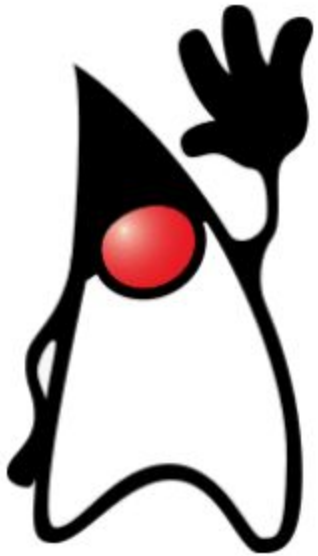
Дивный новый мир

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



Язык программирования Java

1996 - Java 1.0

Язык программирования Java

1996 - Java 1.0

2004 - Java 5.0 - определены основные концепции языка

Язык программирования Java

1996 - Java 1.0

2004 - Java 5.0 - определены основные концепции языка

2014 - Java 8.0 - шаг в сторону функционального программирования

Язык программирования Java

1996 - Java 1.0

2004 - Java 5.0 - определены основные концепции языка

2014 - Java 8.0 - шаг в сторону функционального программирования

2019 - Java 13.0 - текущая версия

Кто ответственен

1) Гослинг, Нотон, Шеридан - три инженера компании Sun Microsystems, уставшие от C++ (Нотон просто пытался уволиться)

2) С 2010-го года компанию Syn Microsystems поглотила компания Oracle

Ключевые идеи

1) Кросс-платформенность

2) Автоматическое управление памятью

3) Проверки времени исполнения (отсутствие UB)

Платформа

Процессор/OS, например - IntelX86/Windows

Платформа определяет Application Binary Interface (ABI) - набор правил, по которым программа будет исполняться на платформе

Компилятор собирает программу под определённую платформу

Кросс-платформенность

Желание того, чтобы программы работали одинаково на разных платформах

Кросс-платформенность

Желание того, чтобы программы работали одинаково на разных платформах

1) Инкапсуляция ABI-зависимого кода под ABI-независимым интерфейсом
2) Для каждой платформы подставляется своя версия кода

Кросс-платформенность

```
bool fileExists(char* name) {
#if defined(_WIN32)
	return GetFileAttributesA(name) != -1;
#elif defined(linux)
	return access(filename, 0) != 0;
#else
	unknown platform
#endif
}
```

Кросс-платформенность

Можно (и нужно) спрятать за библиотеками

```
boost::filesystem::exists(name)
```

И определить такие библиотеки, как обязательную частью языка

Кросс-платформенность

Для запуска программы на языке *L* на платформе *P* в любом случае нужен компилятор языка *L* для платформы *P*

Кросс-платформенность

Для запуска программы на языке *L* на платформе *P* в любом случае нужен компилятор языка *L* для платформы *P*

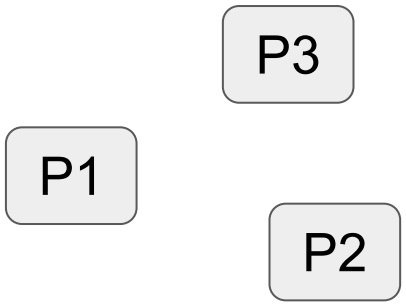
Точно?

Кросс-платформенность

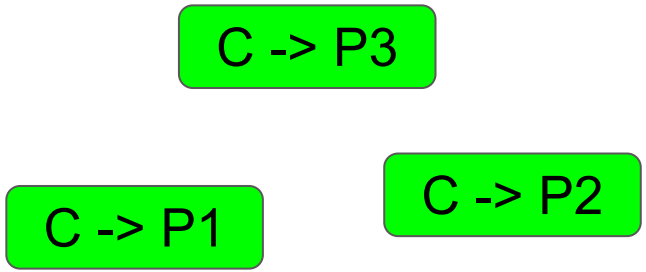
Для запуска программы на языке *L* на платформе *P* в любом случае нужен компилятор языка *L* для платформы *P*

Или интерпретатор языка *L*, написанный на любом языке, для которого есть компилятор для платформы *P*

Как реализовать новый язык L на {P1, P2, P3}?

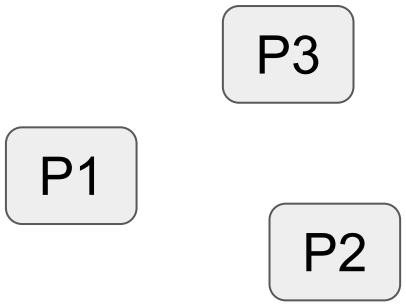


Платформы



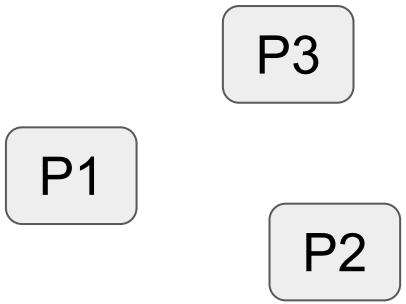
Компиляторы языка C

Как реализовать новый язык L на {P1, P2, P3}?

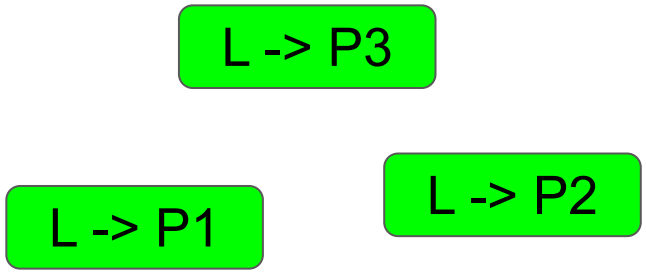


Платформы

Как реализовать новый язык L на {P1, P2, P3}?



Платформы

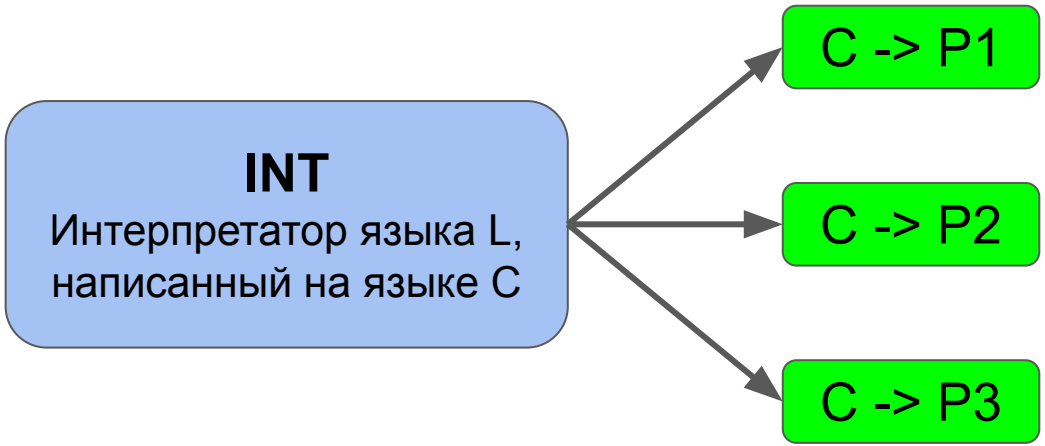


Компиляторы языка L

Как реализовать новый язык L на {P1, P2, P3}?

