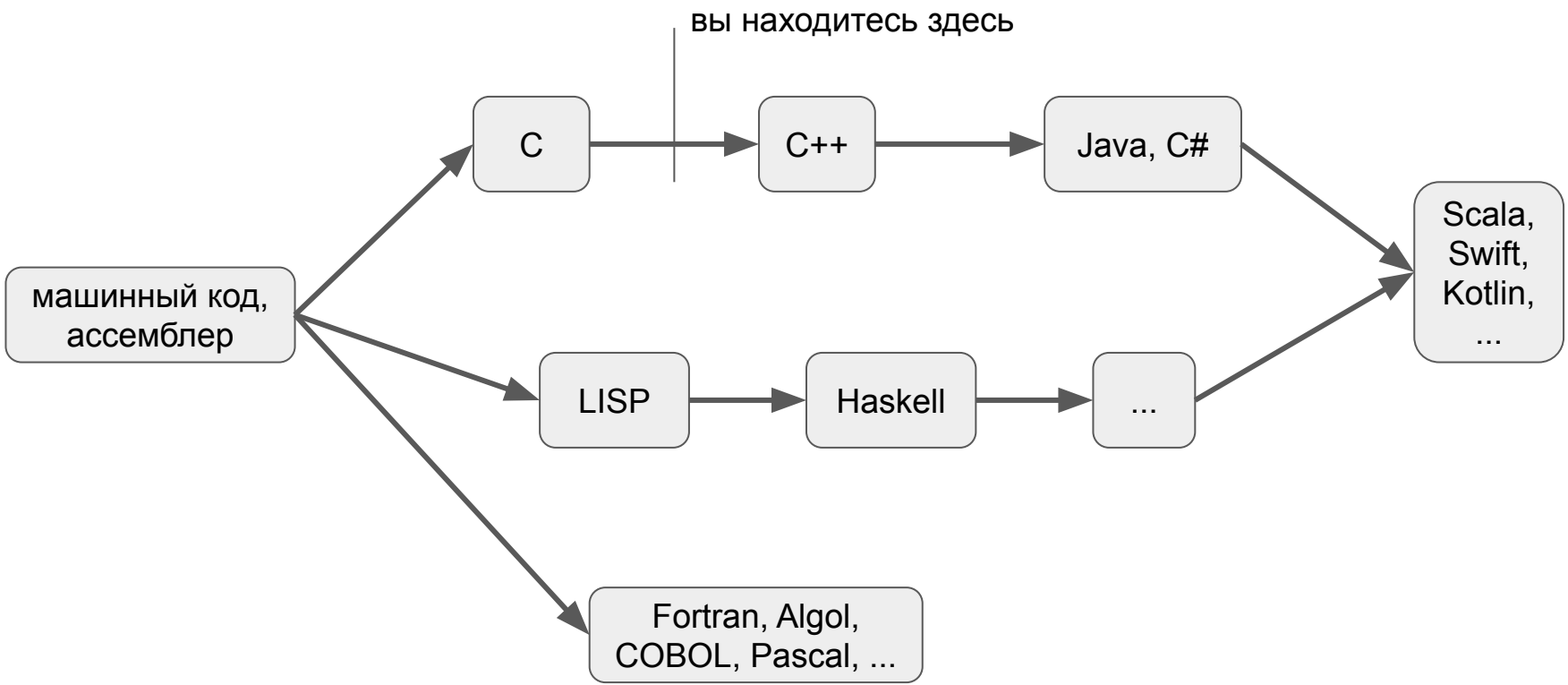
# Первый кит

Соловьёв Владимир Валерьевич
Huawei, НГУ, СУНЦ
vladimir.conwor@gmail.com
vk.com/conwor

