Урок 6

Работа с динамической памятью

Представление в памяти коллекций. Управление памятью.

Введение

В предыдущих уроках мы изучали алгоритмы работы с различными данными. Теперь пришло время посмотреть, а как они действуют в памяти системы. Как размещаются и хранятся данные в памяти? Как управлять памятью для более эффективной работы программы?

Управление памятью с точки зрения разработчика компилятора

Под управлением операционной системы находятся все запускаемые приложения, в том числе и компиляторы. Поэтому при обращении к системным ресурсам компилятор вынужден полагаться на предоставляемые стандартные функции и примитивы. Управление памятью с точки зрения компилятора существенно ограничено возможностями целевой архитектуры и операционной системы.

Основные фазы работы с памятью

Один из самых важных ресурсов компьютера – это память. Сейчас именно на компилятор языка программирования возлагается обязанность обеспечивать доступ к физической памяти, распределять и освобождать ее.

Компилятор должен выполнять следующие задачи:

- при объявлении переменной выделять под нее память;
- инициализировать выделенную память некоторым начальным значением;
- предоставлять программисту возможность использовать выделенную память;
- как только память перестает использоваться, освобождать ее (возможно, предварительно очистив);
- обеспечивать возможность последующего повторного использования освобожденной памяти.

Хоть эта схема и кажется простой, правильно реализовать эти действия в компиляторе и добиться корректной работы использующих его программ достаточно сложно. Большинство ошибок, возникающих в современных программах, связано с некорректным использованием памяти.

Проблемы управления памятью

Перечислим основные проблемы работы с памятью.

1. **Память компьютера не бесконечна.**

Приходится постоянно учитывать, что свободная память может закончиться. Эта проблема – наиболее распространенная, но и решается проще остальных: можно купить дополнительную аппаратуру, и память появится. Но не стоит надеяться только на расширение памяти: все-таки оптимально использовать тот объем, что имеется на данный момент.

2. **Механизмы управления памятью (автоматические и ручные) в зависимости от языка программирования.**

В современных языках программирования уже встроены автоматические механизмы распределения и освобождения памяти: new/delete в C++ и JAVA, автоматические средства управления в Python. В этом случае вся ответственность переходит от компилятора к программисту, и человеческий фактор

не позволяет гарантировать, что память используется эффективно и оптимально. Ошибки, возникающие при некорректной работе с памятью, относительно непредсказуемы: они могут возникать в крайне редких случаях, могут зависеть от порядка исполнения предыдущих операторов программы. Поэтому выявлять и устранять их гораздо сложнее, чем обычные «алгоритмические» неисправности. Пример типичной ошибки – выделение ресурса лишь в одной из возможных ветвей условного оператора с дальнейшим безусловным использованием или освобождением этого ресурса в последующих частях программы.

3. **Работа с памятью, которая является общей для нескольких процессов.**

Не забываем: в многопоточных приложениях несколько процессов работают с общими ресурсами. Всегда держите в уме синхронизацию потоков, иначе возникнут проблемы с общей областью памяти.

Память становится свободной, как только выполняется оператор явного освобождения памяти (free/delete). Другой варианта – в момент окончания времени жизни последней переменной, использующей данную область памяти. Эти операции, делающие структуру данных логически недоступной, называются освобождением памяти. Но для компилятора в этот момент работа только начинается.

Освобожденную память необходимо утилизировать – вернуть системе как свободную. Сами операции уничтожения памяти и утилизации могут быть разнесены по времени.

Статическая и динамическая память

При создании компилятора необходимо различать два важных класса информации о программе:

- Статическая информация, известная во время компиляции;
- Динамическая информация – сведения, неизвестные во время компиляции, но становящиеся известными во время выполнения программы.

Это разделение позволяет определить, каким механизмом распределения памяти необходимо пользоваться для переменной, объекта или функции.

Если использовать простые переменные, то их проще определить на этапе компиляции – это статическая информация. Если задать переменную (массив) и запрашивать у пользователя его размер, то это уже динамическая информация.

Статический способ выделения памяти предпочтителен:заранее известно, сколько памяти выделено. Но не всегда это удобно и уместно. С массивом выделение большего объема памяти будет неэффективно: у пользователя должен быть выбор необходимого размера для него. Поэтому иногда распределение памяти под массивы откладывают на этап выполнения программы.

