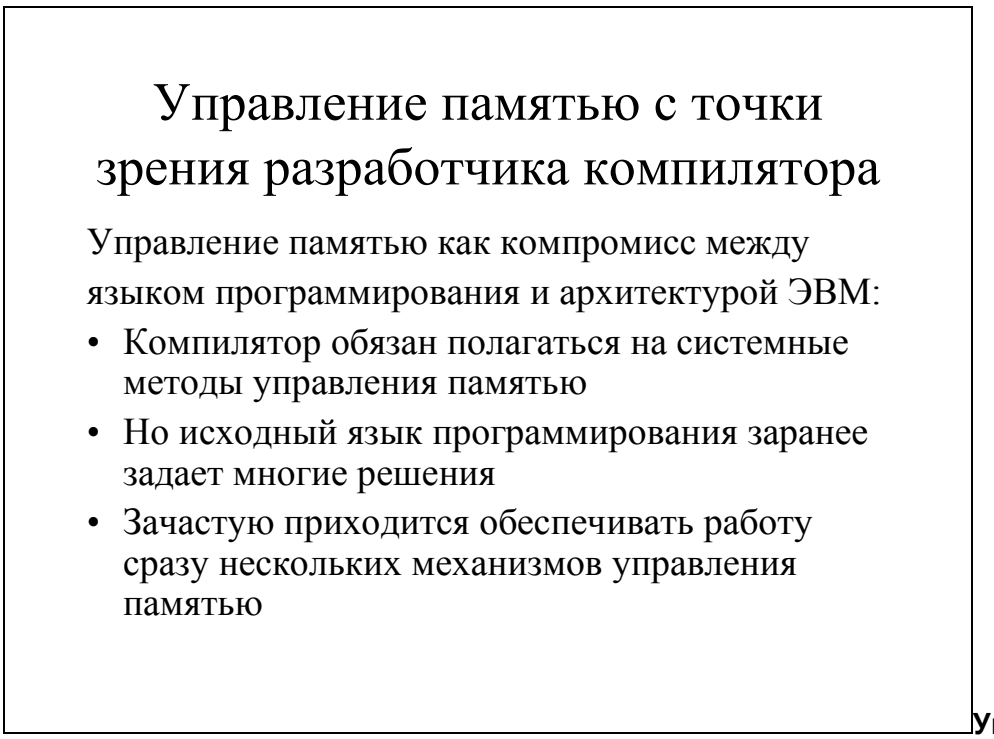
10. Управление памятью и сборка мусора

- Управление памятью с точки зрения разработчика компилятора
- Проблемы управления памятью
- Статическое и динамическое размещение памяти
- Стековый механизм управления памятью
- Управление кучей
- Подсчет ссылок и разметка памяти
- Управление памятью в .NET

Лекция 10. Управление памятью и сборка мусора

В этой лекции обсуждаются следующие вопросы:

- Управление памятью с точки зрения разработчика компилятора
- Проблемы управления памятью
- Статическое и динамическое размещение памяти
- Стековый механизм управления памятью
- Управление кучей
- Подсчет ссылок и разметка памяти
- Управление памятью в .NET

