Автоматическое управление памятью

Работа с памятью – один из основных источников ошибок в программах

- Двойное освобождение
- Доступ к освобождённой памяти
- Выход за границы блока памяти
- Утечка памяти (блоки памяти уже не используются, но не освобождаются)
- Ошибки совместного доступа к памяти из разных потоков исполнения
- И тд.

Основные способы борьбы с такими ошибками

- Ограничения на работу со ссылочными типами на уровне семантики языка:
  - Ada – области видимости типов, ссылочные типы не могут выходить за область видимости основного типа. При грамотном использовании локальных определений типов и локальных определений ссылочных типов, исключается проблема “висячих” ссылок.
  - Rust – рассчёт времён жизни ссылок, borrowing checker (проверка невозможности одновременного доступа на изменение данных по ссылке из разных мест программы). Это позволяет полностью исключить класс проблем связанных со случайным изменением данных, с висячими ссылками и тд. При работе в рамках “безопасного” подмножества языка, основная часть проблем работы с памятью исключена, актуальной остаётся только проблема утечек памяти.
    Но у программиста всегда есть возможность перейти в “небезопасный” режим и отстрелить себе ногу ☺

Основные способы борьбы с такими ошибками

- Ликвидация ссылочных типов на уровне языка полностью, без прибегания к “сборке мусора”:
  - Например, интересный язык ParaSail, разрабатываемый одним из основных архитекторов языка Ada – Tucker Taft, вообще не содержит механизмов работы с указателями, там используется интересный вариант работы с регионами памяти, объекты могут быть рекурсивно определены с использованием типов Optional.
    В этом языке ликвидированы все основные проблемы работы с памятью, управление памятью полностью автоматическое и детерминированное, нет никаких проблем, присущих “сборщикам мусора”.
    Из минусов – пока в состоянии экспериментальной разработки, из-за серьёзной ориентированности на параллельные вычисления, производительность обычных программ оставляет желать лучшего.

Основные способы борьбы с такими ошибками

- Умные указатели (полуавтоматическое управления памятью - подсчёт ссылок)
  - Широко используемый подход во многих ОО языках без автоматического управления памятью, например C++
  - Позволяют управлять владением, подсчётом ссылок и пр, при неиспользовании объекта (счётчик ссылок нулевой, например) автоматически вызывают деструктор и освобождают память

Основные способы борьбы с такими ошибками

- Полностью автоматическое управления на основе “сборки мусора”
  - Используется во многих популярных языках: Java, Python, JS, Haskell, Ocaml, Perl, PHP, Lua, C#, F# и пр
  - У программиста нет головной боли относительно корректности работы с памятью совсем
  - Зато появляется другая головная боль ☺, например при управлении временем жизни объектов, паузы при выполнении программы и тд.

Частичное решение проблемы – “умные указатели” (тоже относятся к виду автоматического управления памятью)

- Позволяют управлять владением (uniq_ptr)
- Подсчитывают ссылки и позволяют автоматически освобождать неиспользуемую память (shared_ptr)
- Позволяют отслеживать достижимость объекта (weak_ptr)
- И другие, более специализированные
- В C++ основа реализации умных указателей – идиома RAII и move/copy семантика

RAII

- Эта идиома использует особенности реализации ООП в C++: наличие конструкторов и деструкторов
- При создании объекта вызывается соответствующий конструктор и объект начинает владеть каким-либо ресурсом
- При выходе управления за пределы видимости объекта (“{ … }” блока кода), вызывается деструктор объекта, который деинициализирует ресурс и освобождает его.
- Часто используется для файлов, мьютексов и тд.
- Ну и для smart pointers, конечно же.

Move/copy семантика

- В C++ реализуется на уровне конструкторов и операторов присваивания
- Позволяет управлять состоянием старого объекта при создании/присваивании нового

```
struct A {
A(const A&) = default; // конструктор копирования
A(A&&)      = default; // конструктор перемещения, старый
                       // объект, как правило, становится невалидным
A& operator=(const A&) = default; // копирующее присваивание
A& operator=(A&&)      = default; // перемещающее присваивание
~A()                   = default; // деструктор
}
```