Это приемлемо для языков, разрешающих описание динамических массивов (массивов с границей, неизвестной во время компиляции). К ним относятся JAVA, Python. В этом случае механизм распределения памяти будет одинаковым для всех массивов. А в C/C++ память под массивы всегда можно выделять статически.

Влияние управления памятью на языки программирования

При выборе того или иного метода распределения памяти необходимо учитывать специфику исходного языка программирования. Практически все языки ориентированы на конкретный механизм управления памяти.

Языки Java, C# и Python не предоставляют программисту никакого механизма для явного выделения или освобождения памяти. Вся ответственность за управление памятью лежит на механизме сборки мусора, а значит – на разработчиках компилятора.

Неявное управление памятью

В современных языках программирования наблюдается тенденция к усложнению средств управления памятью, требуемых при реализации транслятора. Непосредственно это касается реализации сборки мусора. При этом программисту полностью или частично отказывается в явных средствах управления памятью. Так разработчики компиляторов пытаются тем уйти от человеческого фактора и ошибок использования памяти.

Недостатки такого подхода:

- замедление программ, использующих сборку мусора. Это может отрицательно сказаться на приложениях, работающих в реальном времени.

Преимущества:

- неявное управление памятью освобождает программиста от большого объема рутинной работы по отслеживанию занимаемой и возвращаемой памяти. Все проблемы этого процесса «скрываются» от него компилятором.

Поэтому неявное управление стало наиболее популярным методом. Но теория и практика языков программирования активно развиваются: то, что забыто сегодня, может стать актуальным завтра.

С появлением и распространением языка C во главу угла стали ставить эффективность и предоставление программисту широких возможностей влияния на системные процессы. Системы со сборкой мусора в последние 20-25 лет преимущественно оставались уделом академических исследователей. Только удешевление аппаратуры и желание избавиться от ошибок помогли неявному управлению памятью возвратить себе позиции в практических языках программирования.

Фазы управления памятью

Действия любого метода управления памятью можно разделить на стандартные фазы. Обозначим три крупных этапа: начальное выделение памяти, освобождение и уплотнение, повторное использование.

На первом этапе необходимо разметить всю память как свободную или используемую. Также нужно учитывать свободную память.

На втором этапе система должна определить, какие из уже использованных участков памяти стали ненужными, и освободить эту память.

На третьем этапе память должна быть повторно использована. Для этого может потребоваться механизм уплотнения свободной памяти, который создаст нескольких больших блоков свободной памяти из множества маленьких.

Выделяют три основных метода управления памятью:

- Статическое распределение памяти;
- Стековое распределение памяти;
- Представление памяти в виде «кучи» (heap).

Большинство языков программирования требуют от компиляторов использования всех трех механизмов. Обычно подразумевается применение самого «дешевого» из пригодных способов.

Статическое управление памятью

Простейший и наиболее распространенный способ управления памятью – статический.

Он позволяет достичь максимальной эффективности, экономя время на выделение и освобождение памяти во время выполнения программы.

Желание свести все к статическому управлению памятью заставляет разработчиков отказаться от многих привычных конструкций. Это функции рекурсивного перебора (так как неизвестно, сколько раз будет вызвана функция), массивы с неконстантыми или изменяющимися границами, вложенные процедуры и подпрограммы.

В распоряжении программиста оказываются только простые переменные, структуры и массивы фиксированного размера.

Рассмотрим выделение статической памяти на примере массива.

Представление в памяти массива

Сначала разберемся с отдельными переменными.

Целые числа могут представляться со знаком или без знака. Обычно в памяти они занимают один или два байта. Соответственно, принимают в однобайтном формате от 00000000 до 11111111, а в двухбайтовом – от 00000000 00000000 до 11111111 11111111.

Например, число 78 будет выглядеть в памяти при однобайтном размере следующим образом: 01001110.

Номер разряда	7	6	5	4	3	2	1	0
бит числа	0	1	0	0	1	1	1	0

А в двухбайтовом представлении так:

Номер разряда	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
бит числа	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0

Целые числа со знаком обычно занимают в памяти от одного до четырех байт. При этом информацию о знаке числа содержит самый левый (старший) разряд. Знак «+» кодируется нулем, а «-» – единицей.

Рассмотрим особенности записи целых чисел со знаком на примере однобайтового формата, при котором для знака отводится один разряд, а для цифры абсолютной величины – семь разрядов.