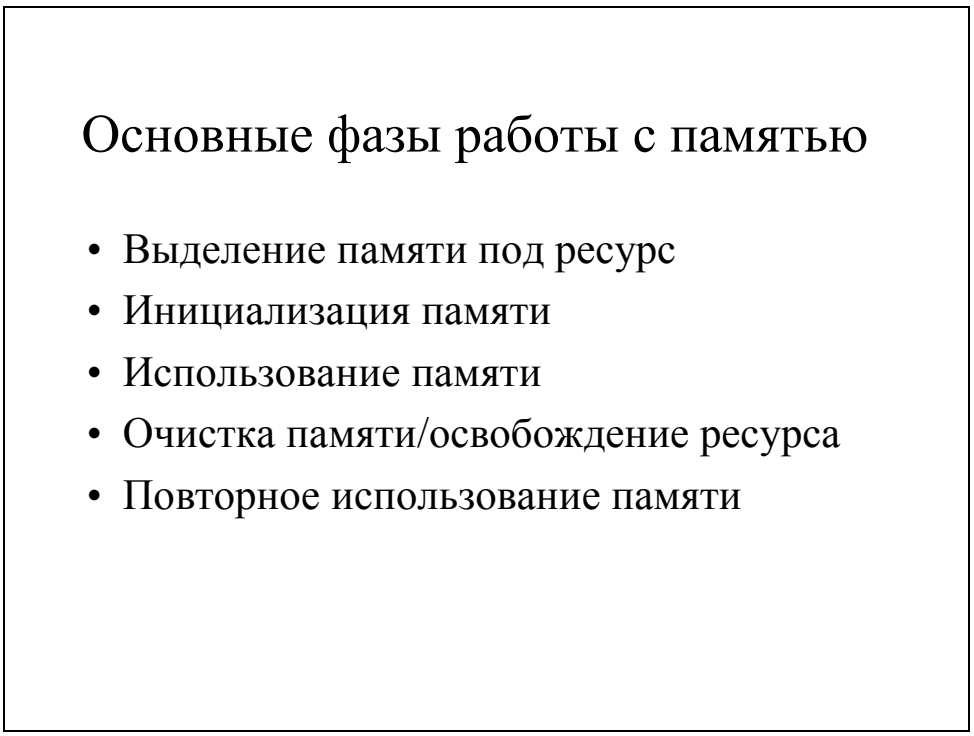


Управление памятью с точки зрения разработчика компилятора

Любой компилятор является всего лишь одним из многочисленных приложений, работающих под управлением данной операционной системы, и потому при обращении к системным ресурсам компилятор вынужден полагаться на предоставляемые стандартные функции и примитивы. Таким образом, управление памятью с точки зрения компилятора существенно ограничено возможностями целевой архитектуры и операционной системы. С другой стороны, большое количество решений по управлению памятью делается уже на этапе создания языка программирования (подробнее об этом ниже).

Итак, управление памятью при разработке компилятора является вопросом одновременно и машинно-зависимым, и языково-зависимым. В связи с этим разработчик компилятора должен найти наиболее эффективное отображение средств управления памятью, предлагаемых языком программирования, на заданную аппаратуру и ОС.

При этом зачастую возникает ситуация, когда приходится мириться с существованием сразу нескольких параллельных механизмов управления памятью. Даже в тех языках, в которых программист имеет возможность явного управления памятью, это никак не отменяет стандартных системных механизмов, так как разработчик компилятора обязан обеспечить корректную работу многих элементов программы, скрытых от конечного программиста (например, компилятор должен выделять память под саму оттранслированную программу, системные программы времени выполнения, точки входа и возврата из подпрограмм, временную память для вычисления выражений и т.п.).



Основные фазы работы с памятью

Память — это один из самых важных ресурсов компьютера. Так как современные языки программирования не обязывают программиста работать напрямую с физическими ячейками памяти, на компилятор языка программирования возлагается ответственность за обеспечение доступа к физической памяти, ее распределение и утилизацию. В качестве ресурса могут выступать самые разные логические и физические единицы: обычные переменные примитивного типа, массивы, структуры, объекты, файлы и т.д. Со всеми этими объектами необходимо работать и, следовательно, обеспечить выделение памяти под связанные с ними переменные в программах.

Для этого компилятор должен последовательно выполнить следующие задачи:

- выделить память под переменную;
- инициализировать выделенную память некоторым начальным значением;
- предоставить программисту возможность использования этой памяти;
- как только память перестает использоваться, необходимо ее освободить (возможно, предварительно очистив)
- наконец, необходимо обеспечить возможность последующего повторного использования освобожденной памяти.

С точки зрения программиста, описанная выше схема кажется очень простой. Тем не менее, аккуратно реализовать эти действия в компиляторе и добиться корректной работы программ, использующих этот механизм, достаточно сложно из-за различных проблем. Об этом свидетельствует и тот факт, что большинство ошибок, возникающих в современных программах, связано с некорректным использованием памяти. На следующих слайдах мы рассмотрим наибоеле распространенные проблемы, связанные с управлением памятью.

Проблемы управления памятью

- Память не бесконечна!
- Ошибки явного управления памятью
- Ошибки при работе с памятью возникают редко и потому труднонаходимы
- Проблема освобождения ресурса — система не знает, как освободить сетевой ресурс или снять блокировку в базе данных

Проблемы управления памятью

Самая большая неприятность управления памятью заключается в том, что память не бесконечна и потому приходится постоянно учитывать возможность исчерпания свободной памяти. Конечно, эта проблема постепенно теряет свою остроту в связи с постоянным удешевлением аппаратуры компьютеров, но учитывать эту опасность придется всегда.