Unique_ptr – пример реализации

```
#include <assert>

template<typename T>
class uniqPtr {
   T* ptr{nullptr};
public:
   uniqPtr() = default;
   uniqPtr(const uniqPtr& other) = delete; // копировать нельзя!
   uniqPtr(uniqPtr&& other) : ptr{other.ptr} { other.ptr = nullptr; }
   uniqPtr& operator=(const uniqPtr& other) = delete; // копирующее присваивание тоже запрещено!
   uniqPtr& operator=(uniqPtr&& other) {
      delete ptr;
      ptr = other.ptr;
      other.ptr = nullptr;
      return *this;
   }
   ~uniqPtr() = { delete ptr; }
   T& operator*() { assert(is_valid()); return *ptr; }
   bool is_valid() { return ptr != nullptr; }
};
```

```
#include <assert>
#include <utility>

template<typename T>
class uniqPtr {
   T* ptr{nullptr};
public:
   template<typename...Args>
   uniqPtr(Args&&..args) { ptr = new T(std::forward<Args>(args)...); }
   uniqPtr() = default;
   uniqPtr(const uniqPtr& other) = delete; // копировать нельзя!
   uniqPtr(uniqPtr&& other) : ptr{other.ptr} { other.ptr = nullptr; }
   uniqPtr& operator=(const uniqPtr& other) = delete; // копирующее присваивание тоже запрещено!
   uniqPtr& operator=(uniqPtr&& other) {
      delete ptr; ptr = other.ptr; other.ptr = nullptr;
      return *this;
   }
   ~uniqPtr() = { delete ptr; }
   T& operator*() { assert(is_valid()); return *ptr; }
   bool is_valid() { return ptr != nullptr; }
};
```

Пример использования

```
void g() {
    uniqPtr<MyClass> p{1,2,3};
    f(std::move(p));
    assert(!p.is_valid());
}
```

Создание объекта MyClass и сохранение указателя на него внутри p

template<typename…Args> uniqPtr(Args&&..args)

Перемещение указателя в функцию f

uniqPtr(uniqPtr&& other)

P после перемещения становится невалидный

```
void f(uniqPtr<MyClass>&& ptr) {
    (*ptr).DoSomething();
}
```

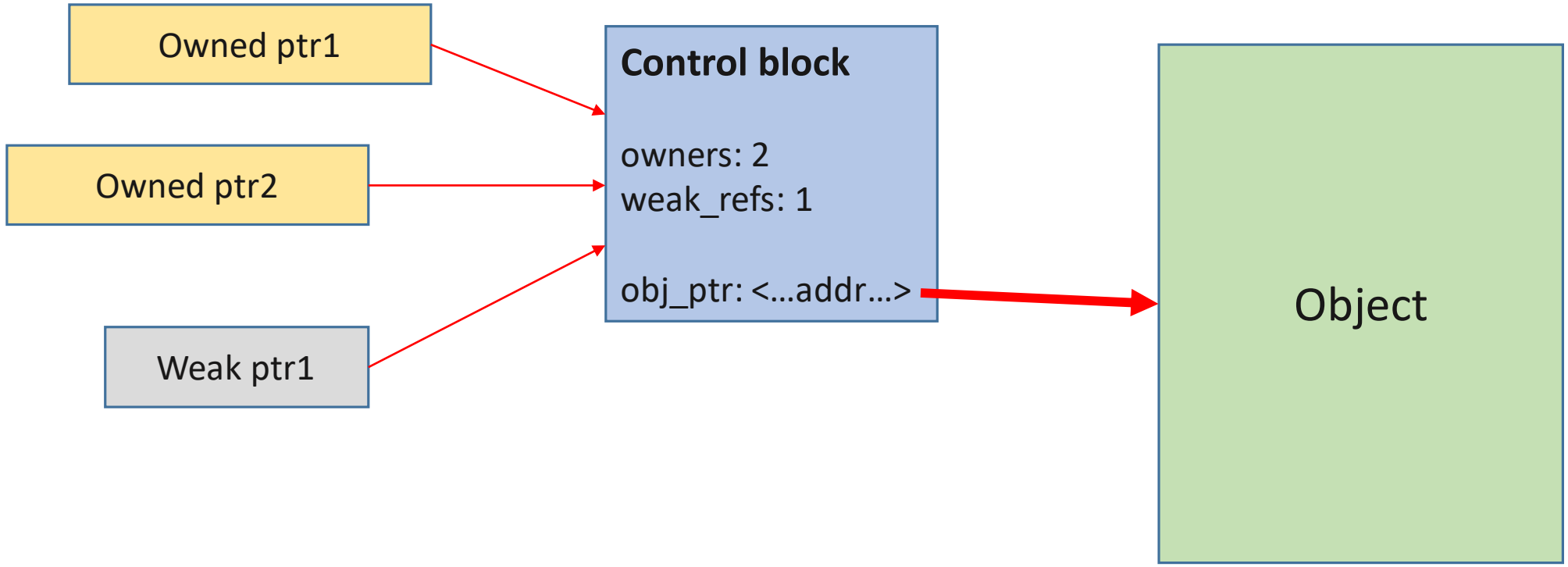
Теперь f эксклюзивно владеет указателем

При выходе из f вызывается деструктор указателя и соответственно удаление объекта MyClass