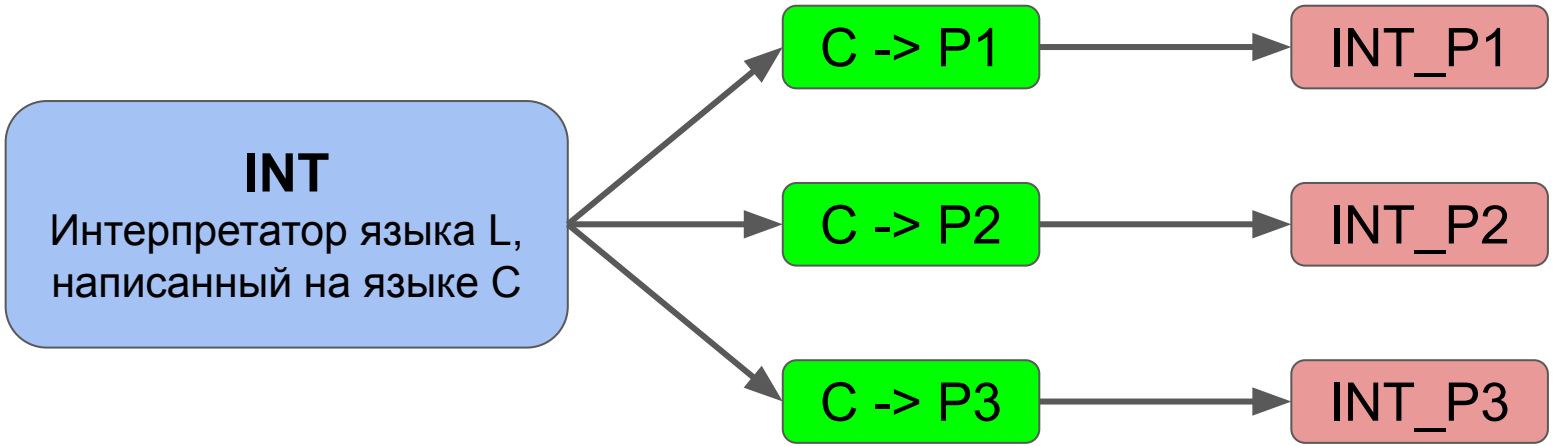
INT
Интерпретатор языка L,
написанный на языке C

Как реализовать новый язык L на {P1, P2, P3}?



Компиляторы языка C

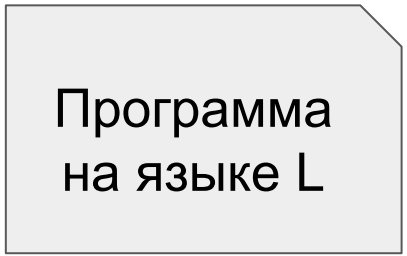
Как реализовать новый язык L на {P1, P2, P3}?



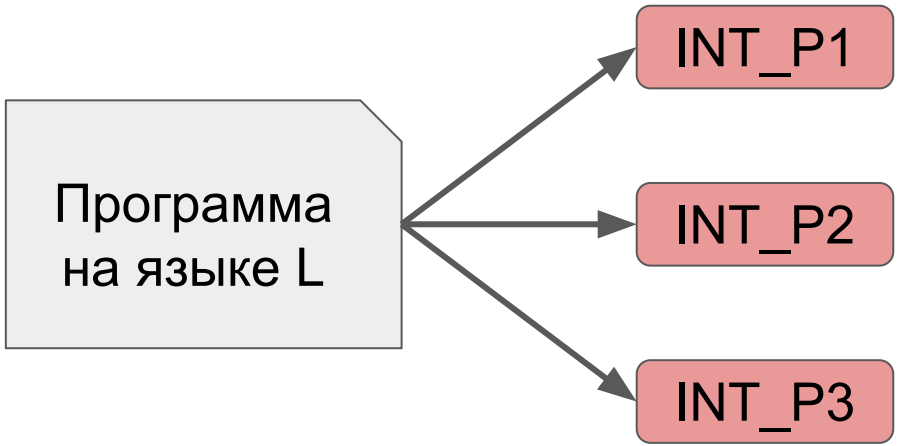
Компиляторы языка C

Одна и та же программа **INT** для разных платформ

Как реализовать новый язык L на {P1, P2, P3}?

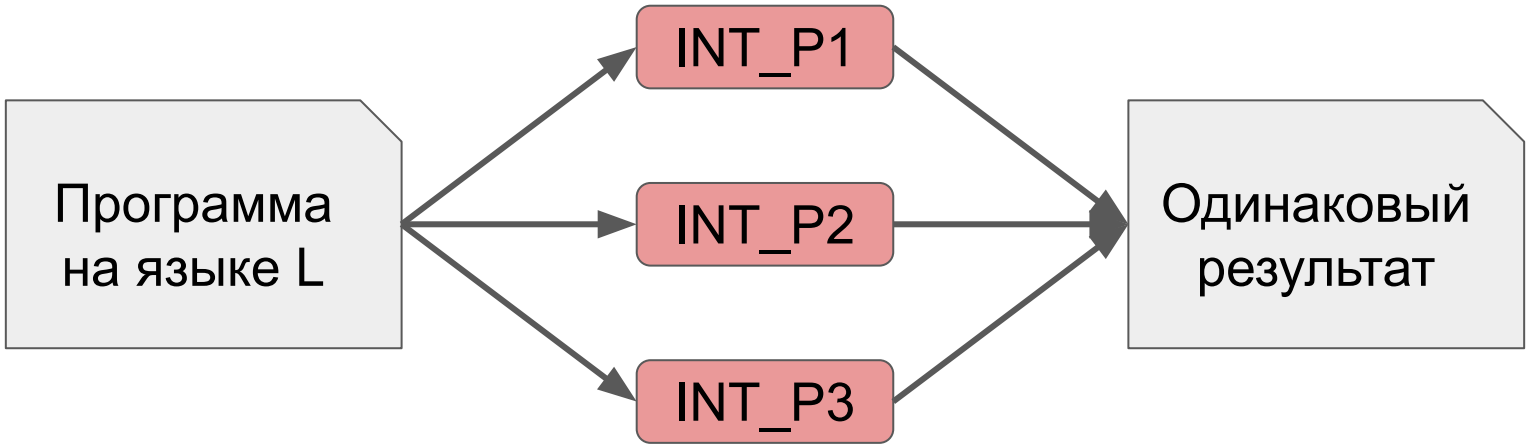


Как реализовать новый язык L на {P1, P2, P3}?



Одна и та же
программа **INT** для
разных платформ

Как реализовать новый язык L на {P1, P2, P3}?



Одна и та же
программа **INT** для
разных платформ

Кросс-платформенность в Java

Исходники Java компилируются в Java bytecode (специальное промежуточное представление)

Java bytecode интерпретируется на JVM (Java Virtual Machine) по строго определённой платформенно-независимой спецификации

Кросс-платформенность в Java

Идея не нова

1966 - язык BCPL, промежуточное представление - **O-code**

1976 - язык Pascal, промежуточное представление - **p-code**

Кросс-платформенность в Java

“Разработчики языка Java изучали исходные коды Оберона. Потом испортили Оберон синтаксисом Си и назвали получившееся словом Java”

Никлаус Вирт

Автоматическое управление памятью

Сборка мусора - объекты, созданные в динамическом классе памяти, удаляются автоматически, когда перестают быть нужными

Автоматическое управление памятью

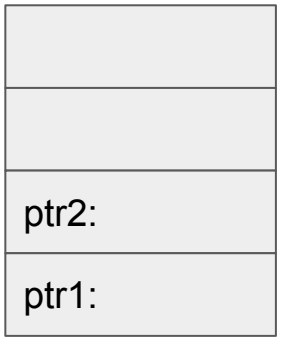
```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2;
        cout << ptr2->x;
        delete ptr2;
}
```

Автоматическое управление памятью

```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```

Автоматическое управление памятью

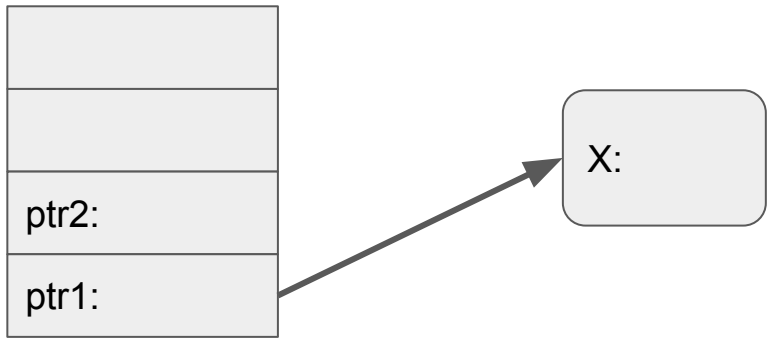
```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```



стек (локальные переменные)