В памяти применяются три формы записи (кодирования) целых чисел со знаком: прямой, обратный и дополнительный код. Последние две формы применяются особенно широко, так как позволяют

упростить конструкцию арифметико-логического устройства компьютера: происходит замена разнообразных арифметических операций сложением.

Примеры записи:

Положительные числа в прямом, обратном и дополнительном кодах изображаются одинаково: двоичными кодами с цифрой 0 в знаковом значении.

Число 89 в памяти при условии знака будет выглядеть в памяти так:

0	1	0	1	1	0	0	1

Для целых чисел все просто и понятно. Разберемся с вещественными числами.

Для них есть правило: ***любое число M в системе счисления с основанием y можно записать в виде M = L * y^i, где L называется мантиссой числа, а i - порядком.*** Такой способ записи чисел называется представлением числа с плавающей точкой (вещественных чисел).

Если плавающая точка расположена в мантиссе перед первой значащей цифрой, то обеспечивается запись максимального количества значащих цифр числа – при фиксированном количестве разрядов, отведенных под мантиссу. Получаем максимальную точность представления числа в машине. Мантисса должна быть правильной дробью, первая цифра которой отлична от нуля.

Это представление вещественных чисел называют нормализованным.

Разберем пример представления вещественного числа 6.25 = 110.01 = 0.11001 *2^11/

0	0	0	0	0	0	0	1	1	1	1	0	0	1	0	0	0	...	0	0	0
знак числа	знак порядка	порядок							мантисса											

Массив – это упорядоченная, проиндексированная последовательность однотипных данных. Доступ к его элементам производится по индексу. Наиболее распространенный способ индексации – с нуля.

Например, чтобы обратится к третьему элементу массива Array, необходимо написать Array[2].

Каждый элемент массива занимает строго определенное количество байт – в зависимости от типа данных. Если массив типа int, то каждый его элемент будет занимать в памяти 2 байта.

Перейдем к представлению в памяти совокупностей чисел.

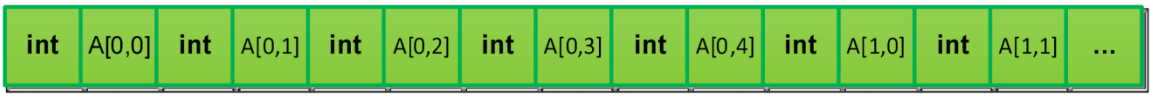
Любой массив (одномерный, двумерный, n-мерный) располагается в памяти последовательно.

Рассмотрим матрицу.

```
from random import random

M = 10
N = 5
a = []
for i in range(N):
    b = []
    for j in range(M):
        b.append(int(random() * 11))
        print("%3d" % b[j], end='')
        a.append(b)
    print('   ')
```

Под каждый элемент **a[i,j]** типа **integer** выделяется две ячейки памяти. Элементы размещаются в порядке изменения индекса. Это соответствует схеме вложенных циклов: сначала размещается первая строка, затем вторая, третья и так далее. Внутри строки по порядку идут элементы: первый, второй...



Доступ к любой переменной возможен, только если известен адрес ячейки памяти, в которой она хранится. Конкретная память выделяется для переменной при загрузке программы: устанавливается взаимное соответствие между переменной и адресом ячейки. Но если мы объявили переменную как массив, то программа «знает» адрес его первого элемента. Как же происходит доступ к остальным?

При реальном доступе к ячейке памяти, в которой хранится элемент двумерного массива, система вычисляет ее адрес по формуле:

Adress+SizeElem*Cols*(I-1)+SizeElem*(J-1), где

- **Adress** – фактический начальный адрес, по которому массив располагается в памяти;
- **I, J** – индексы элемента в двумерном массиве;
- **SizeElem** – размер элемента массива (например, два байта для элементов типа **integer**);
- **Cols** – количество элементов в строке.

Также есть ***понятие смещенного относительного адреса***. Он вычисляется по формуле **SizeElem*Cols*(I-1)+SizeElem*(J-1**).

Для работы программы память выделяется сегментами по 64 Кбайт. Как минимум один из них определяется как сегмент данных. В нем располагаются те данные, которые будет обрабатывать программа. Ни одна переменная программы не может располагаться более чем в одном сегменте. Даже если в сегменте находится только одна переменная, описанная как массив, она не сможет получить более чем 65536 байт. Но наверняка кроме массива в сегменте данных будут описаны и другие переменные. Поэтому реальный объем памяти, который может быть выделен под массив, находится по формуле **65536-S**, где S – объем памяти, уже выделенный под другие переменные.