~uniqPtr() = { delete ptr; }

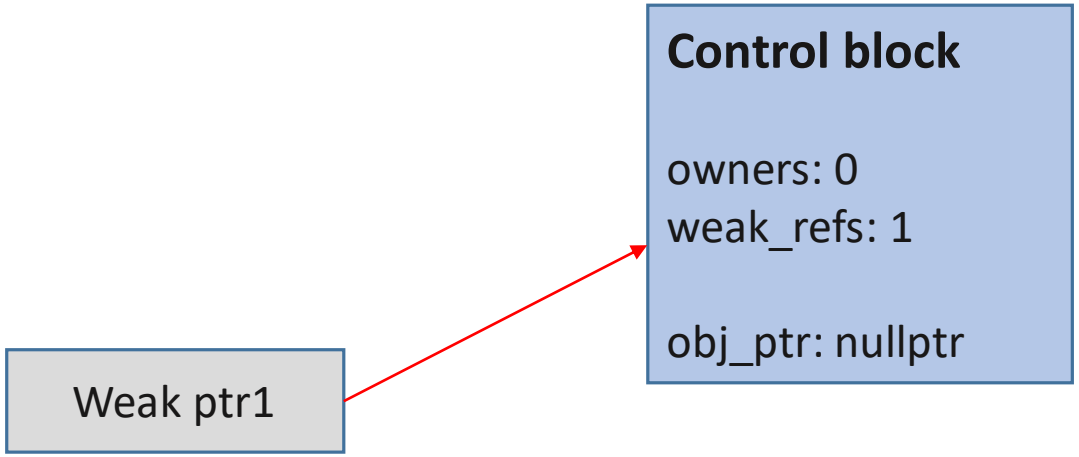
Shared_ptr

- Содержит счётчик ссылок владельцев, при обнулении которого объект удаляется.



Shared_ptr

- Control block существует, пока есть невладеющие указатели
- Это нужно для того, чтобы можно было проверять валидность невладеющих указателей



Полезные ссылки

- Основные идиомы C++: More C++ idioms
- Неплохое видео с объяснением внутренностей умных указателей и основных проблем с ними: C++ Smart Pointers - Usage and Secrets - Nicolai Josuttis
- Глава 7 из книги Александреску: chapter from Modern C++ Design
- cppreference.com: Dynamic memory management