Автоматическое управление памятью

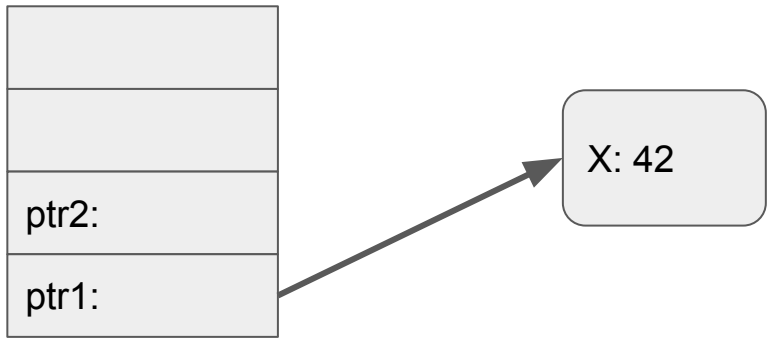
```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```



стек (локальные переменные)

Автоматическое управление памятью

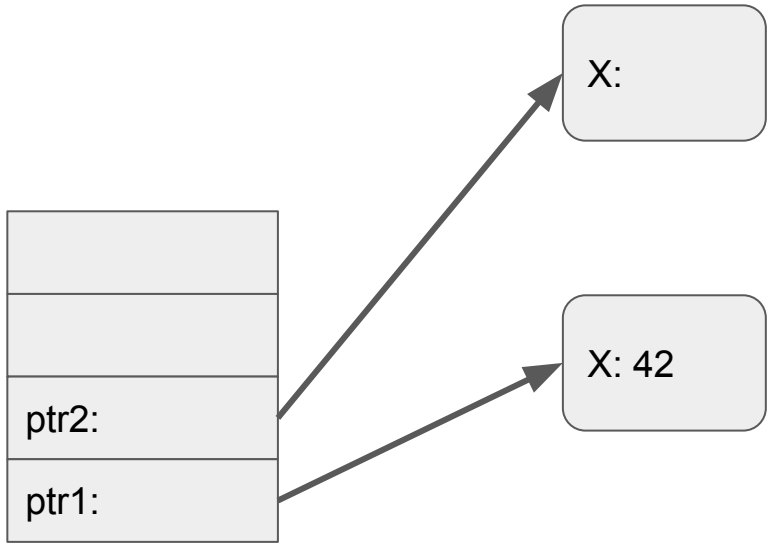
```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```



стек (локальные переменные)

Автоматическое управление памятью

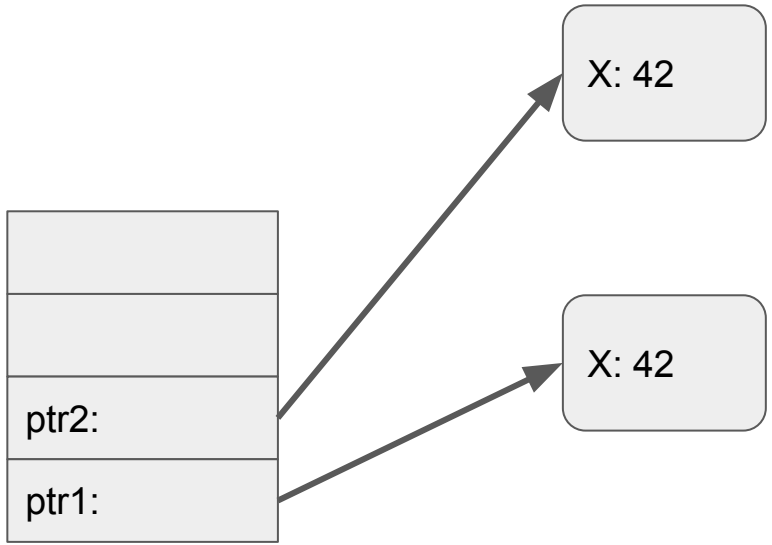
```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```



стек (локальные переменные)

Автоматическое управление памятью

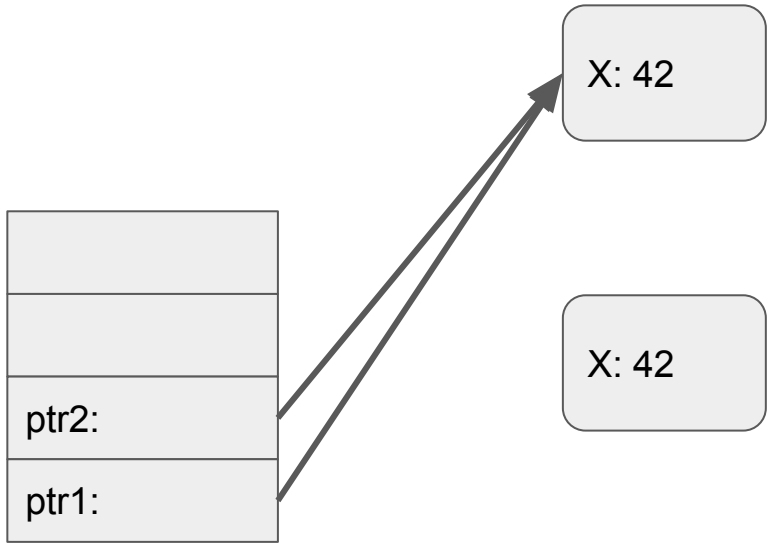
```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```



стек (локальные переменные)

Автоматическое управление памятью

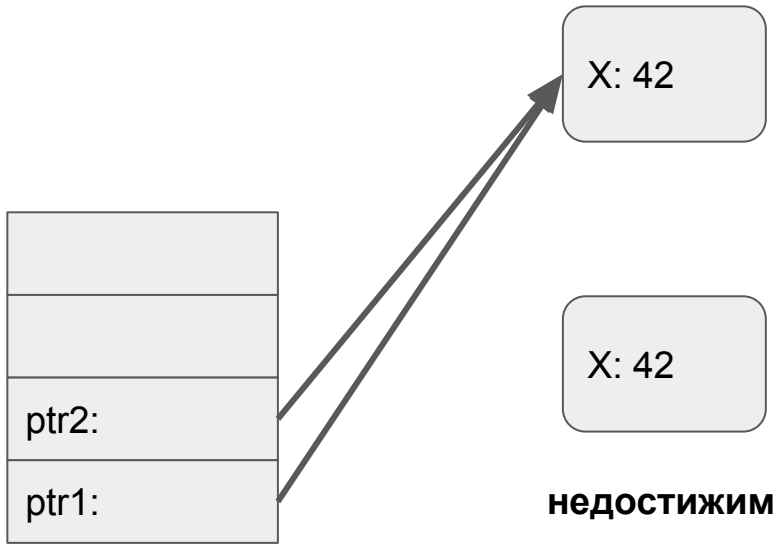
```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```



стек (локальные переменные)

Автоматическое управление памятью

```
void some() {
        Foo* ptr1 = new Foo();
        ptr1->x = 42;
        Foo* ptr2 = new Foo();
        ptr2->x = ptr1->x;
        ptr1 = ptr2; // утечка памяти!
        cout << ptr2->x;
        delete ptr2;
}
```



стек (локальные переменные)

Достижимость объектов

Объект **достижим**, если его адрес записан в глобальной или локальной переменной или в поле другого достижимого объекта

Достижимость объектов

Это свойство зависит от времени, причём изменяется только в одну сторону

- Недостижимый объект не может стать достижимым

Для языков с ручным управлением памятью недостижимый объект - это утечка памяти

- Для него уже нельзя вызвать free/delete, т.к. нет способов получить его адрес

Автоматическое управление памятью

Удаление недостижимого объекта гарантированно не приведёт к невалидным указателям в программе

ведь их на него нет, либо есть в также недостижимых объектах

Сборщик мусора (garbage collector, GC) это и делает

Счётчики ссылок

Идея - хранить в объекте (или в другом связанном с ним объекте) количество живых указателей на него. Если оно равно 0, удалять объект

Воплощено в C++ - smart pointers

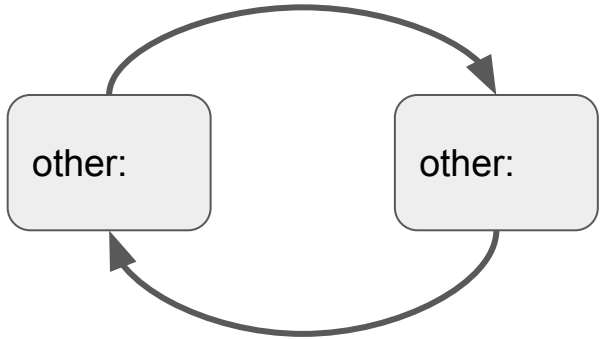
Возникают проблемы с циклическим мусором

Счётчики ссылок