Другая проблема возникает в тех случаях, когда язык программирования предоставляет программисту явный механизм управления памятью (такой, как malloc/free в языке C или new/delete в C++). В этих случаях компилятор не может гарантировать правильность работы обрабатываемых им программ и эта ответственность возлагается на программиста. К сожалению, люди значительно менее надежны, имеют тенденцию ошибаться и даже повторять свои ошибки, а во многих случаях и попросту игнорируют предоставленные им механизмы. Поэтому при таком подходе обычно возникает множество ошибок, что, в свою очередь, ведет к необходимости кропотливой отладки программ и существенно затрудняет работу программиста.

Особая неприятность ошибок, возникающих при некорректной работе с памятью, заключается в том, что эти ошибки относительно непредсказуемы, могут возникать в крайне редких случаях, могут зависить от порядка исполнения предыдущих операторов программы и потому существенно труднее в обнаружении и исправлении, чем обычные "алгоритмические" ошибки. Например, типичной ошибкой является выделение ресурса лишь в одной из возможных ветвей условного оператора с последующим безусловным использованием или освобождением этого ресурса в последующих частях программы.

Однако в некоторых случаях трудно обойтись без участия программиста. Например, освобождение ресурсов, ассоциированных с какими-либо внешними сущностями (файлами на диске, записями баз данных, сетевыми соединениями и т.п.), обычно требует явных операций по закрытию. В таких случаях простое освобождение памяти, занимаемой переменной в программе, решит только часть проблемы, так как после этого файл или запись в базе данных останутся недоступными для других приложений.

Проблемы управления памятью (окончание)

- Необходимо различать уничтожение памяти (уничтожение объекта/уничтожение путей доступа) и утилизацию памяти (сборка мусора)
- Проблема отслеживания различных путей доступа к структуре (различные указатели на одно и то же место, передача параметром в процедуру и т.д.), поэтому утилизация памяти обычно проблематична

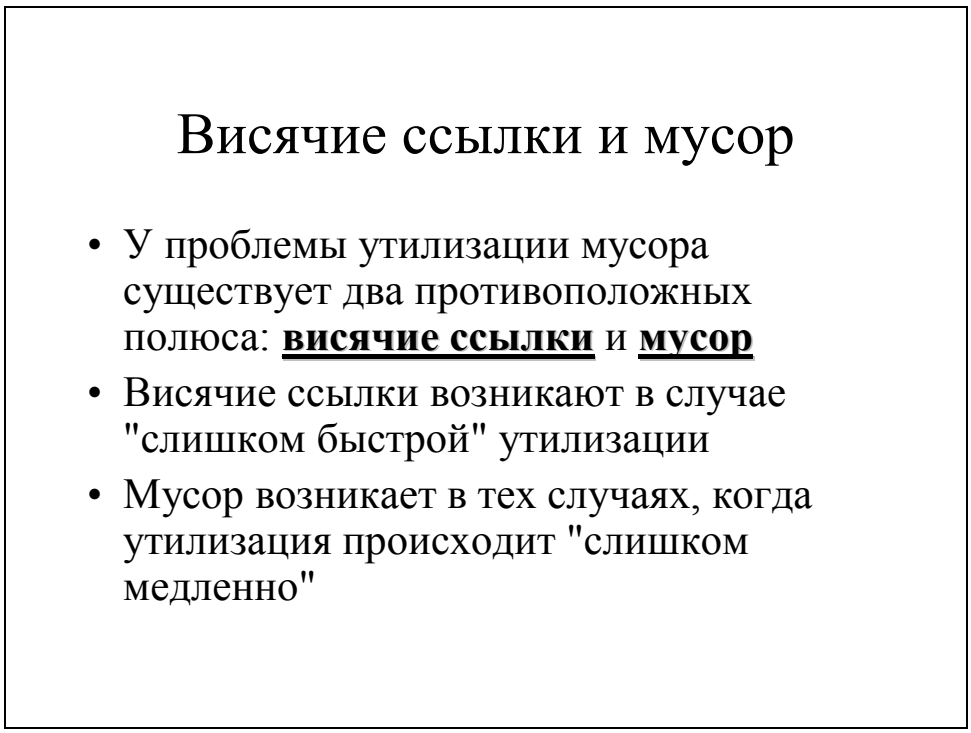
Проблемы управления памятью (окончание)

С точки зрения программиста, память становится свободной как только выполняется оператор явного освобождения памяти (free/delete) или в момент окончания времени жизни последней переменной, использующей данную область памяти. Эти операции, делающие структуру данных логически недоступной, называются *уничтожением памяти*. Однако с точки зрения разработчика компилятора в этот момент вся работа только начинается.

