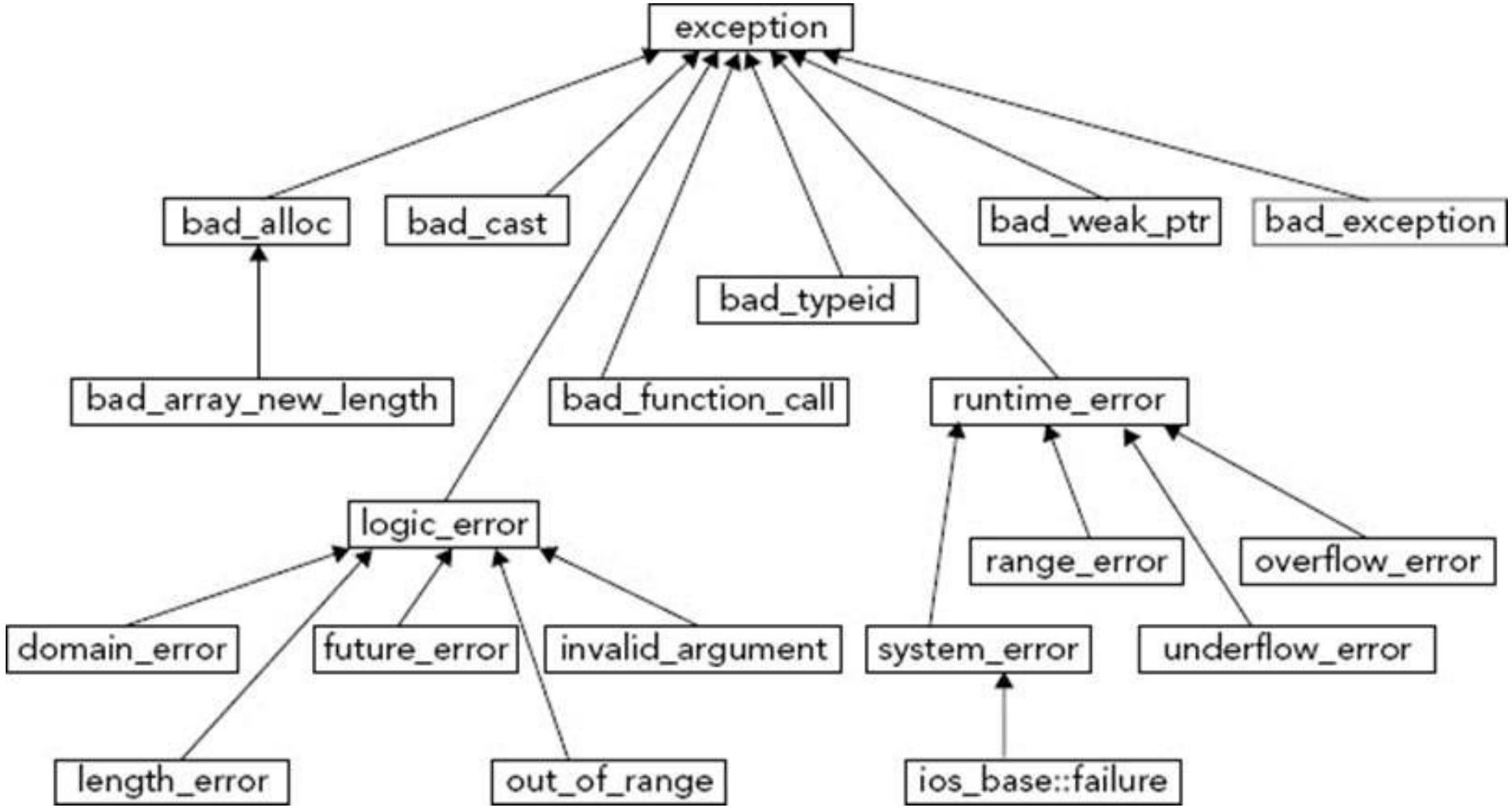
```
source.push_back(getNumber());
source.push_back(getNumber()); // <-- Исключение
source.push_back(getNumber());
```

Никаких гарантий (No exception safety)

Так делать не стоит.

Поиск подходящего обработчика

Иерархия исключений



Поиск подходящего обработчика

1. Поиск подходящего обработчика идет в порядке следования обработчиков в коде
2. Полного соответствия типа не требуется, будет выбран первый подходящий обработчик
3. Если перехватывать исключение по значению, то возможна срезка до базового класса
4. Если наиболее общий обработчик идет раньше, то более специализированный обработчик никогда не будет вызван
5. Три точки - перехват любого исключения

Поиск подходящего обработчика идет в порядке следования обработчиков в коде

```
try {
    ...
} catch (...) {
    ...
}
catch (std::invalid_argument &ex) { // этот блок никогда
    ...                             // не получит управление
}
```

Полного соответствия типа не требуется, будет выбран первый подходящий обработчик

```
try {
    ...
} catch (std::logic_error & ex) {
    ...
}
catch (std::invalid_argument &ex) { // этот блок никогда
    ...                             // не получит управление
}
```

Три точки - перехват любого исключения

```
try {
    // какой-то код
} catch (...) {
    // какой-то код
}
```

Раскрутка стека

Поиск подходящего обработчика вниз по стеку вызовов с вызовом деструкторов локальных объектов - раскрутка стека.