```
struct Foo {
	Foo* other;
}

void some() {
	Foo* ptr1 = new Foo();
	Foo* ptr2 = new Foo();
	ptr1->other = ptr2;
	ptr2->other = ptr1;
	ptr1 = ptr2 = NULL;
}
```

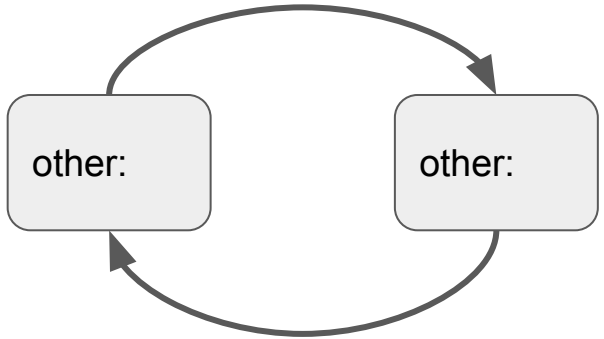
Счётчики ссылок

```
struct Foo {
        Foo* other;
}

void some() {
        Foo* ptr1 = new Foo();
        Foo* ptr2 = new Foo();
        ptr1->other = ptr2;
        ptr2->other = ptr1;
        ptr1 = ptr2 = NULL;
}
```



Счётчики ссылок

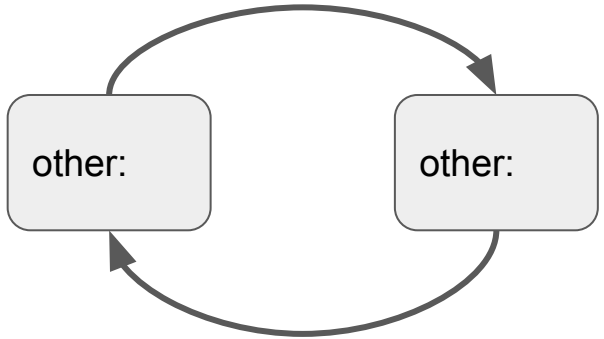
```
struct Foo {
        Foo* other;
}

void some() {
        Foo* ptr1 = new Foo();
        Foo* ptr2 = new Foo();
        ptr1->other = ptr2;
        ptr2->other = ptr1;
        ptr1 = ptr2 = NULL;
}
```



Счётчики ссылок для каждого показывают 1

Счётчики ссылок

```
struct Foo {
        Foo* other;
}

void some() {
        Foo* ptr1 = new Foo();
        Foo* ptr2 = new Foo();
        ptr1->other = ptr2;
        ptr2->other = ptr1;
        ptr1 = ptr2 = NULL;
}
```

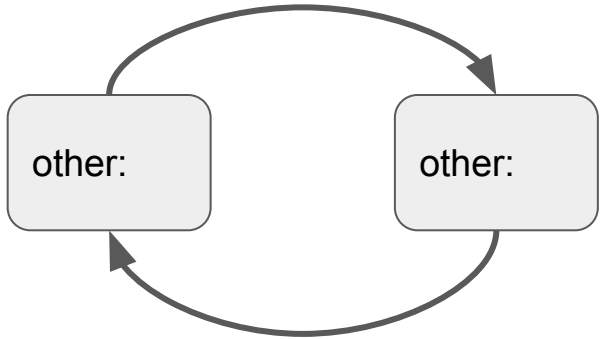


Счётчики ссылок для каждого показывают 1

Оба объекта недостижимы

Счётчики ссылок