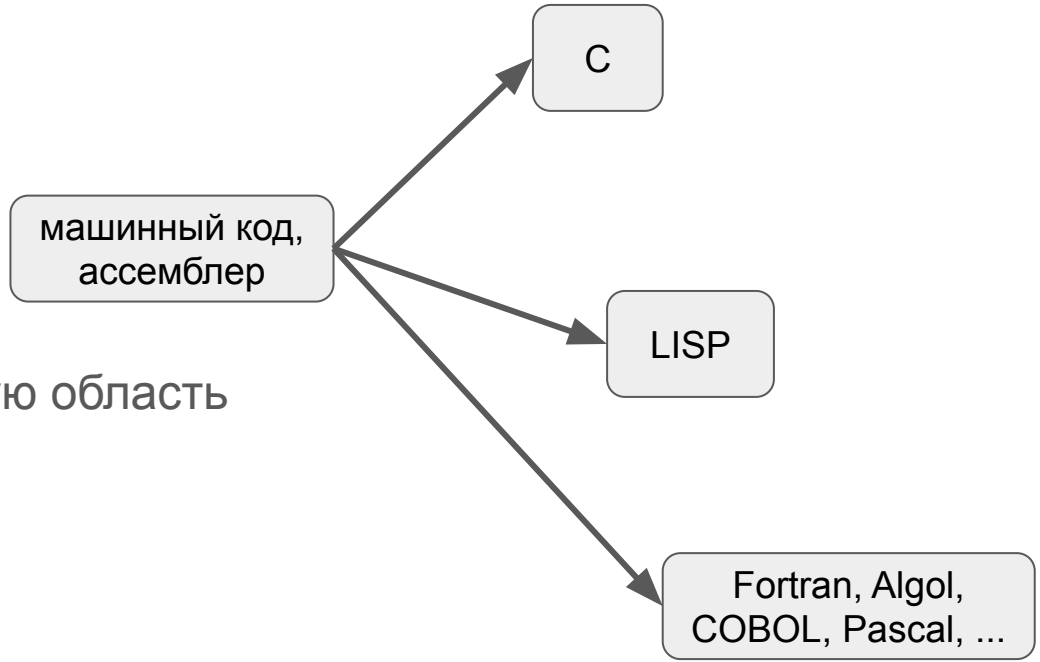
# Развитие ЯП (приблизительное)



# Переход от ассемблеров к языкам

Причины:

1) Сложно и долго писать
2) Машинно-зависимый код
3) Язык не выражает предметную область



# Переход к ООП

Причины:

?


в вашем случае:

зачем учить C++, если любую программу можно написать на C?

# Переход к ООП

Причины:

1) Большие объемы кода и сроки его жизни
2) Многообразие типов данных со сложным поведением


в вашем случае:

зачем учить C++, если любую программу можно написать на C?

# Тип данных

# Тип данных

1) Множество значений
2) Множество операций

# Простые (примитивные) типы

Например, long int языка C

Множество значений - $\{-2^{31}, 2^{31}-1\}$

Множество операций - +, -, *, /, %, …

# Сложные типы

```
struct Stack {
        int* buffer;
        int capacity;
        int size;
}
```

Множество значений -

Множество операций -

# Сложные типы

```
struct Stack {
        int* buffer;
        int capacity;
        int size;
}
```

Множество значений - декартово произведение множеств значений полей

Множество операций - чтение и запись полей

# Сложные типы

```
struct Stack {
        int* buffer;
        int capacity;
        int size;
}
```

Множество значений - декартово произведение множеств значений полей

Множество операций - чтение и запись полей

Это точно то, что нам нужно?

# Сложные типы

***Состояние*** объекта - значения его полей

| | |
|---|---|
| buffer: | 0xABCD0 |
| capacity: | 8 |
| size: | 3 |

# Сложные типы

***Корректные состояния*** - подмножество множества значений, определяющееся тем, какой смысл мы вкладываем в тип

Например, эти состояния не являются корректными для типа Stack

| | |
|---|---|
| buffer: | 0 |
| capacity: | 8 |
| size: | 0 |

| | |
|---|---|
| buffer: | 0xABCD0 |
| capacity: | -24 |
| size: | 23 |

| | |
|---|---|
| buffer: | 0xABCD0 |
| capacity: | 8 |
| size: | 9 |

# Сложные типы

***Инициализация*** объекта - создание начального корректного состояния

***Методы*** - операции, переводящие объект из одного корректного состояния в другое, выполняя при этом какие-то полезные действия