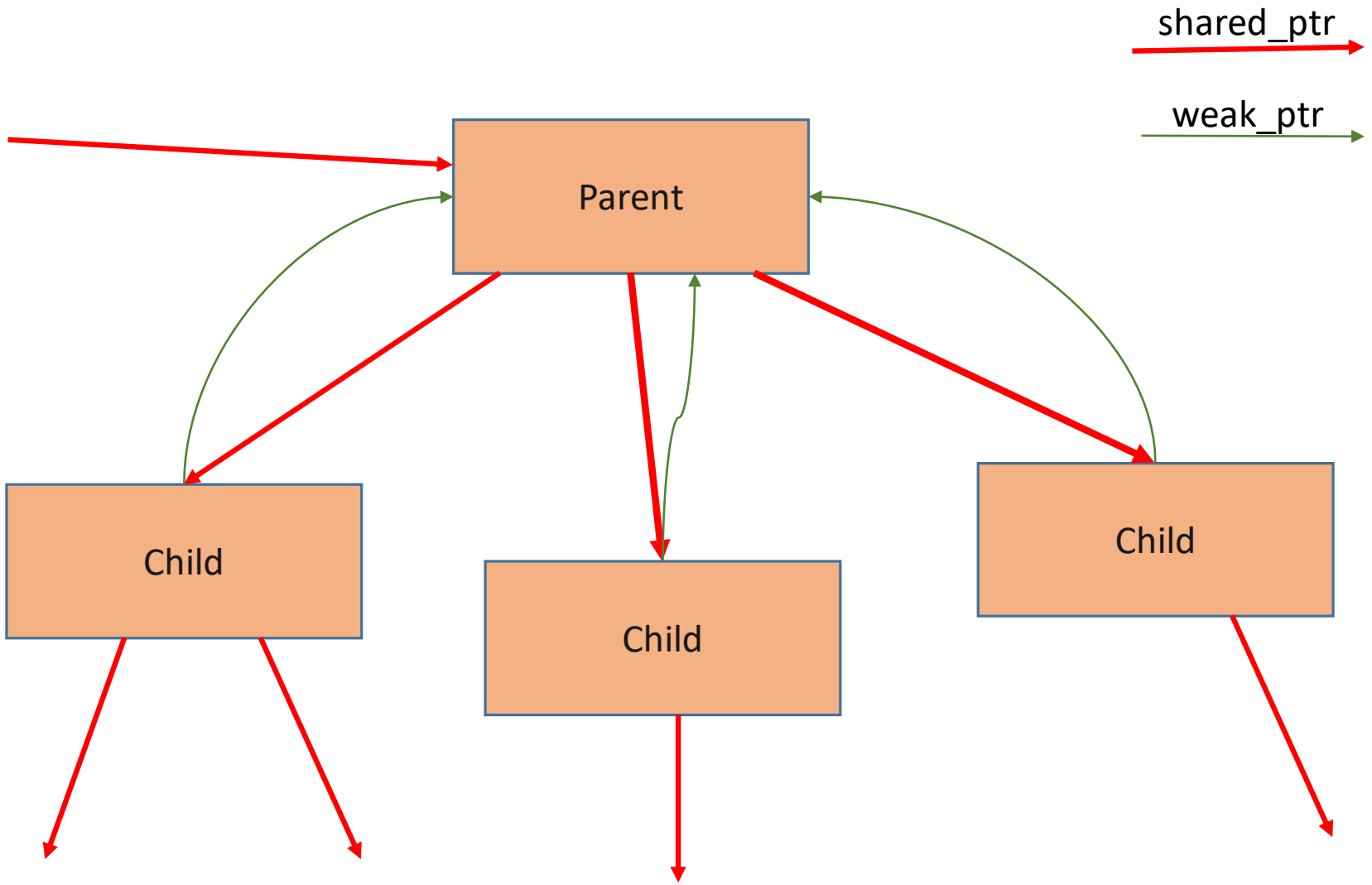
Основные проблемы умных указателей

- Основная проблема по сути одна: циклические структуры данных.
- Некоторые циклические структуры можно сделать на умных указателях, если грамотно выбрать weak_ptr/shared_ptr:
  - Дерево со ссылкой на родительский узел может быть огранизовано так: из узла на дочерние узлы указывают shared_ptr, а на родительский узел указывает weak_ptr. Тогда мы можем хранить shared_ptr указатели на поддеревья, а неиспользуемые части дерева будут автоматически освобождены
  - Дважды-связанный список: next – shared_ptr, prev – weak_ptr. Тогда мы можем хранить и передвигать shared_ptr указатель на новую голову списка, а неиспользуемые узлы будут автоматически освобождаться

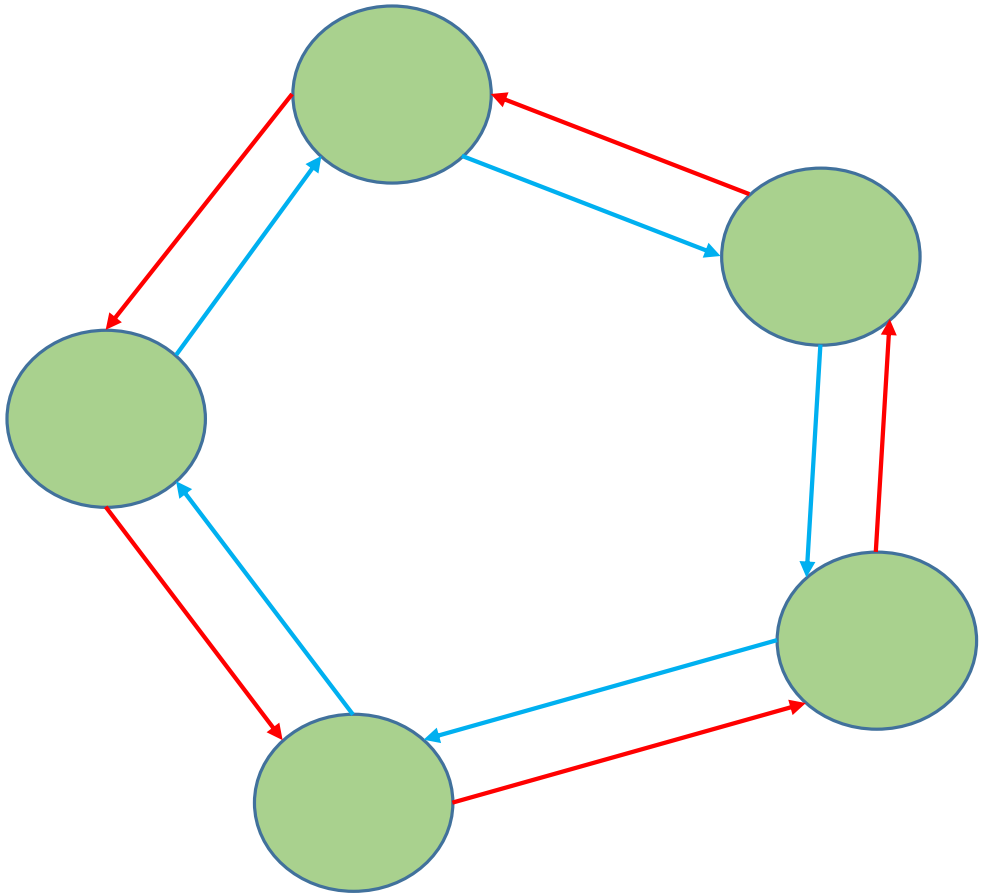
Основные проблемы умных указателей

- Одна из проблем заключается в том, что некоторые структуры данных могут иметь неочевидные циклы, и тогда легко ошибиться с выбором shared_ptr/weak_ptr, что приведёт либо к утечкам памяти, либо к преждевременному освобождению данных.
  Как пример – это сложные деревья Abstract Syntax Tree, которые используются в компиляторах, всевозможных парсерах и тд.
- Другая проблема – это то, что некоторые циклические структуры данных невозможно сделать на умных указателях в принципе.