```
struct Foo {
        Foo* other;
}

void some() {
        Foo* ptr1 = new Foo();
        Foo* ptr2 = new Foo();
        ptr1->other = ptr2;
        ptr2->other = ptr1;
        ptr1 = ptr2 = NULL;
}
```



Счётчики ссылок для каждого показывают 1

Оба объекта недостижимы в совокупности

Трассирующая сборка мусора

Граф объектов (вершины - объекты, дуги - указатели из полей одного объекта на другие)

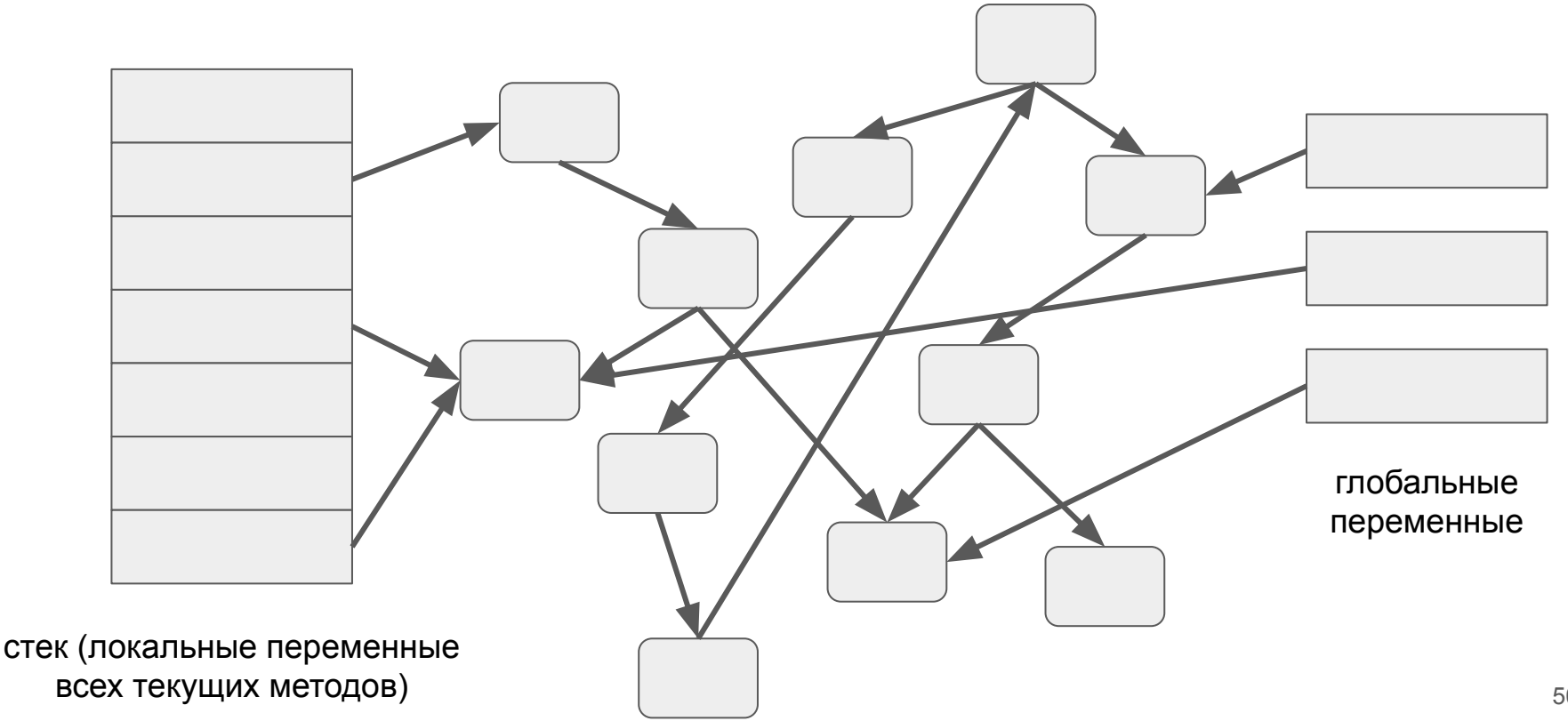
Корни - объекты, на которые есть указатели из локальных и глобальных переменных

Трассирующая сборка мусора

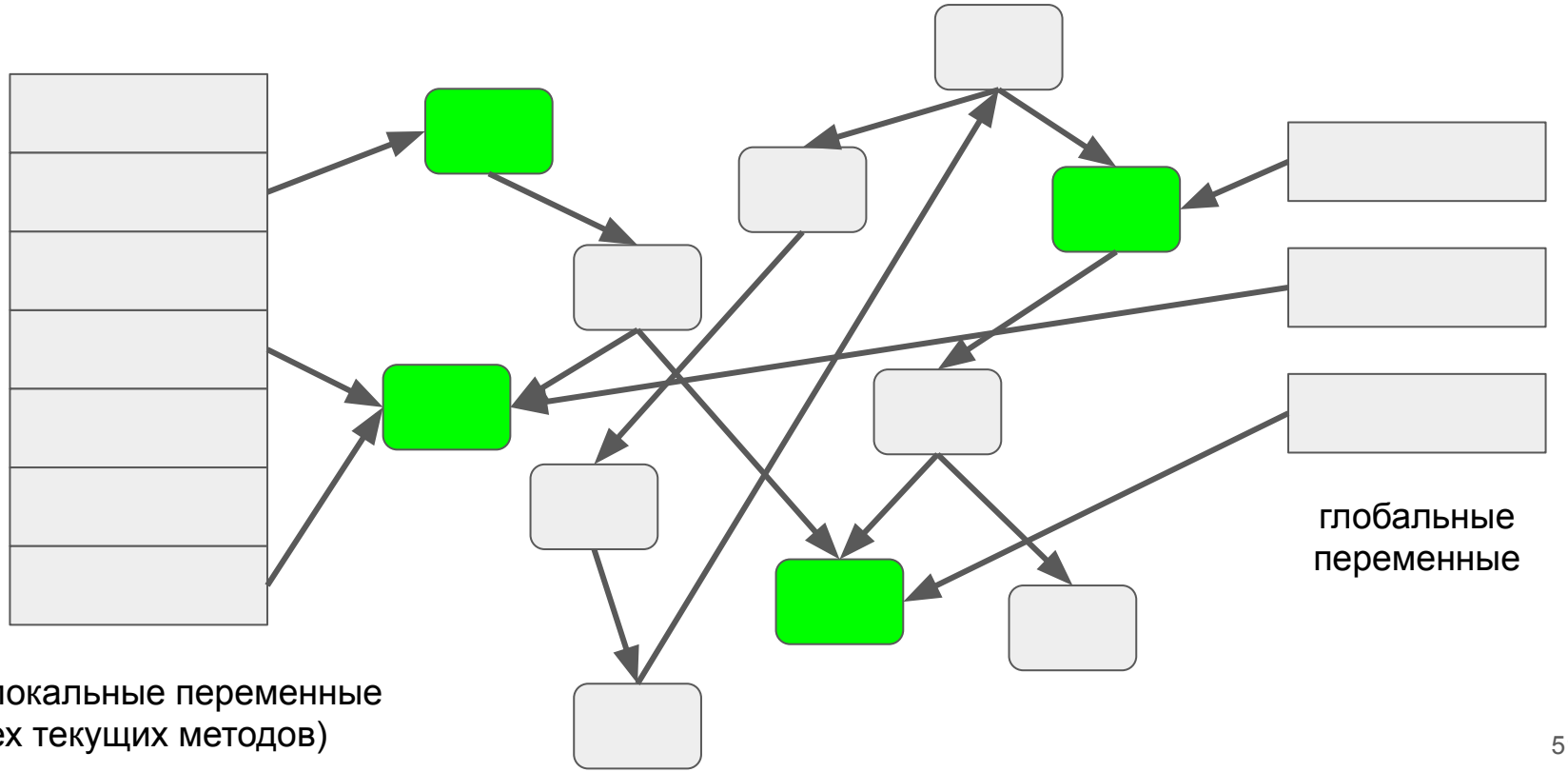
Обходим граф объектов от корней (mark)

Все, до которых не смогли добраться, считаем недостижимыми и удаляем (sweep)

Трассирующая сборка мусора

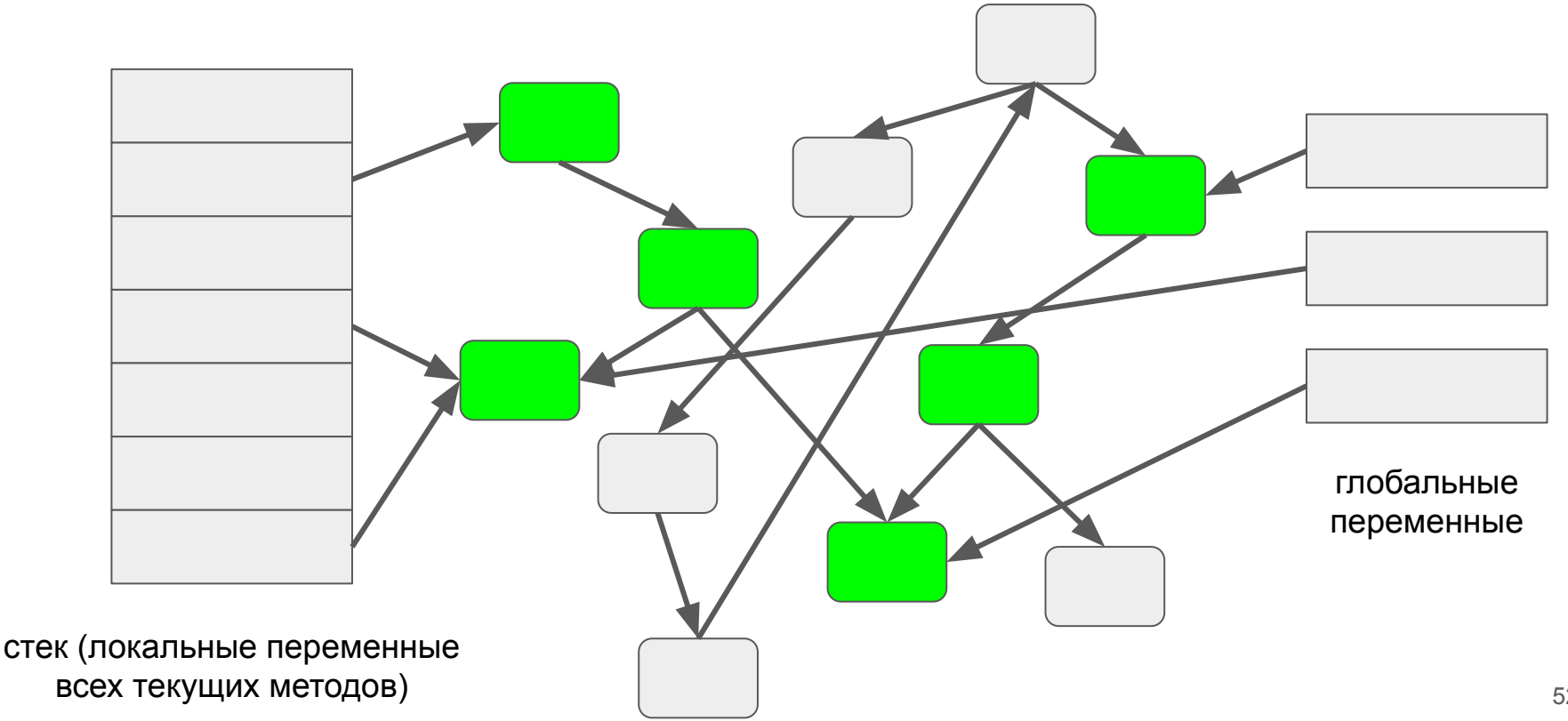


Трассирующая сборка мусора