# Инициализация

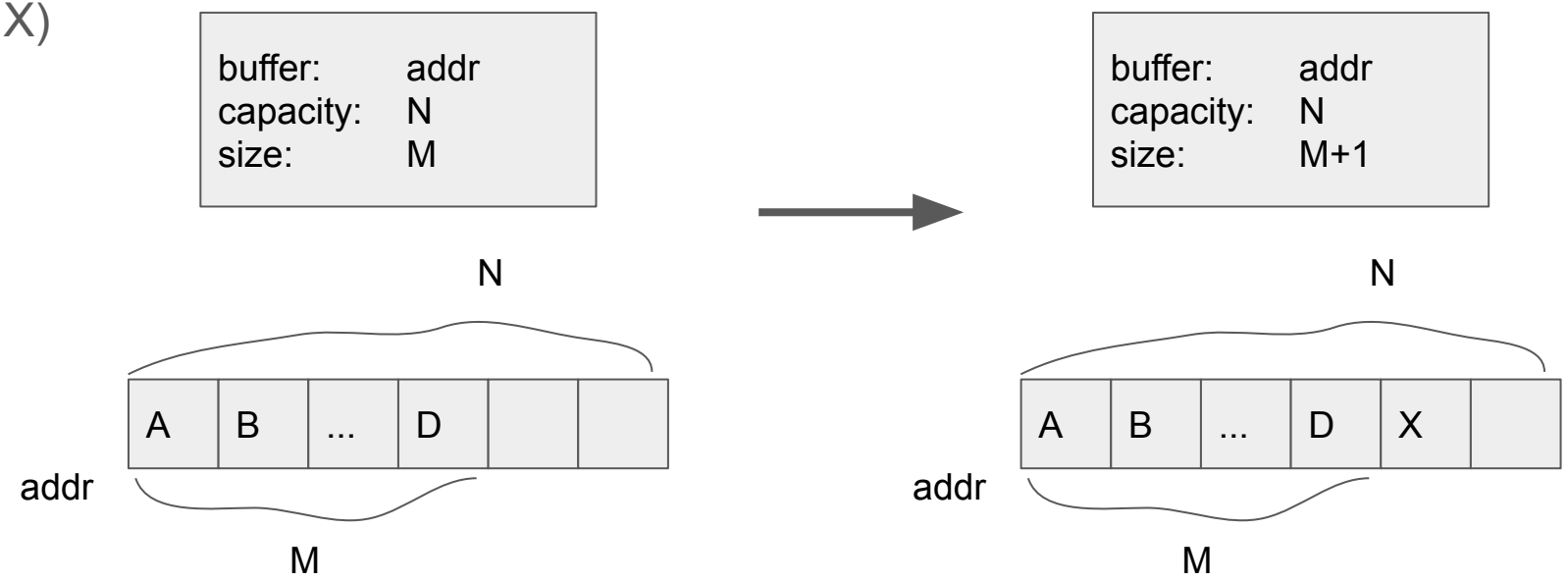
| | |
|---|---|
| buffer: | addr |
| capacity: | N |
| size: | 0 |

Где:

1) N > 0
2) addr - адрес куска памяти размера N*sizeof(int)

# Методы

push(int X)



# А что не так с языком C?

```
typedef struct {
        int* buffer; int capacity; int size;
} Stack;

void stack_init(Stack* s, int cap) {
        s->buffer = (int*) malloc(cap*sizeof(int));
        s->capacity = cap;
        s->size = 0;
}

void stack_push(Stack* s, int x) {
        if (s->size == s->capacity) { … }
        s->buffer[s->size++] = x;
}
```

```
int stack_pop(Stack* s) {
        if (s->size == 0) { … }
        return s->buffer[--(s->size)];
}

---------------
void main() {
        Stack st;
        stack_init(&st, 8);
        stack_push(&st, 37);
        printf(“stack size = %d”, st.size);
        printf(“%d”, stack_pop(&st));
        ...
}
```

# А что не так с языком C?

```
typedef struct {
        int* buffer; int capacity; int size;
} Stack;

void stack_init(Stack* s, int cap) {
        s->buffer = (int*) malloc(cap*sizeof(int));
        s->capacity = cap;
        s->size = 0;
}

void stack_push(Stack* s, int x) {
        if (s->size == s->capacity) { … }
        s->buffer[s->size++] = x;
}
```

```
int stack_pop(Stack* s) {
        if (s->size == 0) { … }
        return s->buffer[--(s->size)];
}

---------------
void main() {
        Stack st;
        stack_init(&st, 8);
        stack_push(&st, 37);
        printf("stack size = %d", st.size);
        printf("%d", stack_pop(&st));
        ...
}
```