Пример структуры с циклами, которая реализуется на умных указателях



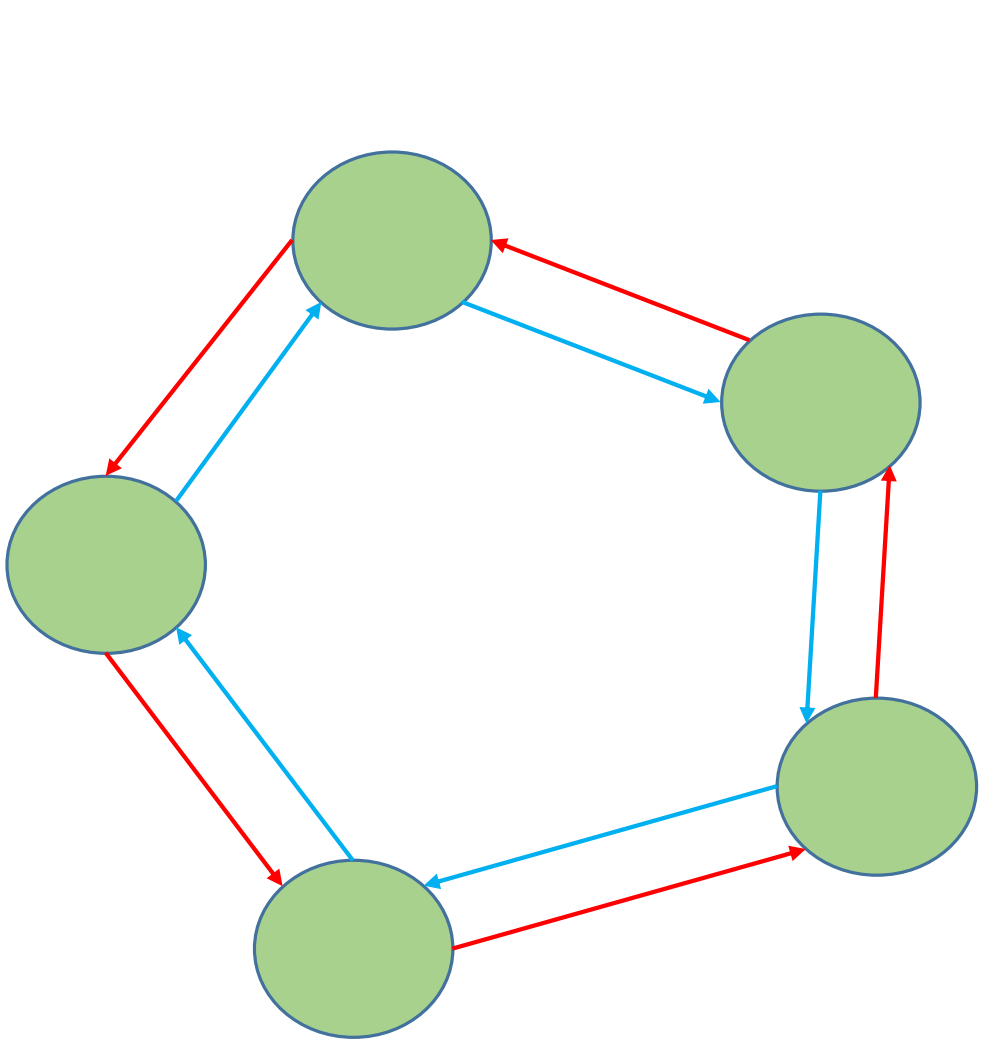
Циклический дважды-связанный список

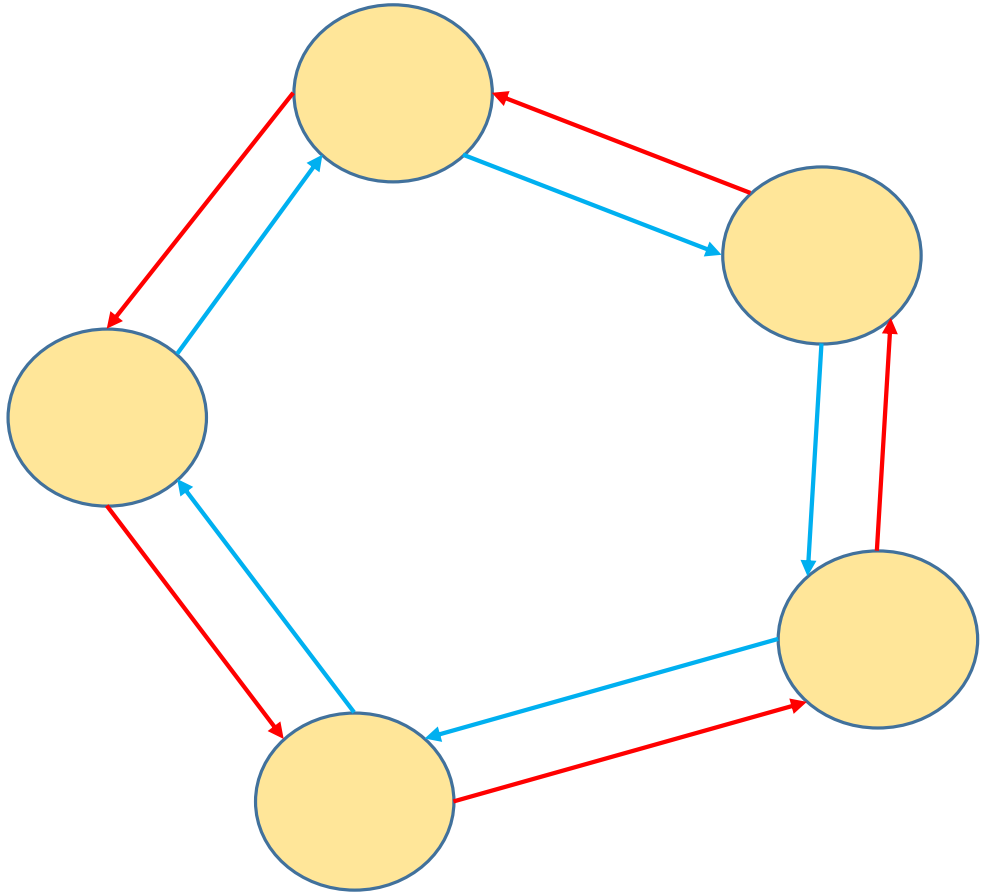


Плюсы:

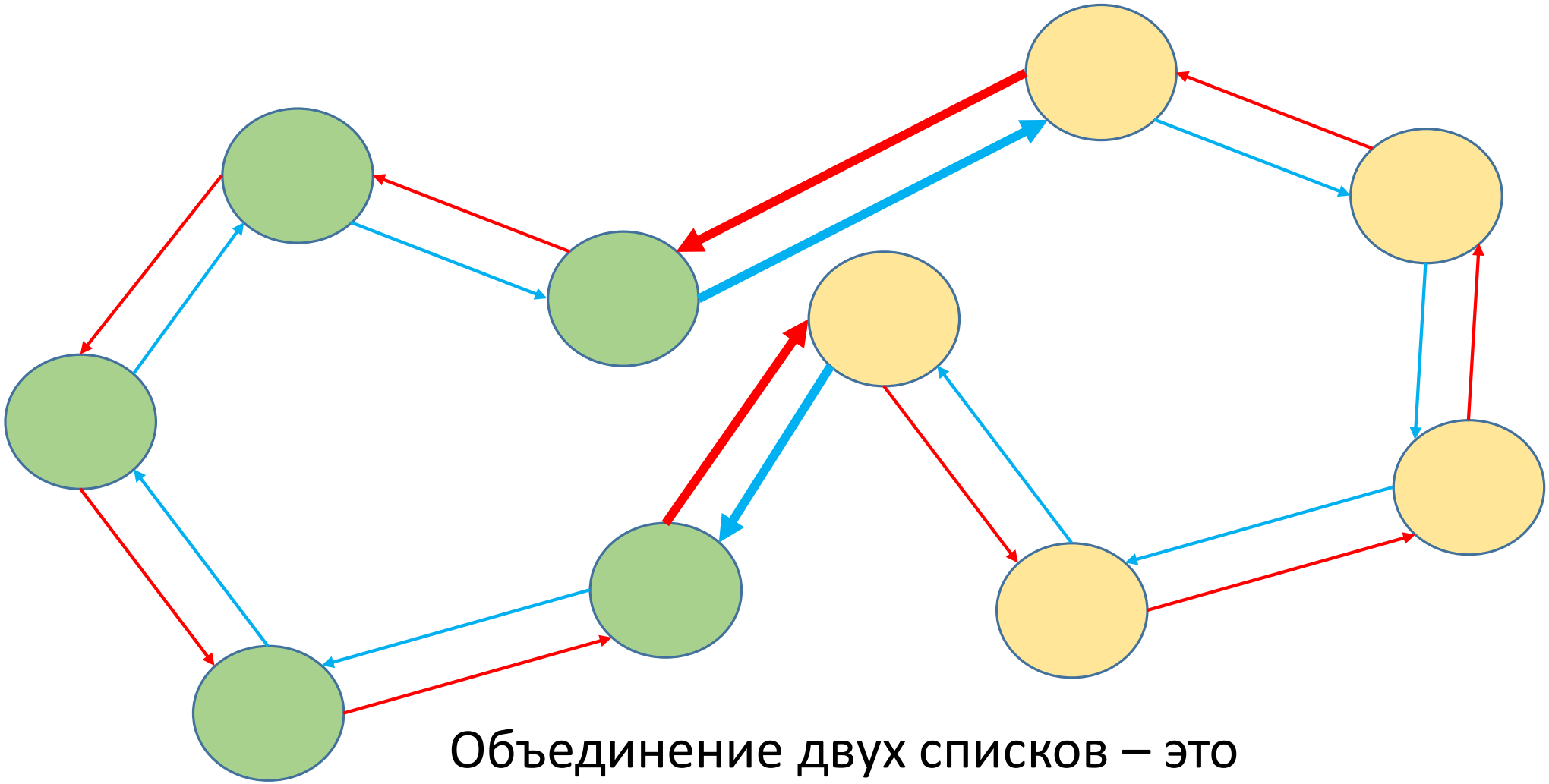
1. Любой элемент можно считать началом списка
2. Нет необходимости дополнительной структуры данных для хранения головы и хвоста
3. Обход всего списка с любого элемента прост и всегда одинаков по времени O(N)
4. Вставка элемента быстрая и простая