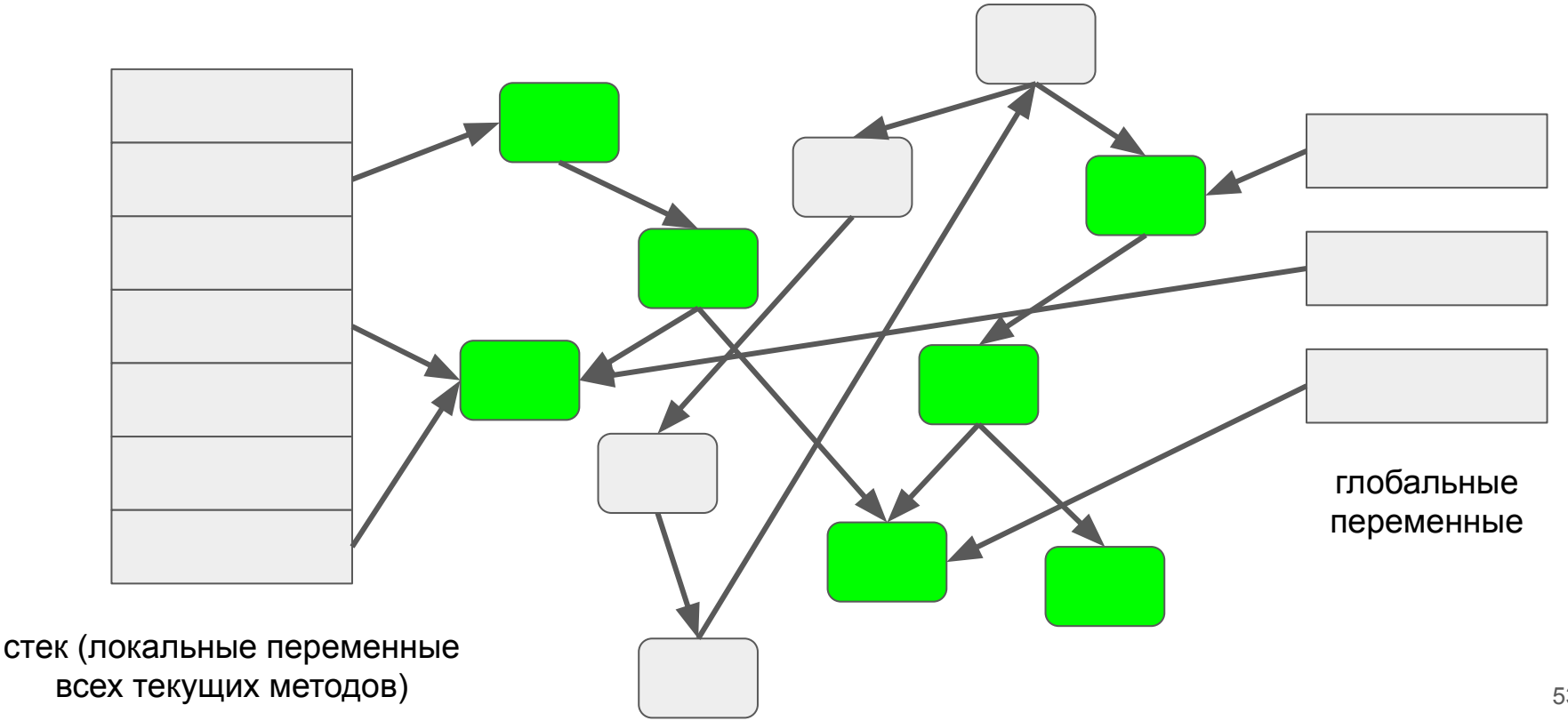


стек (локальные переменные
всех текущих методов)

Трассирующая сборка мусора

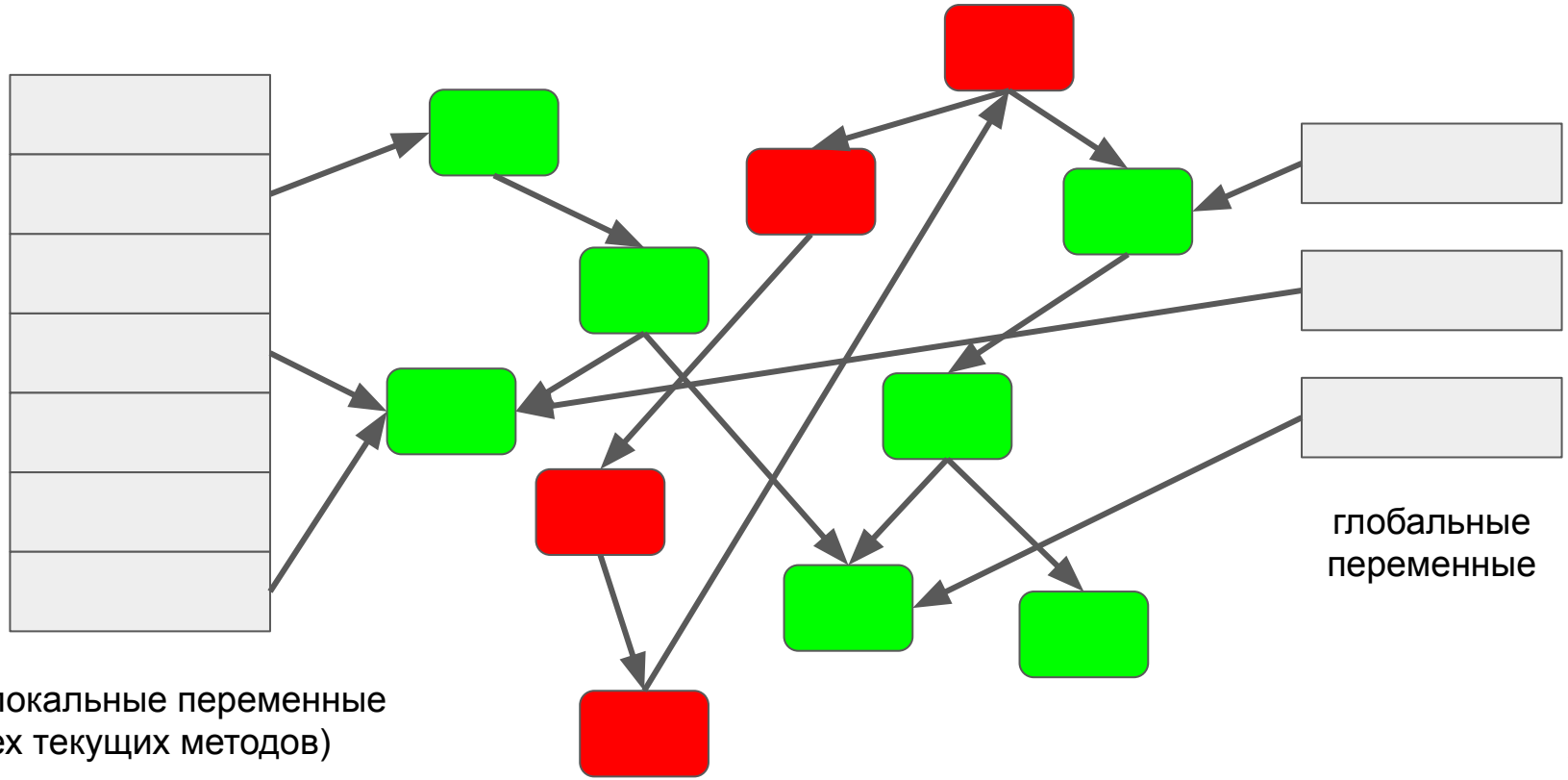


Трассирующая сборка мусора



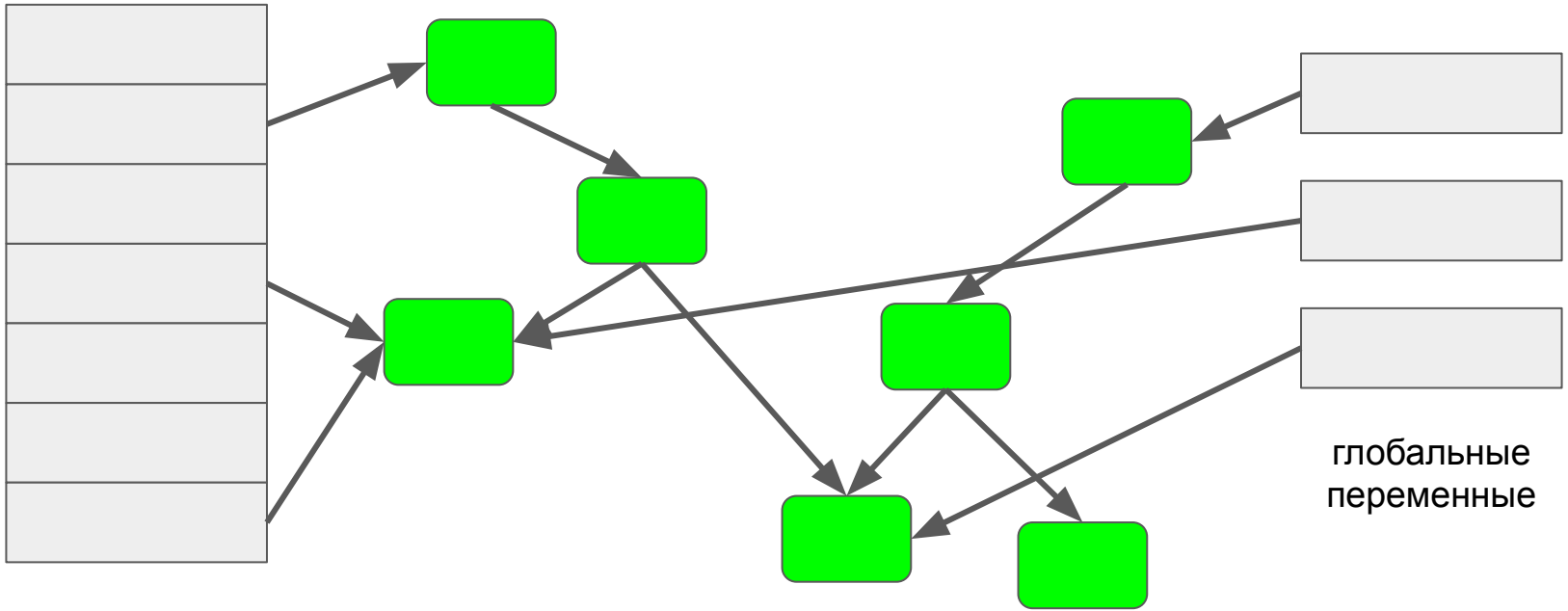
стек (локальные переменные
всех текущих методов)

Трассирующая сборка мусора



стек (локальные переменные
всех текущих методов)

Трассирующая сборка мусора



стек (локальные переменные
всех текущих методов)

Трассирующая сборка мусора

1) Требует видеть весь набор объектов - динамический класс памяти должен быть полностью подконтролен сборщику мусора

2) Требует типизации локальных, глобальных переменных и полей - нужно уметь отличать хранение адреса объекта от хранения числа

3) Требует приостановки программы для анализа стека/глобалов

В C/C++ TCM затруднена

В C/C++ TCM затруднена