# А что не так с языком C?

```
typedef struct {
        int* buffer; int capacity; int size;
} Stack;

void stack_init(Stack* s, int cap) {
        s->buffer = (int*) malloc(cap*sizeof(int));
        s->capacity = cap;
        s->size = 0;
}

void stack_push(Stack* s, int x) {
        if (s->size == s->capacity) { … }
        s->buffer[s->size++] = x;
}
```

```
int stack_pop(Stack* s) {
        if (s->size == 0) { … }
        return s->buffer[--(s->size)];
}

---------------
void main() {
        Stack st;
        stack_init(&st, 8);
        stack_push(&st, 37);
        printf("stack size = %d", st.size);
        printf("%d", stack_pop(&st));
        ...
}
```

# А что не так с языком C?

1) Методы не имеют формального отношения к структуре
   a) Плоское пространство имён + префиксы
   b) Нельзя указать область доступа к деталям реализации (для компилятора функции stack_push и main имеют одинаковое отношение к структуре Stack)

2) Выделение памяти != инициализация объекта

3) Любой код может перевести объект в некорректное состояние

# А что не так с языком C?

1) Плохая ситуация

   Переименовали поле size в length

2) Отвратительная ситуация

   Переименовали поле size в length, а поле capacity - в size

# А что не так с языком C?

На C можно смоделировать сколь угодно хороший высокоуровневый код, но язык при этом:

1) Не предлагает для этого инструментов
2) Вставляет грабли (например, плоское пространство имён)
3) Не вынуждает этого делать

# А что не так с языком C?

На C можно смоделировать сколь угодно хороший высокоуровневый код, но язык при этом:

1) Не предлагает для этого инструментов
2) Вставляет грабли (например, плоское пространство имён)
3) **Не вынуждает этого делать**

А это разве плохо?

# Реальность

1) Большие проекты

   Windows XP - 45 млн. строк кода, Google (2015) - 2 млрд. строк кода

2) Большие сроки поддержки

   Код живет десятилетиями, “с нуля” ничего не переписывается

3) Большие команды + текучка кадров

   Средняя продолжительность (2017) работы в Uber - 1.8 лет, Facebook - 2.5 года, Twitter - 3 года

# А что если

Явно перечислить функции, относящиеся к структуре

# А что если

Явно перечислить функции, относящиеся к структуре

- На основе этого перечисления можно настроить права доступа к полям

# А что если

Явно перечислить функции, относящиеся к структуре

- На основе этого перечисления можно настроить права доступа к полям
- Можно перевернуть сознание
  - Не функция работает со структурой (остальные функции же тоже так могут)
  - А структура “владеет” функцией (так же, как и полями)

# Переворот сознания

“Код, изменяющий данные” уходит на второй план

На первый план выходят типы данных, описывающие:

- Множество значений - корректных состояний
- Множество операций - **методов**, изменяющих состояния в пределах корректности

# Почему это важно

# Почему это важно

Потому что позволяет описывать системы взаимодействия разнородных объектов со сложными правилами поведения так, что:

1) Реализация одной части может изменяться в любой момент (а это годы и десятилетия разработки) независимо от остальных частей
2) Поведение системы становится более предсказуемым (в том числе, верифицируемым с помощью математических моделей)
3) Система лучше защищена от ошибок программистов

# Композиция структуры и методов в C++

Методы описываются в теле структуры

Компилятор делит весь код программы на **код структуры** и **сторонний код**

После этого можно сделать так, что доступ к полям структуры из стороннего кода является ошибкой компиляции

# C

```
typedef struct {
        int* buffer; int capacity; int size;
} Stack;

void stack_init(Stack* s, int cap) {
        s->buffer = (int*) malloc(cap*sizeof(int));
        s->capacity = cap;
        s->size = 0;
}

void stack_push(Stack* s, int x) {
        if (s->size == s->capacity) { … }
        s->buffer[s->size++] = x;
}
```

```
void main() {
        Stack st;
        stack_init(&st, 8);
        stack_push(&st, 37);
        printf(“stack size = %d”, st.size);
        printf(“%d”, stack_pop(&st));
        ...
}
```

# C++

```
struct Stack {

        int* buffer; int capacity; int size;

        Stack(int cap) {
                this->buffer = (int*) malloc(cap*sizeof(int));
                this->capacity = cap;
                this->size = 0;
        }

        void push(int x) {
                if (this->size == this->capacity) { … }
                this->buffer[this->size++] = x;
        }
};
```

```
void main() {
        Stack st(8);
        st.push(37);
        printf(“stack size = %d”, st.size);
        printf(“%d”, st.pop());
        ...
}
```