В случае с явным освобождением памяти все более или менее очевидно, хотя, как мы видели выше, и связано с проблемами для программиста. Но все равно большинство переменных освобождается автоматически (в конце блока, процедуры и т.д.). Поэтому момент окончания использования памяти еще необходимо отследить, т.е. понять, что данный фрагмент памяти действительно никто больше не использует. Это не всегда тривиально, так как в программе может существовать несколько элементов, связанных с данной областью памяти. В таких случаях говорят о существовании различных *путей доступа* к структуре. Наиболее простой пример — это два указателя, указывающих на один и тот же адрес. Другой пример — передача массива параметром в процедуру. В общем случае отслеживание всех путей доступа к структуре трудно реализуемо и дорогостояще.

Затем освобожденную память необходимо вернуть системе как свободную — утилизировать. Отметим, что операции уничтожения памяти и утилизации могут быть сильно разнесены по времени. Более того, в большинстве языков у программиста нет возможности форсировать утилизацию данного конкретного объекта (хотя в C# такая операция предусмотрена для крупных объектов).

Понятно, что утилизация памяти сильно затруднена из-за проблем с определением единственности доступа к уничтожаемой области памяти. На следующем слайде мы рассмотрим проблемы, которые могут возникнуть при различных ошибках в этом процессе.



Висячие ссылки и мусор

Различные ошибочные сценарии, возникающие в процессе утилизации мусора, могут быть сведены к двум основным проблемам (для простоты изложения будем считать, что программа написана на языке, в котором допускается явное управление памятью):

1. Предположим, что программист создает две различных переменных, указывающих на одну и ту же структуру данных, а затем уничтожает одну из переменных вместе с ее содержимым (т.е. уничтожение памяти вместе с утилизацией). После этого вторая переменная указывает на неопределенную область памяти. Такие переменные называются *висячими ссылками*. Приведем пример на C:

```
void* p = malloc (32000);
q = p;
free (p); // освобождает память, на которую указывает p, но указатель в q не
          // уничтожается и возникает висячая ссылка
```

2. Для избежания первой проблемы можно предложить такую схему работы, в которой уничтожение памяти сводится только к разрушению пути доступа, а физически память не возвращается до тех пор, пока не будет уничтожена последняя переменная, использующая эту память. Однако тогда из-за трудности отслеживания всех путей доступа к данной структуре может возникнуть такая ситуация, когда все переменные будут уничтожены, а память так и не возвращена. В таком случае говорят, что память стала *мусором*. Проиллюстрируем на еще одном примере:

```
void* p = malloc (32000);
p = q; // уничтожает единственный указатель на память, делая ее мусором
```

Можно сказать, что висячие ссылки возникают в тех случаях, когда память утилизируется "слишком быстро" (т.е. раньше, чем память действительно перестает использоваться), а мусор — когда память утилизируется "слишком медленно" (т.е. позже, чем она могла бы быть возвращена). Висячие ссылки более опасны, так как могут приводить к некорректной работе программы, в то время как появление мусора вполне допустимо. Борьбу с мусором обычно возлагают на специальный процесс, называемый *сборкой мусора (garbage collection)*.

Статическая и динамическая память

- С точки зрения компилятора необходимо различать два класса информации:
  - *Статическую*, т.е. известную во время компиляции
  - *Динамическую*, т.е. информацию, которая становится известной только во время выполнения
- Соответственно, необходимо использовать оба этих механизма при распределении памяти
- У создателя компилятора остается некоторая свобода выбора метода распределения памяти в "пограничных" случаях

Статическая и динамическая память

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

- *Статическая информация*, т.е. информация, известная во время компиляции
- *Динамическая информация*, т.е. сведения, неизвестные во время компиляции, но которые станут известны во время выполнения программы

Например, значения констант в строго типизированных языках известны уже во время компиляции, в то время как значения переменных в общем случае становятся известными уже только во время выполнения программы. Статически мы знаем количество веток в операторе switch, но определить, какая из них выполнится, мы сможем уже только во время выполнения программы.