Стековое управление памятью

Чтобы уйти от явных ограничений работы с памятью при статическом управлении, можно воспользоваться стековым. Его идея в том, что при входе в блок или процедуру на вершине специального стека выделяется память, необходимая для размещения переменных, объявленных внутри этого блока. При выходе же из блока память снимается не «вразнобой», а всегда только с вершины стека. Задачи утилизации и повторного использования упрощаются, а проблема уплотнения снимается.

При таком подходе все значения каждого блока или процедуры объединяются в одну часть стека – рамку. Для управления памятью потребуется указатель стека, показывающий на первый свободный элемент, и указатель рамки, хранящий адрес ее дна (это нужно при выходе из блока или процедуры).

Стековое управление памятью особенно выгодно для языков со строгой вложенной структурой входов в процедуры и выходов из них. Дополнительно необходимо потребовать, чтобы структуры данных имели фиксированный размер, могли создаваться программистом только при входе в процедуру и обязательно уничтожались при выходе. Тогда всю информацию, необходимую для работы процедуры или подпрограммы, можно собрать активационную запись. Она полностью определяет экземпляр процедуры. В этом случае стекового управления памятью достаточно для всех элементов данных, используемых в программе.

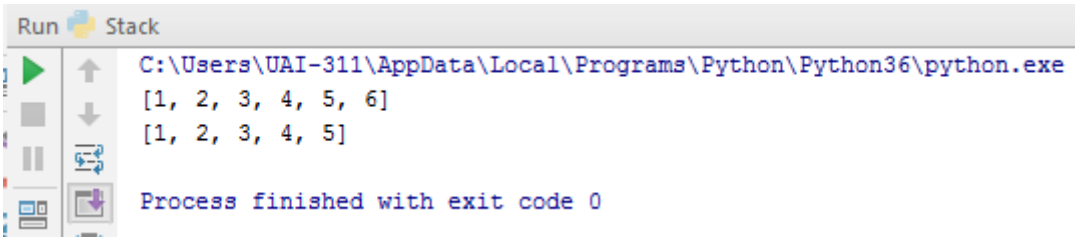
По сравнению со статическим распределением памяти, удалось снять ограничения на вложенные процедуры и рекурсивные вызовы. Но от всех переменных по-прежнему требуются фиксированные размеры. Зато при выходе из процедуры компилятор точно знает, каков размер освободившегося блока и сколько ячеек памяти надо снять с вершины стека.

Рассмотрим алгоритм работы простой структуры стека.

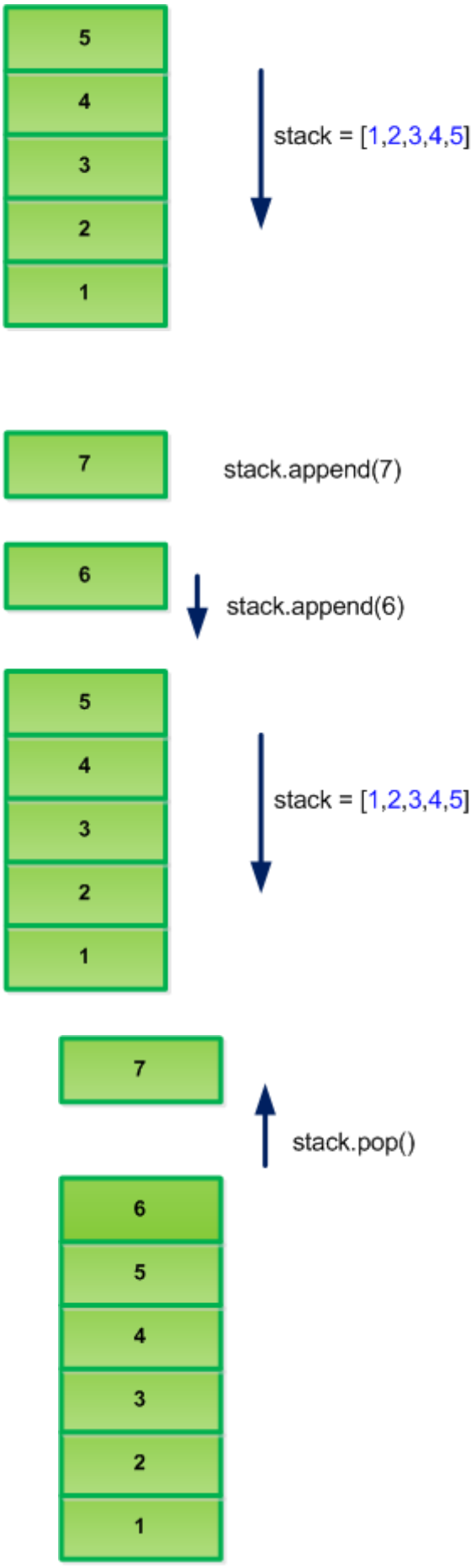
Стек – это организация структуры данных, при которой элементы добавляются в конец списка (массива) и удаляются тоже с конца. Принцип – «последний пришел, первый ушел».