Циклический дважды-связанный список





Циклический дважды-связанный список



Объединение двух списков – это
поменять местами всего 4 указателя!

Циклический дважды-связанный список

- Сделать на умных указателях не получится!
- Действительно, так как все элементы списка равноправны, содержат shared_ptr на следующий/предыдущий, то shared_ptr никогда не обнулится.
- Даже если у нас уже в программе нет нигде никаких ссылок на элементы списка
- Следовательно такие структуры данных будут приводить к утечке памяти, если не удалять их специальным способом (но тогда уже нет речи про автоматическое управление памятью, а всё, что не автоматизировано надёжно – потенциальный источник ошибок:))

Полностью автоматическое управление памятью на основе сборки “мусора”

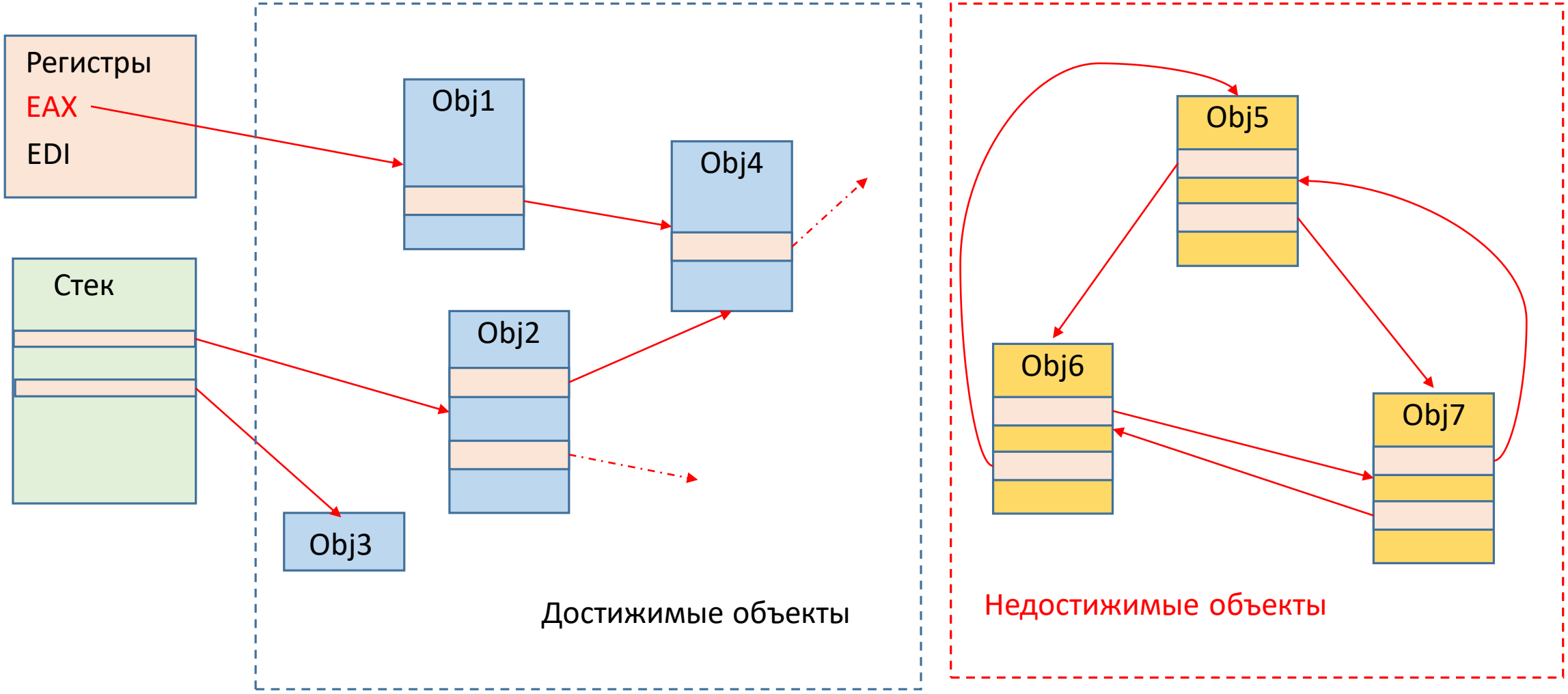
Что такое “сборка мусора”?