В приложении к управлению памятью разделение всей информации на статическую и динамическую позволяет определить, каким механизмом распределения памяти необходимо пользоваться для той или переменной, структуры или процедуры. Например, размер памяти, необходимой под простые переменные, можно вычислить (и, соответственно, выделить необходимую память) уже во время компиляции, а вот память, запрашиваемую пользователем с размером, заданным с помощью переменной, придется выделять уже во время выполнения программы. Понятно, что статическое распределение памяти при прочих равных условиях предпочтительнее ("дешевле").

Особенно интересны "пограничные" случаи, такие, как выделение памяти под массивы. Дело в том, что размер памяти, необходимой под массивы фиксированного размера, в большинстве современных языках программирования можно посчитать статически. И тем не менее, иногда распределение памяти под массивы откладывают на этап выполнения программы. Это может быть осмысленно, например, для языков, разрешающих описание динамических массивов, т.е. массивов с границей, неизвестной во время компиляции. К таким языкам относятся Алгол 68, PL/I, C#. В этом случае механизм распределения памяти будет одинаковым для всех массивов. А вот в Паскале или C/C++ память под массивы всегда можно выделять статически.

Влияние управления памятью на языки программирования

Разработка практически всех языков программирования ориентирована на ту или иную методику управления памятью:

- Фортран: запрет рекурсивных вызовов подпрограмм ⇒ не нужен стек точек возврата ⇒ статическое управление памятью
- C: исключительное использование явного освобождения памяти
- Java, платформа .NET: сборка мусора

Влияние управления памятью на языки программирования

При выборе того или иного метода распределения памяти необходимо учитывать специфику исходного языка программирования, так как практически все языки программирования ориентированы на какой-то механизм управления памяти, который должен быть основным (или даже единственным) для данного языка.

Например, язык Фортран изначально проектировался таким образом, чтобы для программ на Фортране достаточно было использовать только статическое распределения памяти и, следовательно, написание компиляторов Фортрана упрощалось. Для этого пришлось пожертвовать многими языковыми возможностями, которые сегодня считаются общепринятыми. Например, синтаксис Фортрана легко было бы расширить для включения в него рекурсивных процедур, но тогда при компиляции программ пришлось бы использовать стековый механизм распределения программ.

В языке C подразумевается использование более широкого набора методов управления памятью, но при этом ответственность за управление памятью в большинстве случаев возлагается на самого программиста. Таким образом, достигается некоторое равновесие между создателем компилятора и конечным программистом: сложность написания компилятора несколько увеличивается, но при этом язык предоставляет бóльшие возможности. Программист же получает в свои руки мощный механизм управления памятью. Если программист с ним не справится — тем хуже для программиста.

Наконец, языки Java и C# не предоставляют программисту никакого механизма для явного выделения или освобождения памяти. В этих языках вся ответственность за управление памятью лежит на механизме сборки мусора, а следовательно, на разработчиках компилятора.

Неявное управление памятью

- Тенденция развития современных языков программирования: предоставить программисту только неявные средства управления памятью, через использование возможностей языка.
  - Сказывается постоянное увеличение доступного объема аппаратной памяти
  - Явное управление памятью отвлекает программиста от его основных задач; пусть все занимаются своим делом!
  - Сходство с "проблемой языков высокого уровня"

Неявное управление памятью

В последнее время в развитии языков программирования наблюдается отчетливая тенденция к усложнению средств управления памятью, требуемых при реализации транслятора, в частности, необходимость реализации сборки мусора. Параллельно с этим происходит полный или частичный отказ от предоставления программисту явных средств управления памятью.

Естественно, что у такого подхода есть как свои преимущества, так и свои недостатки. Среди недостатков следует особо выделить потенциальное замедление программ, использующих сборку мусора, что может отрицательно сказаться на приложениях, работающих в масштабе реального времени. Однако проблема нехватки памяти становится все менее острой по мере удешевления аппаратной памяти: большинство приложений попросту не будет испытывать потребности в дополнительной памяти. Это особенно приятно, так как такие приложения практически не несут никаких накладных расходов на управление памятью.