Коллекцию-список в Python можно использовать как стек, когда последний добавленный элемент извлекается первым. Для извлечения элемента с вершины стека есть метод **pop()**:

```
stack = [1, 2, 3, 4, 5]
stack.append(6)
stack.append(7)
stack.pop()
print(stack)
stack.pop()
print(stack)
```



Проиллюстрируем стековый механизм управления памятью:


5
4
3
2
1
stack = [1,2,3,4,5]
7
stack.append(7)
6
stack.append(6)
5
4
3
2
1
stack = [1,2,3,4,5]
7
stack.pop()
6
5
4
3
2
1

Даже если в программе есть динамическая информация, то все равно сначала распределяется вся статическая, а динамическая размещается над ней. В момент компиляции мы еще не знаем адреса рамок, но можем распределять статические адреса относительно начала определенной рамки.

Управление кучей

Управление кучей – механизм, предназначенный для работы со всеми структурами данных, которые непригодны для статического или стекового распределения памяти. Оно позволяет выделять и освобождать память в любой момент работы программы.

При управлении кучей важно иметь в виду, что:

1. Моменты освобождения памяти не всегда очевидны;
2. Порядок освобождения памяти непредсказуем;
3. Память для новых объектов уже к моменту их распределения напоминает решето.

Решение этих проблем возлагается на механизм сборки мусора. По сути, это процесс дефрагментации памяти: поиск неиспользуемой и перераспределение текущей.

Критичная проблема в этом виде управления – отслеживание активных элементов в памяти. Как только необходимо освободить дополнительную память, процесс сборки мусора должен утилизировать все неиспользуемые фрагменты.

Определить, используется ли данный момент в программе, можно с помощью счетчиков ссылок и алгоритмов разметки памяти.

Отслеживание свободной памяти с помощью подсчета ссылок

Этот способ заключается в приписывании каждому объекту в памяти специального счетчика ссылок. Он показывает количество активных переменных, использующих данный объект.

1. Сначала счетчику присваивается значение, равное 1;
2. При создании новых указателей на данный фрагмент памяти значение счетчика увеличивается на единицу;
3. Если переменная, указывающая на данный фрагмент памяти, перестает существовать, то значение счетчика уменьшается на единицу;

4. При достижении счетчиком 0 фрагмент памяти считается более не используемым и может быть утилизирован.

Недостатки способа:

1. Алгоритм может не справиться с определением свободной памяти. Для циклического списка уничтожение внешнего указателя на эту структуру делает ее мусором, и в то же время его счетчики ссылок не становятся равными нулю, так как элементы списка указывают друг на друга;

2. Использование механизма счетчиков ссылок приводит к потере эффективности во время исполнения. При каждом присваивании в программе необходимо производить соответствующие арифметические операции.

Из-за этих проблем механизм счетчиков ссылок не получил широкого распространения.

Отслеживание свободной памяти с помощью разметки

При этом подходе все действия по поиску неиспользуемых переменных откладываются до возникновения недостатка памяти – то есть до того момента, когда программа требует выделить фрагмент слишком большого размера. Тогда стартует процесс сборки мусора, начинающийся именно с разметки памяти.

В отличие от счетчиков ссылок, механизм разметки памяти не приводит к замедлению программ, не использующих сборку мусора. Но если потребность в сборке мусора все-таки возникает, то все процессы будут заморожены на некоторое время. Это может заметить и пользователь.

Управление памятью – важнейшая особенность языка Python

Автоматическое управление памятью

В итоге управление памятью сводится к тому, чтобы:

- Для каждой совокупности данных в программе выделить место в оперативной памяти;
- Следить, чтобы данные не вышли за пределы памяти;
- И не забыть освободить эту память, когда данные более не нужны.