Если подходящий обработчик не был найден вызывается стандартная функция `terminate`.

Раскрутка стека

```
struct A {};
struct Error {};
struct FileError : public Error {};

void foo() {
    A a1;
    throw Error();
}
```

```
void bar() {
    A a2;
    try
    {
        A a3;
        foo();
    }
    catch (const FileError&)
    {
    }
}

bar();
```

Исключения под капотом

```
struct A
{
    A() {}
    ~A() {}
};

void bar() noexcept
{
}

void foo()
{
    A a;
    bar();
}
```

```
A::A() [base object constructor]:
        ret
A::~A() [base object destructor]:
        ret
bar():
        ret
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        lea     rdi, [rbp - 8]
        call    A::A() [base object constructor]
        call    bar()
        lea     rdi, [rbp - 8]
        call    A::~A() [base object destructor]
        add     rsp, 16
        pop     rbp
        ret
```

Убираем noexcept

```
struct A
{
    A() {}
    ~A() {}
};

void bar() {}

void foo()
{
    A a;
    bar();
}
```

```
A::A() [base object constructor]:
        ret
A::~A() [base object destructor]:
        ret
bar():
        ret
foo():
        call    A::A() [base object constructor]
        call    bar()
        jmp     .LBB1_1
.LBB1_1:
        call    A::~A() [base object destructor]
        ret
.LBB1_2: # landing pad
        call    A::~A() [base object destructor]
        call    _Unwind_Resume
```

Добавляем блок catch

```
struct A
{
    A() {}
    ~A() {}
};

void bar() {}
void baz() noexcept {}

void foo()
{
    A a;
    try {
        bar();
    }
    catch (...) {
        baz();
    }
}
```

```
foo():
        call    A::A() [base object constructor]
        call    bar()
        jmp     .LBB2_1
.LBB2_1:
        jmp     .LBB2_5
.LBB2_2:
        call    __cxa_begin_catch
        call    baz()
        call    __cxa_end_catch
        jmp     .LBB2_4
.LBB2_4:
        jmp     .LBB2_5
.LBB2_5:
        call    A::~A() [base object destructor]
        ret
.LBB2_6:
        call    A::~A() [base object destructor]
        call    _Unwind_Resume
```

Исключения в конструкторе/деструкторе

Исключения в конструкторе

В C++ удаляются только *полностью сконструированные* объекты, то есть такие, конструкторы которых уже завершили выполнение кода.

1. Скотт Мейерс. Наиболее эффективное использование C++. Правило 10: Не допускайте утечки ресурсов в конструкторе.

Исключения в деструкторе

Исключение покинувшее деструктор во время раскрутки стека или у глобального/статического объекта приведет к вызову `terminate`.

Начиная с C++11 все деструкторы компилятором воспринимаются как помеченные `noexcept` — теперь исключения не должны покидать деструктора никогда.

Code time

- Рассмотрим чем грозит исключения в конструкторе;

Точки следования (sequence points)

Точки следования (sequence points)

Точки следования — это точки в программе, где состояние реальной программы полностью соответствует состоянию следуемого из исходного кода.

Точки следования необходимы для того, чтобы компилятор мог делать оптимизацию кода.

Местонахождение точек

1. В конце каждого полного выражения - ;
2. В точке вызова функции после вычисления всех аргументов
3. Сразу после возврата функции, перед тем как любой другой код из вызываемой функции начал выполняться
4. После первого выражения (a) в следующих конструкциях:

```
a || b
a && b
a, b
a ? b : c
```

Примеры

```
foo(
    std::shared_ptr<MyClass>(new MyClass()),
    bar());
```

Компилятор может заменить это выражение на следующее:

```
auto tmp1 = new MyClass();
auto tmp2 = bar();
auto tmp3 = std::shared_ptr<MyClass>(tmp1);
foo(tmp1, tmp3);
```

Примеры

```
void f(int, int);
int g();
int h();
f(g(), h());
```

По Стандарту неизвестно, какая из функций `g` или `h` будет вызвана первой, но известно, что `f()` будет вызвана последней.

Домашнее задание

Домашнее задание #6 (1)

Написать функцию для форматирования строки, поддерживаться должен любой тип, который может быть выведен в поток вывода. Формат строки форматирования:

```
"{0} any text {1} {0}"
```

Номер в фигурных скобках - номер аргумента.

Домашнее задание #6 (2)

Пример:

```
auto text = format("{1}+{1} = {0}", 2, "one");
assert(text == "one+one = 2");
```

Так же написать свои классы исключений, выбрасывать их и обрабатывать для:

- Фигурные скобки - зарезервированный символ, если встречаются вне контекста {n} выбрасывать исключение.
- Если аргументов меньше, чем число в скобках, и в случае прочих ошибок выбрасывать исключение.

Полезная литература в помощь

- Скотт Мейерс “Эффективный и современный C++”
- Скотт Мейерс “Наиболее эффективное использование C++”
- Бьерн Страуструп “Языка программирования C++”

Спасибо за внимание!

Вопросы?