# C++

```
struct Stack {

        int* buffer, int capacity; int size;

        Stack(int cap) {
                this->buffer = (int*) malloc(cap*sizeof(int));
                this->capacity = cap;
                this->size = 0;
        }

        void push(int x) {
                if (this->size == this->capacity) { … }
                this->buffer[this->size++] = x;
        }
};
```

```
void main() {
        Stack st(8);
        st.push(37);
        printf(“stack size = %d”, st.size);
        printf(“%d”, st.pop());
        ...
}
```

**код структуры**

# C++

```
struct Stack {

        int* buffer; int capacity; int size;

        Stack(int cap) {
                this->buffer = (int*) malloc(cap*sizeof(int));
                this->capacity = cap;
                this->size = 0;
        }

        void push(int x) {
                if (this->size == this->capacity) { … }
                this->buffer[this->size++] = x;
        }
};
```

```
void main() {
        Stack st(8);
        st.push(37);
        printf("stack size = %d", st.size);
        printf("%d", st.pop());
        ...
}
```

**сторонний код**

# C++

```
struct Stack {

        int* buffer; int capacity; int size;

        Stack(int cap) {
                this->buffer = (int*) malloc(cap*sizeof(int));
                this->capacity = cap;
                this->size = 0;
        }

        void push(int x) {
                if (this->size == this->capacity) { … }
                this->buffer[this->size++] = x;
        }
};
```

```
void main() {
        Stack st(8);
        st.push(37);
        printf(“stack size = %d”, st.size);
        printf(“%d”, st.pop());
        ...
}
```

# Области доступа

```
struct Stack {
private:
    int* buffer; int capacity; int size;
public:
    Stack(int cap) {
        this->buffer = (int*) malloc(cap*sizeof(int));
        this->capacity = cap;
        this->size = 0;
    }

    void push(int x) {
        if (this->size == this->capacity) { … }
        this->buffer[this->size++] = x;
    }
};
```

```
void main() {
    Stack st(8);
    st.push(37);
    printf(“stack size = %d”, st.size);
    printf(“%d”, st.pop());
    ...
}
```

# Области доступа

```
struct Stack {
private:
    int* buffer; int capacity; int size;
public:
    Stack(int cap) {
        this->buffer = (int*) malloc(cap*sizeof(int));
        this->capacity = cap;
        this->size = 0;
    }

    void push(int x) {
        if (this->size == this->capacity) { … }
        this->buffer[this->size++] = x;
    }
};
```

```
void main() {
    Stack st(8);
    st.push(37);
    printf("stack size = %d", st.size);
    printf("%d", st.pop());
    ...
}
```

**ошибка компиляции**

# Области доступа

```
struct Stack {
private:
        int* buffer; int capacity; int size;
public:
        Stack(int cap) {
                this->buffer = (int*) malloc(cap*sizeof(int));
                this->capacity = cap;
                this->size = 0;
        }

        void push(int x) {
                if (this->size == this->capacity) { … }
                this->buffer[this->size++] = x;
        }
};
```

```
void main() {
        Stack st(8);
        st.push(37);
        printf("stack size = %d", st.size);
        printf("%d", st.pop());
        ...
}
```

**а здесь доступ есть**

# Области доступа

```
struct Stack {
private:
        int* buffer; int capacity; int size;
public:
        Stack(int cap) { … }
        void push(int x) { … }

        int getSize() {
                return this->size;
        }
};
```

```
void main() {
        Stack st(8);
        st.push(37);
        printf(“stack size = %d”,
                st.getSize());
        printf(“%d”, st.pop());
        ...
}
```

# Указатель this

```
struct Stack {
private:
        int* buffer; int capacity; int size;
public:
        Stack(int cap) {
                this->buffer = (int*) malloc(cap*sizeof(int));
                this->capacity = cap;
                this->size = 0;
        }

        void push(int x) {
                if (this->size == this->capacity) { … }
                this->buffer[this->size++] = x;
        }
};
```

```
void main() {
        Stack st(8);
        st.push(37);
        printf("stack size = %d",
                st.getSize());
        printf("%d", st.pop());
        ...
}
```

# Указатель this

```
struct Stack {
    ...
    void push(/* Stack* this, */ int x) {
        if (this->size == this->capacity) { … }
        this->buffer[this->size++] = x;
    }
```

this - неявный параметр метода типа указателя на структуру, в которой метод объявлен