В Python об этом думать не надо: язык программирования все сделает за вас. Но в том, как именно Python работает с памятью, надо разбираться.

Рассмотрим простой пример: **сколько памяти занимает 1 миллион целых чисел?**

В Python можно получить необходимую информацию прямо из интерактивной консоли, не обращаясь к исходному коду на C.

Определим изнутри типы данных и узнаем, на что именно расходуется память.

Форматы данных, которые нам потребуются:

Символ	Значение C	Значение Python	Длина на 32-х битной машине

c	char	Строка из одного символа	1
i	int	int	4
l	long	int	4
L	unsigned long	int	4
d	double	float	8

В результате получим следующие данные:

Тип	**Имя в Python**	**Формат**	**Формат для вложенных объектов**	**Длина на 32 bit**	**Длина на 64 bit**	**Память для GC***
Int	PyIntObject	LLl		12	24	
float	PyFloatObject	LLd		16	24	
str	PyStringObject	LLLli+c*(длина+1)		21+длина	37+длина	
unicode	PyUnicodeObject	LLLLlL	L*(длина+1)	28+4*длина	52+4*длина	
tuple	PyTupleObject	LLL+L*длина		12+4*длина	24+8*длина	Есть
list	PyListObject	L*5	L*длину	20+4*длина	40+8*длина	Есть
Set/frozenset	PySetObject	L*7+(lL)*8+lL	LL*длина	(<=5 элементов) 100 (>5 элементов) 100+8*длина	(<=5 элементов) 200 (>5 элементов) 200+16*длина	Есть
dict	PyDictObject	L*7+(lLL)*8	lLL*длина	(<=5 элементов) 124 (>5 элементов) 124+12*длина	(<=5 элементов) 248 (>5 элементов) 248+2	Есть

					4*дли на	

Простые типы данных в Python в два-три раза больше своих прототипов на C. Разница обусловлена тем, что необходимо хранить количество ссылок на объект и указатель на его тип.

Частично это компенсируется внутренним кэшированием, которое позволяет повторно использовать ранее созданные объекты. Это возможно только для неизменяемых типов.

Для строк и кортежей разница не такая значительная – добавляется некоторая постоянная величина.

А списки, словари и множества, как правило, занимают на ⅓ больше, чем необходимо. Это связано с алгоритмом добавления новых элементов, который приносит в жертву память ради экономии времени процессора.

Итак, чтобы сохранить 1 миллион целых чисел, нам потребуется 11.4 мегабайт (12*10^6 байт) на сами числа и дополнительно 3.8 мегабайт (12 + 4 + 4*10^6 байт) на кортеж, который будет хранить на них ссылки.

Получается, что в среднем на каждое целое число уходит по 16 байт.

Управление динамической памятью

Python — язык с динамической типизацией и встроенным менеджером управления памятью. Рассмотрим работу этого менеджера:

1. Создается объект, под него выделяется память;
2. Когда объекта становится ненужным, память освобождается.

Но иногда система дает сбой: память постоянно растет, встроенный garbage collector не работает. В таких случаях программист обычно откатывает версию программы до момента ошибки памяти. Неплохой вариант решения, но он не дает понимания сути проблемы и не страхует от нее в дальнейшем.

Большинство проблем работы с памятью возникают при удалении объектов, то есть при освобождении памяти.

Программист думает, что когда объект становится ненужным – он априори удален. А Python считает иначе. В результате – конфликт.

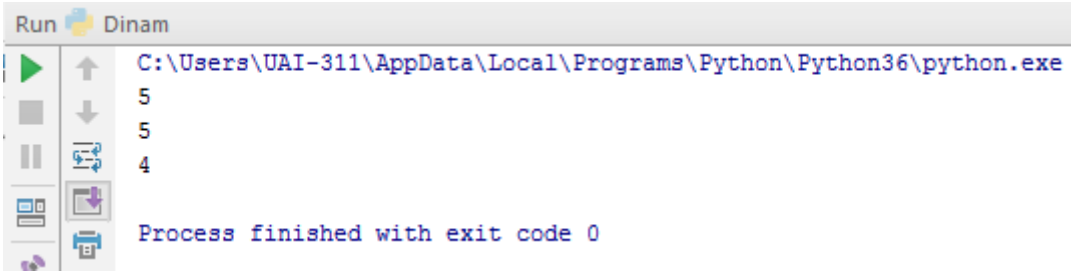
В Python существуют два механизма для освобождения памяти: **decref** и **garbage collector**.

