***Управление памятью: сборка мусора, освобождение ресурсов, создание деструктора, указателей.***

***Создать программу для выполнения архивации и разархивации выбранного каталога пользователя в zip-архив.***

**Управление памятью включает в себя несколько механизмов:**

**Сборка мусора**. Это процесс автоматического управления памятью в куче, который заключается в поиске неиспользующихся участков памяти, которые ранее были заняты под нужды программы. Специальный программный модуль (сборщик мусора) периодически запускается, определяет, какие из созданных в динамической памяти объектов больше не используются, и освобождает занимаемую ими память.

**Освобождение ресурсов.** В процессе своего существования объект может использовать какие-либо ресурсы (например, открытые файлы, открытые соединения с базой данных и т. п.). Корректно завершить работу с ресурсами подобного рода — задача программиста. Для решения подобных задач используется механизм деструкторов. Деструктор — это метод объекта, который вызывается автоматически перед уничтожением объекта (перед освобождением памяти, выделенной для хранения объекта).

Ручное управление памятью. Язык не предоставляет механизмов для автоматического управления памятью. Выделение и освобождение памяти для создаваемых объектов остаётся на совести разработчика. Для создания объекта в динамической памяти программист явно вызывает команду выделения памяти. Эта команда возвращает указатель на выделенную область памяти, который сохраняется и используется для доступа к ней. Когда надобность в объекте проходит, программист явно вызывает команду освобождения памяти, передавая ей указатель на удаляемый объект.

Получение ресурса есть инициализация (RAII). Это программная идиома в ООП, смысл которой заключается в том, что выделяемая для объекта область памяти строго привязывается к его времени существования. Память выделяется в конструкторе и освобождается в деструкторе.

**Сборка мусора (Garbage Collection, GC)**

Автоматическое управление памятью: Сборка мусора – это форма автоматического управления памятью, при которой среда исполнения (например, виртуальная машина Java или .NET CLR) автоматически отслеживает объекты, созданные в памяти, и освобождает память, занимаемую объектами, которые больше не используются программой.

Отслеживание ссылок: GC работает, отслеживая ссылки на объекты. Объект считается “живым”, если на него есть хотя бы одна ссылка из других “живых” объектов или из корней (например, глобальные переменные, переменные в стеке). Если на объект нет ссылок, он считается “мусором” и его память может быть освобождена.

Алгоритмы сборки мусора: Существует множество алгоритмов сборки мусора, например:

Mark-and-Sweep: Помечает “живые” объекты, затем “прометает” и освобождает непомеченные.

Copying GC: Копирует “живые” объекты в другую область памяти, что позволяет заодно дефрагментировать память.

Generational GC: Делит память на “поколения”, предполагая, что более “старые” объекты живут дольше, и проводит сборку мусора чаще в “молодых” поколениях.

**Преимущества:**

Уменьшение ошибок: Устраняет множество ошибок, связанных с ручным управлением памятью, таких как утечки памяти и двойное освобождение.

Ускорение разработки: Упрощает разработку, освобождая программиста от необходимости явно освобождать память.

**Недостатки:**

Нагрузка на производительность: Процесс сборки мусора может занимать процессорное время и приводить к кратковременным задержкам (паузам).

Недетерминированность: Нельзя точно предсказать, когда GC запустится и сколько времени это займет.

Повышенное потребление памяти: GC часто требует больше памяти, чем при ручном управлении.

Области применения: GC широко используется в языках высокого уровня, таких как Java, C#, Python, JavaScript, Go, Ruby и другие.

**Освобождение ресурсов**

Широкое понятие: Освобождение ресурсов – это более общее понятие, чем просто освобождение памяти. Оно включает в себя закрытие файлов, сетевых соединений, освобождение других ресурсов операционной системы и т.п.

Ручное управление: В языках, где нет автоматической сборки мусора (например, C, C++), программист должен явно освобождать все захваченные ресурсы, включая память, выделенную через malloc или new, а также ресурсы, полученные от операционной системы.

Идиомы RAII (Resource Acquisition Is Initialization): В C++ часто используют RAII, где захват и освобождение ресурсов связаны с созданием и уничтожением объектов. Деструкторы играют ключевую роль в RAII.

Блоки try...finally/using: В языках с исключениями часто используется блок try...finally (или using в C#) для гарантированного освобождения ресурсов, даже если возникает исключение.

**Деструкторы**

Специальные методы: Деструктор – это специальный метод (функция) класса, который автоматически вызывается, когда объект класса перестает существовать (например, когда он выходит из области видимости или когда память, занимаемая им, должна быть освобождена).

Освобождение ресурсов: Деструкторы часто используют для освобождения ресурсов, принадлежащих объекту, включая память, открытые файлы и т.д.

Определение деструктора: Синтаксис определения деструктора зависит от языка: в C++ он обозначается как ~ClassName(), в Python – метод \_\_del\_\_().

Важность деструкторов в C++: В C++ деструкторы играют критическую роль в RAII и гарантируют, что ресурсы будут освобождены, даже если при выполнении кода возникают исключения.

Детерминированность (в C++): Вызов деструктора происходит детерминированно, когда объект выходит из области видимости или когда память, занимаемая им, освобождается явным вызовом delete.

При GC (например, в Java): В языках со сборкой мусора деструкторы (финализаторы) работают по-другому. Их вызов не гарантирован и может произойти значительно позже, чем объект станет мусором. Поэтому финализаторы не подходят для освобождения ресурсов, которые должны быть освобождены немедленно. В таких случаях используются другие механизмы, как, например, try-with-resources в Java.

**Указатели**

Прямой доступ к памяти: Указатели – это переменные, которые хранят адрес памяти. Они предоставляют прямой доступ к памяти и позволяют манипулировать ее содержимым.

Ручное управление памятью: В языках, таких как C и C++, указатели играют ключевую роль в ручном управлении памятью. Программист сам должен следить за выделением и освобождением памяти, используя malloc, new, free и delete.

Опасности: Использование указателей может быть опасным и привести к ошибкам, таким как:

Утечки памяти: Когда память выделена, но не освобождена.

Висячие указатели: Когда указатель указывает на уже освобожденную память.

Переполнение буфера: Когда запись данных происходит за пределы выделенной памяти.

Разделение сегментации: Когда происходит попытка доступа к памяти, к которой у процесса нет доступа.

Ограниченное использование в языках со сборкой мусора: В языках со сборкой мусора, таких как Java и C#, указатели обычно ограничены. В этих языках существуют понятия “ссылок” или “управляемых указателей”, но прямое управление адресами памяти обычно недоступно.

Безопасность типов: В некоторых языках (например, Rust) существует система типов, которая позволяет избежать ошибок с указателями во время компиляции.