Кроме того, неявное управление памятью освобождает программиста от большого объема рутинной работы по отслеживанию занимаемой и возвращаемой памяти, так как все проблемы этого процесса "скрываются" от него компилятором. Это значит, что с программиста снимается ответственность за еще один системный вопрос, не относящийся напрямую к его основным задачам и он сможет больше времени посвящать собственно прикладным вопросам.

Приблизительно такой же подход к "скрытию чрезмерной сложности от программиста" обеспечил в свое время широкое распространение языков высокого уровня. Вряд ли переход на неявное управление памятью приведет к такому же резкому увеличению производительности, как при переходе с ассемблера на языки высокого уровня, но упрощение работы программиста должно быть заметным.

Неявное управление памятью

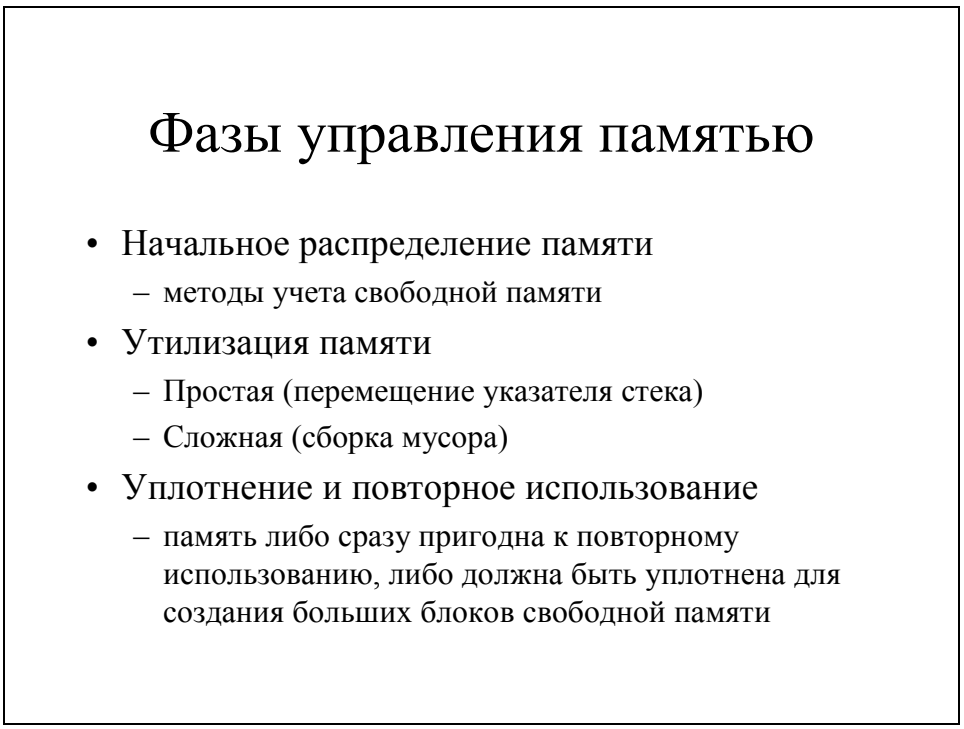
- Другие аргументы в пользу неявного управления памятью:
  - Все равно программист не станет управлять временными переменными!
  - Трудности совмещения двух механизмов управления памятью (система/программист)
- История повторяется: в 70-х годах считали, что неявное управление памятью окончательно вытеснило все остальные механизмы управления памятью

Неявное управление памятью

Другая аргументация в пользу неявного управления памятью связана с тем, что как уже говорилось выше, программист в любом случае не сможет управлять всей памятью целиком — трудно представить себе современного программиста, самостоятельно расставляющего точки возврата из процедур и распределяющего регистры под временные переменные. В то же время наличие одновременно двух механизмов управления памятью (системного и программистского) может существенно усложнить работу компилятора или привести к необходимости создания отдельных областей памяти и программ для поддержки явных механизмов управления памятью. В конечном итоге это приводит к уменьшению эффективности управления памятью в целом.

Итак, неявное управление памятью сегодня является наиболее популярным методом, но для справедливости отметим, что из этого не следует делать далеко идущих выводов. Теория и практика языков программирования активно развиваются, и то, что находится в забвении сегодня, может стать популярным завтра. Так, в середине 70-х годов большинство исследователей было уверено, что неявное управление памятью окончательно вытеснило все остальные методы. Единственным представителем явного управления памятью был уже не очень популярный PL/I с оператором ALLOCATE. При этом стоимость неявного управления казалась уже вполне приемлемой.