- Есть множество объектов, которые называются корневыми. Это те объекты к которым мы можем плучить доступ через указатели в регистрах, на стеке (стеках в случае многих потоков) и глобальные объекты.
- Те объекты которые могут быть найдены через цепочки указателей из корневого множества называются достижимыми. То есть, эвентуально, они могут быть ещё нужными и программа может их использовать.
- Соответственно все остальные объекты являются недостижимыми. То есть у программы уже нет никаких способов добраться до них – доступ к ним уже утерян.

Что такое “сборка мусора”?

- Во время работы программы происходит автоматическое определение недостижимых объектов (тех объектов программы, доступа к которым уже нет из корневого множества)
- После определения множества недостижимых объектов – эти объекты удаляются и освобождается занимаемая ими память.
- Алгоритмы нахождения недостижимых объектов ☺ называются алгоритмами сборки мусора.
- Соответственно, программный код, реализующий эти алгоритмы, называют сборщиком мусора.

Достижимые и недостижимые объекты



Алгоритм сборки мусора

- Определить множество недостижимых объектов
- Вызвать у них деструкторы/финализаторы
- Освободить память
- Как видим всё очень просто ☺

Немного истории

- Впервые подобная техника автоматического освобождения памяти была придумана и применена Джоном МакКарти в 1959 году для упрощения управления памятью в языке Lisp.
  (если не знакомы с этим языком, то могу рекомендовать ознакомиться, так как многие идеи современных языков зародились именно лиспе Джона МакКарти)
- В дальнейшем подобный подход был использован в языке пролог (1972 год создания первой версии языка)
- В языке SmallTalk (1970-1980 годы, Алан Кэй и др.)
- И других
- Эти языки серьёзно повлияли на все современные языки программирования, и дискуссии относительно автоматического управления памятью не утихают уже более полувека

Плюсы автоматического управления памятью

- Нет проблем с “висящими” указателями (когда указатель указывает на уже освобождённый блок)
- Нет проблем с разработкой структур данных на умных указателях (что в случае сложных структур требует довольно хорошего опыта и квалификации программиста)
- Нет проблем с утечкой памяти, как из-за ошибок освобождения (когда программист забыл освободить уже ненужный блок памяти), так и из-за сложных циклических структур данных и тд
- И многое другое (например, перемещающие/копирующие сборщики мусора могут давать небольшую оптимизацию кеширования данных процессором)

Минусы (куда же без них ☺)

- Снижение производительности программы. Иногда довольно ощутимое
- Некоторые типы сборщиков мусора могут вносить непредсказуемые паузы в выполнение программы
- Сложность контроля времени жизни объекта.
  Например, вызов финализатора объекта сетевого сокета может произойти очень поздно и всё это время сервер не сможет принимать соединения из-за переполненного пула открытых сокетов.
- Усложнение программы и её синхронизационного каркаса.
  Сборщик мусора может помочь с ликвидацией ошибок при работе с памятью, но добавить ошибок с многопоточной синхронизацией.
- И тд.

“Серебрянная пуля” отсутствует ☹

- Нет универсально хорошего сборщика мусора
- Под каждый тип задач нужно подбирать свой подходящий сборщик мусора и параметры его работы
- Тем не менее, во многих областях плюсы их использования настолько перевешивают все минусы, что сейчас многие популярные языки программирования рассчитаны на автоматическое управление памятью: Java, Python, C#, WASM, JavaScript, PHP, Groovy, Scala, Go, Ruby, Swift, Perl, Erlang, Haskell, Ocaml, Kotlin, Lua и др.
  Можно уверенно сказать, что среди современных популярных языков наибольшая аудитория программистов у языков со сборщиками мусора.

Основные типы сборщиков мусора

- Stop-the-world Mark-and-Sweep
  Самый простой и надёжный в реализации
- Copying, две области памяти, одна пустая, во второй объекты, при сборе мусора достижимые объекты копируются в пустую область, области меняются ролями
- Generational, причём для разных поколений могут быть разные стратегии сборки мусора
- Compacting, при сборке мусора, перемещает достижимые объекты к началу памяти, недостижимые – просто пропускает. После прохода по всем объектам, цикл повторяется.
- Concurrent, уменьшает паузу за счёт того, что часть работы выполняется во время работы основной программы в виде микропауз на барьерах и других операциях
- Parallel, работает в соседнем потоке не мешая основной программе. Одни из самых низких пауз в работе программы.
- И др.

Попробуем разработать простой вариант Stop-the-world mark-and-sweep GC

- Как узнать/задать множество корневых объектов?
- Как узнать где в данных объекта лежат возможные указатели на другие объекты?
- Как определить корректность алгоритма?
- Каким основным свойствам (инвариантам) должен удовлетворять алгоритм, чтобы быть корректным?
- Какой контекст сборщика мусора у нас будет?
- Какие основные примитивные операции над этим контекстом будут использоваться?
- Каким свойствам (инвариантам) должны удовлетворять эти примитивные операции, чтобы общие свойства алгоритма удовлетворялись?