```
void some() {
        Foo* ptr = new Foo();
        int X = (int) ptr; // intel x86, MSVC
}
```

В C/C++ ТСМ затруднена

```
void some() {
        Foo* ptr = new Foo();
        int X = (int) ptr; // intel x86, MSVC
        int A = X % 759;
        int B = X / 759;
}
```

В C/C++ ТСМ затруднена

```
void some() {
        Foo* ptr = new Foo();
        int X = (int) ptr; // intel x86, MSVC
        int A = X % 759;
        int B = X / 759;
        … // A - записали в файл, B отправили по почте доверенному лицу
        ptr = NULL;
}
```

В C/C++ ТСМ затруднена

```
void some() {
        Foo* ptr = new Foo();
        int X = (int) ptr; // intel x86, MSVC
        int A = X % 759;
        int B = X / 759;
        … // A - записали в файл, B отправили по почте доверенному лицу
        ptr = NULL;
        // можно ли считать объект недостижимым?
}
```

В C/C++ ТСМ затруднена

Разумеется, можно воздержаться от подобных хаков

В C/C++ TCM затруднена

Разумеется, можно воздержаться от подобных хаков

И ещё от адресной арифметики

В C/C++ TCM затруднена

Разумеется, можно воздержаться от подобных хаков

И ещё от адресной арифметики

И убедиться, что все библиотеки, которыми вы пользуетесь, сделали то же

В C/C++ ТСМ затруднена

Разумеется, можно воздержаться от подобных хаков

И ещё от адресной арифметики

И убедиться, что все библиотеки, которыми вы пользуетесь, сделали то же

В Java так и поступили (адресная арифметика не нужна примерно в 100% прикладных программ)

Управление памятью в Java

Указатели не совместимы с числами

Объект можно создать только в динамической памяти и нельзя удалить

- В том числе передача / возврат объекта только по указателям

JVM реализует сборку мусора

Управление памятью в Java

Деструкторы, конструкторы копий, операторы присваивания, правило трёх, RAII-идиома, передача по ссылке, по значению или по указателю, деструкция при выбросе исключения, запрет на выброс исключения из деструктора

Управление памятью в Java

~~Деструкторы, конструкторы копий, операторы присваивания, правило трёх, RAII-идиома, передача по ссылке, по значению или по указателю, деструкция при выбросе исключения, запрет на выброс исключения из деструктора~~

Не нужны

Управление памятью в Java

~~Деструкторы, конструкторы копий, операторы присваивания, правило трёх, RAII-идиома, передача по ссылке, по значению или по указателю, деструкция при выбросе исключения, запрет на выброс исключения из деструктора~~

Не нужны (при этом память чистить вручную не надо, она не утекает и не возникает невалидных указателей)

Управление памятью в Java

~~Деструкторы, конструкторы копий, операторы присваивания, правило трёх, **RAII-идиома**, передача по ссылке, по значению или по указателю, деструкция при выбросе исключения, запрет на выброс исключения из деструктора~~

Не нужны (ну, почти)

Управление памятью в Java

RAII-идиома остаётся актуальной для ресурсов, не являющихся объектами языка - файлы, сокеты, прочие объекты окружения

Для её поддержки есть специальные средства - **финализаторы** и **try-with-resources**

Управление памятью в Java

Не всё идеально:

1) Так как объекты передаются по указателям, возникает риск мутаций
2) Порядок и момент вызова финализаторов не определён
3) GC не избавляет от архитектурных утечек памяти (неочищающиеся кэши, неадекватные требования памяти и т.п.)

Автоматическое управление памятью

Идея не нова

1959 - функциональный язык Lisp

Стиль функциональных языков таков, что операторы создают множество временных объектов

Автоматическое управление памятью