равен адресу объекта, от которого метод вызывается

# Конструкторы

```
struct Stack {
private:
	int* buffer; int capacity; int size;
public:
	Stack(int cap) {
		this->buffer = (int*) malloc(cap*sizeof(int));
		this->capacity = cap;
		this->size = 0;
	}
};
```

```
void main() {
	Stack st(8);
	...
}
```

# Конструкторы

1) **Неотъемлемая** часть создания объекта

2) Могут принимать параметры

3) Может быть несколько (с разными типами параметров - **перегрузка**)

# Перегрузка конструкторов (до C++11)

```
struct Stack {
private:
	int* buffer; int capacity; int size;

	void init(int cap) {
		this->buffer = (int*) malloc(cap*sizeof(int));
		this->capacity = cap;
		this->size = 0;
	}
public:
	Stack(int cap) { this->init(cap); }
	Stack() { this->init(16) }
};
```

```
void main() {
	Stack st(8);
	Stack st2;
}
```

# Перегрузка конструкторов (начиная с C++11)

```
struct Stack {
private:
	int* buffer; int capacity; int size;
public:
	Stack(int cap) {
		this->buffer = (int*) malloc(cap*sizeof(int));
		this->capacity = cap;
		this->size = 0;
	}

	Stack(): Stack(16) { }
};
```

```
void main() {
	Stack st(8);
	Stack st2;
}
```

# Конструктор по умолчанию

Автоматически генерируется компилятором, если в структуре нет ни одного конструктора

Ничего не делает (кроме *инициализации*)

# Инициализация объекта

Заполнение полей объекта до тела конструктора:

1) Поля примитивных типов и указателей - ничем (мусором)
2) Поля, являющиеся вложенными объектами - объектами, созданными конструкторами по умолчанию

# Список инициализации

```
Stack(int cap) {
        this->buffer = (int*) malloc(cap*sizeof(int));
        assert(this->buffer != NULL)
        this->capacity = cap;
        this->size = 0;
}
```

```
Stack(int cap): capacity(cap), size(0), buffer((int*) malloc(cap*sizeof(int))) {
        assert(this->buffer != NULL);
}
```

# Список инициализации

Выполняется в порядке описания полей в структуре

В большинстве простых случаев приводит к тому же результату, что и присваивания в теле конструктора

В ряде случаев необходим (например, когда присваивание невозможно)

# Список инициализации

**Выполняется в порядке описания полей в структуре**

В большинстве простых случаев приводит к тому же результату, что и присваивания в теле конструктора

В ряде случаев необходим (например, когда присваивание невозможно)

# Список инициализации

```
struct Stack {
private:
	int capacity;
	int size;
	int* buffer;
public:
	Stack(int cap): capacity(cap), size(0), buffer((int*) malloc(cap*sizeof(int))) {
		assert(this->buffer != NULL);
	}
};
```

# Список инициализации

```
struct Stack {
private:
	int capacity;
	int size;
	int* buffer;
public:
	Stack(int cap): capacity(cap), size(0), buffer((int*) malloc(capacity*sizeof(int))) {
		assert(this->buffer != NULL);
	}
};
```

# Список инициализации

```
struct Stack {
private:
        int size;
        int* buffer;
        int capacity;
public:
        Stack(int cap): capacity(cap), size(0), buffer((int*) malloc(capacity*sizeof(int))) {
                assert(this->buffer != NULL);
        }
};
```

ошибка: поле buffer инициализируется раньше, чем поле capacity, но пытается его использовать (читает мусор)

# Список инициализации

Выполняется в порядке описания полей в структуре

В большинстве простых случаев приводит к тому же результату, что и присваивания в теле конструктора

**В ряде случаев необходим (например, когда присваивание невозможно)**

# Список инициализации

```
struct Foo {
private:
	int field;
public:
	Foo(int value) {
		this->field = value;
	}
};
```

Создание объекта структуры Foo обязательно должно проходить через конструктор, причём параметризованный

# Список инициализации

```
struct Foo {
private:
	int field;
public:
	Foo(int value) {
		this->field = value;
	}
};
```

```
struct Bar {
private:
	Foo foo;
public:
	Bar(int valueForFoo) {
	}
};
```

Создание объекта структуры Foo обязательно должно проходить через конструктор, причём параметризованный

# Список инициализации

```
struct Foo {
private:
        int field;
public:
        Foo(int value) {
                this->field = value;
        }
};
```

Создание объекта структуры Foo обязательно должно проходить через конструктор, причём параметризованный

```
struct Bar {
private:
        Foo foo;
public:
        Bar(int valueForFoo) {
        }
};
```