Первый работает по принципу отслеживания свободной памяти с помощью подсчета ссылок на объект.

Наглядный пример:

```
import sys
a = "Hello world"
b = a
print(sys.getrefcount(a))
print(sys.getrefcount(b))
```

```
del(a)
print(sys.getrefcount(b))
```

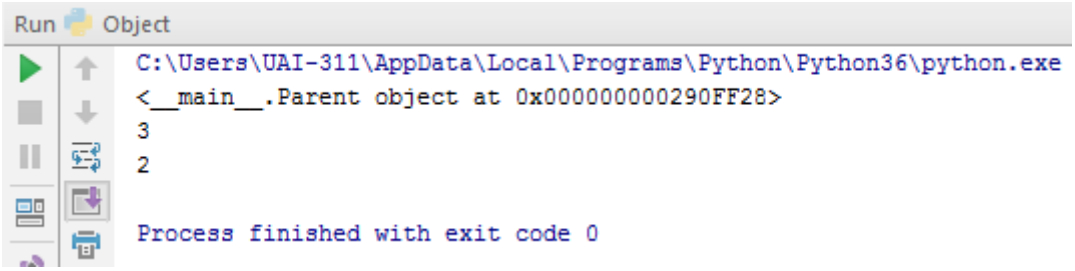


Рассмотрим действие этого механизма на примере организации работы дерева:

```
import sys
class Parent(object):
     def __init__(self):
          self.children = []
     def add(self, ch):
          self.children.append(ch)
          ch.parent = self

class Child(object):
     def __init__(self):
          self.parent = None

p = Parent()
p.add(Child())
print(p)
p.children
print(sys.getrefcount(p))
print(sys.getrefcount(p.children[0]))
```



Parent имеет ссылку на **Child**, а тот, в свою очередь, на родителя. Даже если программист удалит все ссылки, счетчик не изменит значение и останется равным 1. Объекты все равно остались в памяти, хотя уже не нужны. Появляется мусор.

Для решения этой проблемы можно воспользоваться вторым способом освобождения памяти, с помощью разметки – garbage collector.

Алгоритм сбора мусора по данному механизму:

- Определяем три поколения объектов;

- Когда новый объект создается, он попадает в первое поколение;
- Фиксируется количество созданных и удаленных объектов;
- Если разница больше порога, то запускается умный **cycle finder;**
- Если объект все еще не удален даже **gc**, то он перемещается в более старое поколение;
- Если проблема все еще не решена, то ловим нарушителя в gc.garbage.

Gc предоставляет много интересной информации: кто ссылается на объект, на кого ссылается он, кто попадает в garbage, список всех объектов, живущих в Python, и другое.

Cycle finder пытается найти **cycle dependencies** – циклические зависимости – и удалить их. Например, объекты A и B ссылаются друг на друга, но при этом «снаружи» на них никто больше не ссылается. Образуется циклическая зависимость. Ее определяет garbage collector и разыменовывают их.

Обычно это позволяет ничего не делать в случае parent-child. Проблемы возникают, когда один из объектов циклической зависимости имеет метод **__del__**.

Но **Garbage collector** вычисляет циклическую зависимость, в которой есть несколько объектов с **__del__**. И может возникнуть проблема ссылки метода **__del__** на уже несуществующий, удаленный элемент.

Подведем итоги урока:

1. Наиболее простой способ выделения памяти, позволяющий повысить производительность работы программы – это статический метод. Но не всегда такой способ возможно применить в реальных задачах.
2. В современном программировании не нужно задумываться об управлении памятью. Это дело интерпретатора/ компилятора. Но важно понимать, как работает механизм управления памятью. Это позволяет определить ошибки работы программы, когда память слишком быстро исчезает (заполняется), и места для дальнейшей работы не остается.

Практическое задание

1. Подсчитать, сколько было выделено памяти под переменные в ранее разработанных программах в рамках первых трех уроков. Проанализировать результат и определить программы с наиболее эффективным использованием памяти.

Примечание: Для анализа возьмите любые 1-3 ваших программы или несколько вариантов кода для одной и той же задачи. Результаты анализа вставьте в виде комментариев к коду. Также укажите в комментариях версию Python и разрядность вашей ОС.

Используемая литература

1. https://www.python.org
2. Марк Лутц. Изучаем Python, 4-е издание.