```
Foo A, B, C, D, E;
…
E = (A + B) * (C + D);
```

Представьте себе, что оператор + возвращал бы указатель на объект в динамическом классе

Автоматическое управление памятью

Многие программисты не замечают плюсов от перехода на языки со сборкой мусора, потому что они и в C/C++ не заморачиваются с удалением каких-то там объектов

Проверки времени исполнения

1) Разыменование null (единственный вид невалидного указателя) - выброс NullPointerException
2) Выход за границы массива - выброс IndexOutOfBoundsException
3) Каст от предка к неправильному потомку - выброс ClassCastException
4) Нехватка памяти - выброс OutOfMemoryError

И так далее

Проверки времени исполнения

JVM интерпретирует команды программы, выполняя все эти проверки

UB пропадает, как явление

Управляемый код

Определение от Microsoft - код, исполняющийся под контролем .NET

CLI (Common Intermediate Language) - аналог Java bytecode

CLR (Common Language Runtime) - аналог JVM

Реализации - .NET, Mono, Portable.NET

Управляемый код

Общие признаки:

1) Поддержка виртуальной машиной
2) Управляемая динамическая память
3) Проверки времени исполнения
4) Отсутствие UB

Управляемый код

Общие признаки:

1) Поддержка виртуальной машиной
2) Управляемая динамическая память
3) Проверки времени исполнения
4) Отсутствие UB

Ни один язык программирования нельзя назвать неуправляемым - это свойство не языка, а способа его реализации

Java тормозит!!!

Java тормозит!!!

От создателей “get-set методы тормозят” и “++i быстрее, чем i++“

Ничего не бывает бесплатным

1) Интерпретация - это очень медленно
2) GC - это накладные расходы в виде пауз и метаинформации
3) Проверки времени исполнения выполняются при всех операциях

Ничего не бывает бесплатным

1) Интерпретация - это очень медленно
2) GC - это накладные расходы в виде пауз и метаинформации
3) Проверки времени исполнения выполняются при всех операциях

Но авторы компиляторов/JVM в курсе

JIT-компиляция

В время интерпретации программы (just in time) JVM компилирует её куски в машинный код конкретного процессора и потом исполняет их вместо интерпретации

JIT-компиляция

В время интерпретации программы (just in time) JVM компилирует её куски в машинный код конкретного процессора и потом исполняет их вместо интерпретации

Мало того, что Java тормозит, она ещё и компилировать что-то будет

JIT-компиляция

Для компиляции выбираются только **горячие** методы - те, которые действительно участвуют в исполнении программы

JIT-компиляция

Для компиляции выбираются только **горячие** методы - те, которые действительно участвуют в исполнении программы

Правило 80/20 - 80% времени работает 20% кода

На практике пропорция намного драматичнее

JIT-компиляция

Горячие методы могут быть скомпилированы со **спекуляциями** - агрессивными предположениями компилятора, возникающими при анализе истории исполнения программы

JIT-компиляция

Горячие методы могут быть скомпилированы со **спекуляциями** - агрессивными предположениями компилятора, возникающими при анализе истории исполнения программы

Например, виртуальный вызов может на практике всегда уходить в одну и ту же реализацию, тогда JIT-компилятор заинлайнит её под проверкой и хорошо соптимизирует наиболее частый случай

JIT-компиляция

Множество проверок времени исполнения может быть удалено компилятором в результате анализа

Объекты, создающиеся в динамической памяти, компилятор может при некоторых условиях разместить на стеке

Строгий язык - сильнее оптимизации

```
struct Foo { int x; };
struct Bar { float y; };

int foo(Foo* f, Bar* b) {
        int r = 0;
        for (int i = 0; i < N; i++) {
                r += f.x;
                b.y = (float)r;
        }
        return r;
}
```

Строгий язык - сильнее оптимизации

```
struct Foo { int x; };
struct Bar { float y; };

int foo(Foo* f, Bar* b) {
        int r = 0;
        int tmp = f.x;
        for (int i = 0; i < N; i++) {
                r += tmp;
                b.y = (float)r;
        }
        return r;
}
```

Допустима ли такая оптимизация в C++?

Строгий язык - сильнее оптимизации

```
struct Foo { int x; };
struct Bar { float y; };

int foo(Foo* f, Bar* b) {
        int r = 0;
        int tmp = f.x;
        for (int i = 0; i < N; i++) {
                r += tmp;
                b.y = (float)r;
        }
        return r;
}
```

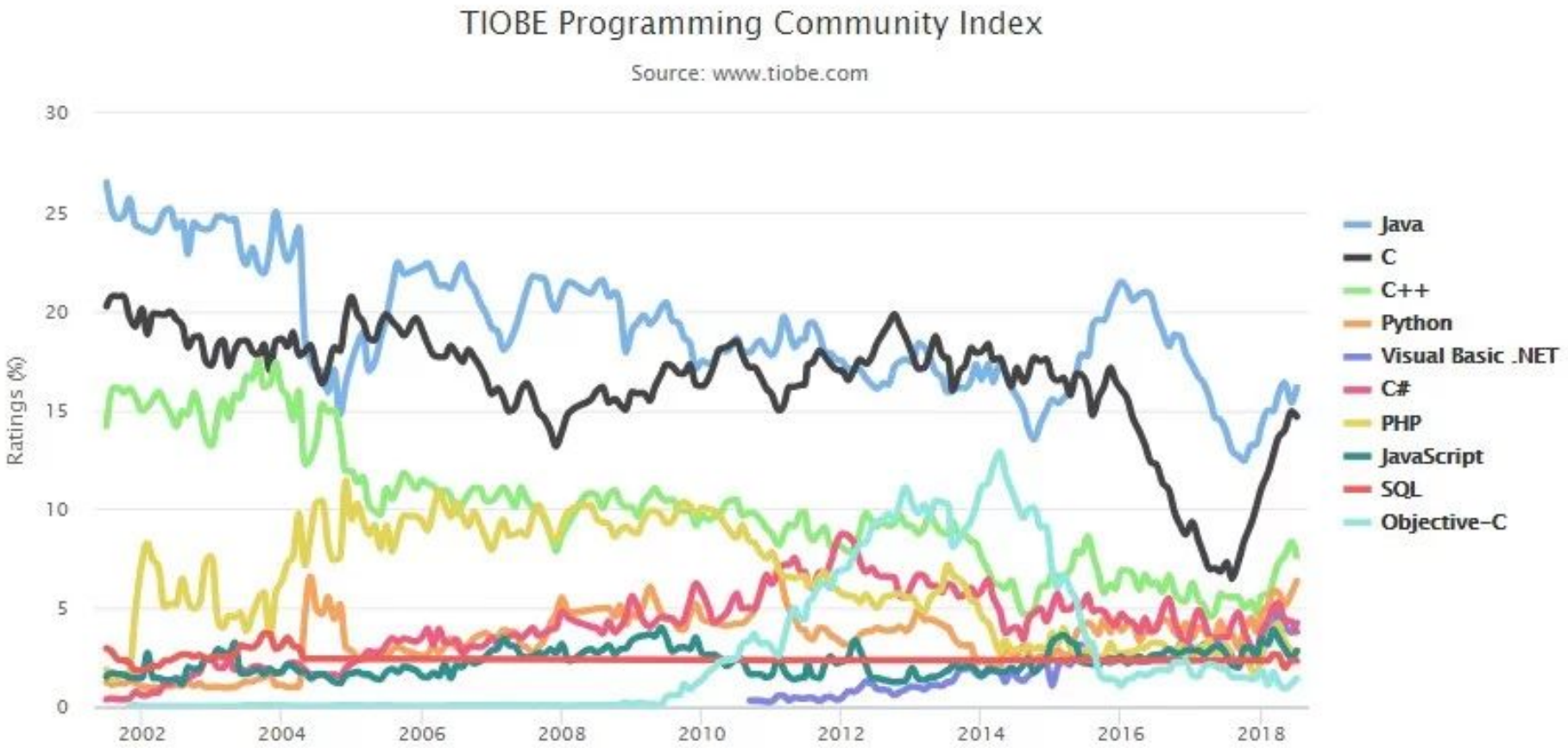
```
Foo* ptr = new Foo();
foo(ptr, (Bar*)ptr);
```

Допустима ли такая оптимизация в C++?

Java тормозит

Реальность:

- Есть бенчмарки, на которых C++ быстрее
- Есть бенчмарки, на которых Java быстрее


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
Java
C
C++
Python
Visual Basic .NET
C#
PHP
JavaScript
SQL
Objective-C

Убедитесь, что вынесли с этой лекции

Кросс-платформенность - средства реализации

Автоматическое управление памятью, сборка мусора

Особенности модели памяти языка Java

Проверки времени исполнения

Managed код и языки

Вопросы производительности, JIT-компиляция

Проверочные вопросы

1) С помощью каких конструкций пишется кросс-платформенный код?
2) Что такое интерпретация, и какие у неё плюсы?
3) Что такое счётчики ссылок?
4) Что такое трассирующая сборка мусора?
5) Возможно ли сборка мусора в C++?
6) Что защищает управляемый код от UB?
7) Какие плюсы по производительности есть у языка Java?

Проверочные вопросы*

1) Реализуйте сборку мусора счётчиками ссылок на C++

2) *** - Реализуйте сборку мусора (для объектов фиксированного набора классов) трассирующим сборщиком мусора на C++, в предположении, что программа не будет содержать адресной арифметики

Q & A