Ошибка компиляции: поле foo не может быть инициализировано, потому что для него нужно вызывать конструктор

# Список инициализации

```
struct Foo {
private:
        int field;
public:
        Foo(int value) {
                this->field = value;
        }
};
```

Создание объекта структуры Foo обязательно должно проходить через конструктор, причём параметризованный

```
struct Bar {
private:
        Foo foo;
public:
        Bar(int valueForFoo) {
                this->foo = Foo(valueForFoo);
        }
};
```

Не помогает - поле должно быть инициализировано **до** тела конструктора

# Список инициализации

```
struct Bar {
private:
	Foo foo;
public:
	Bar(int valueForFoo): foo(valueForFoo) {
	}
};
```

# Оператор new

```
void main() {
        Stack st(8);
        Stack st2;

        Stack* st3 = new Stack(42);
        Stack* st4 = new Stack();

        ...
}
```

# Оператор new

1) Пытается выделить нужное количество байт в динамической памяти

2) Если не удалось, создаёт ошибку (исключение)

3) Вызывает соответствующий конструктор

# Деструкторы

```
struct Stack {
private:
        int* buffer; int capacity; int size;
public:
        ~Stack() {
                free(this->buffer);
        }
};
```

```
void main() {
        Stack st(8);
        ...
}
```

# Деструкторы

```
struct Stack {
private:
        int* buffer; int capacity; int size;
public:
        ~Stack() {
                free(this->buffer);
        }
};
```

```
void main() {
        Stack st(8);
        ...
} // st.~Stack()
```

# Деструкторы

1) **Неотъемлемая** часть уничтожения объекта

2) Для локальных объектов вызываются **неявно**

3) Как следствие, не принимают параметров и не могут быть перегружены

# Оператор delete

```
void main() {
        Stack* st = new Stack(42);
        …
        delete st;
}
```

# Оператор delete

1) Вызывает деструктор

2) Удаляет объект (аналогично free)

# Динамическая память

Средства работы с дин. памятью языка C (malloc/calloc/realloc, free) доступны в языке C++

Главное - не мешать их с new/delete, если память была выделена одними средствами, она должна быть удалена соответствующими

# Первый кит

1) Запись методов в структуре определяет *код структуры* и *сторонний код*
2) Области доступа определяют *детали реализации* и *интерфейс*
3) Создание объекта сопровождается конструктором
4) Уничтожение объекта сопровождается деструктором

# Инкапсуляция (ООП)

1) Объединение данных и функций в единую сущность

2) Ограничение доступа стороннего кода к деталям реализации

# Инкапсуляция (ООП)

1) Объединение данных и функций в единую сущность

2) Ограничение доступа стороннего кода к деталям реализации

В программировании в целом второе иногда называют **сокрытием** и формально отделяют от инкапсуляции

В ООП принято считать эти свойства едиными

# Инкапсуляция (ООП)

Позволяет приблизить код, описывающий тип данных, к модели корректных состояний этого типа

# class

Ключевое слово, отличающееся от struct, областью доступа по умолчанию:

- struct - public
- class - private

# Убедитесь, что вынесли с этой лекции

Причины возникновения ООП

Множество значений и корректные состояния. Разница.

Области доступа, сторонний код

Указатель this

Конструкторы и деструкторы

Операторы new/delete

Инкапсуляция

# Проверочные вопросы

1) Сколько можно описать деструкторов в одном классе?
2) Чему равен указатель this в коде метода?
3) Чем сокрытие отличается от инкапсуляции?
4) В какой момент вызываются деструкторы для локальных объектов?
5) А для объектов в динамическом классе памяти?

*

1) Как создать объект, все конструкторы которого описаны с private областью доступа?

# Q & A

# Что C++ не даёт

# Что C++ не даёт

**Безопасности**

# Что C++ не даёт

**Безопасности**

Утечки памяти, инвалидные указатели, выходы за границы выделенной памяти, … - всё это доступно

# Создание объекта в обход конструктора

```
void main() {
    void* ptr = malloc(sizeof(Stack));
    Stack* st = (Stack*) ptr;
    ...
}
```

# Public Морозов (формально запрещён)

```
struct Foo {
private:
    int x;
...
}
```

foo.h

```
#define private public
#include <foo.h>

void bar() {
    Foo f;
    f.x = 42; // ok
}
```

bar.cpp