Однако с появлением языка C и резким нарастанием его популярности ситуация кардинально изменилась. Во главу угла стали ставить эффективность и предоставление программисту широких возможностей по влиянию на системные процессы. В результате, системы со сборкой мусора в течение ближайших 20-25 лет преимущественно оставались уделом академических исследователей. Только дальнейшее удешевление аппаратуры и желание избавиться от ошибок управления памятью помогли неявному управлению памятью возвратить себе позиции среди практических языков программирования.



Фазы управления памятью

Как мы уже говорили выше, действия любого метода управления памятью можно разделить на некоторые стандартные фазы. Мы будем выделять три крупных этапа:

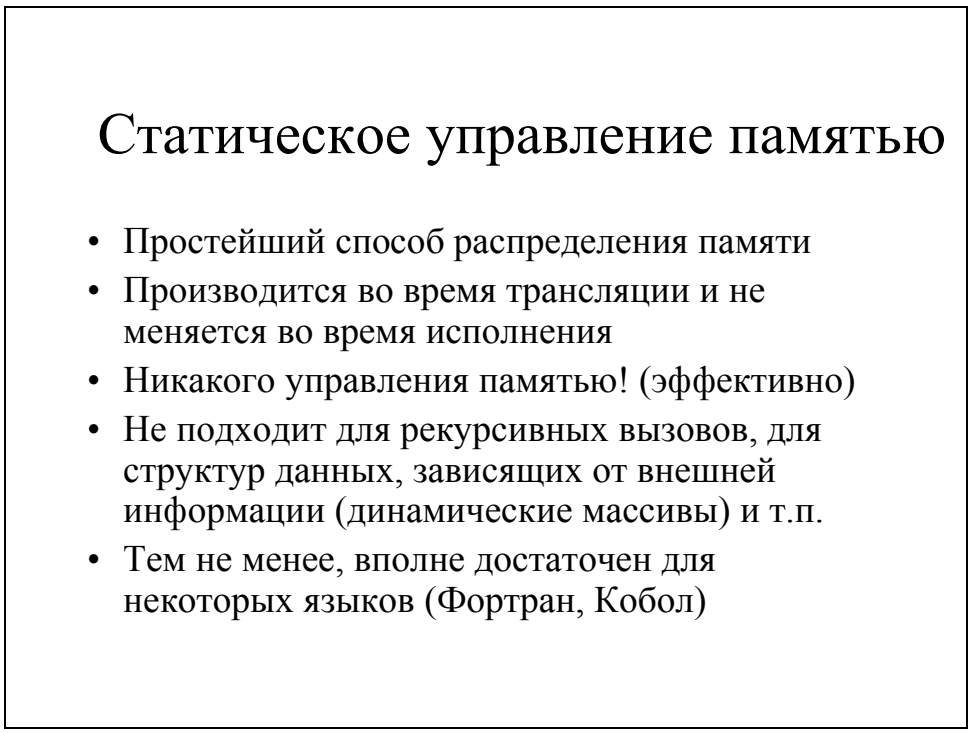
- Начальное выделение памяти
- Утилизация памяти
- Уплотнение и повторное использование

Во время начального распределения памяти необходимо разметить всю память как свободную либо используемую в каких-то целях. Кроме того, необходимо каким-то образом учитывать свободную память. Далее, система должна обнаруживать фрагменты памяти, которые были использованы, но уже стали ненужными. Такие фрагменты подлежат утилизации для дальнейшего повторного использования. Утилизация может быть как простой (в тех случаях, когда освобождаемые участки памяти смежны и непрерывны), так и достаточно сложной (в тех случаях, когда участки освобождаются в различных областях памяти или в случайном порядке). Наконец, память должна быть повторно использована. Для этого, возможно, потребуется применение механизмов уплотнения свободной памяти в целях создания нескольких больших блоков свободной памяти из множества маленьких.

Можно выделить три основных метода управления памятью (мы приводим их в порядке возрастания сложности реализации):

- Статическое распределение памяти
- Стековое распределение памяти
- Представление памяти в виде *кучи (heap)*

Ранее многие языки проектировались в расчете на какой-то один их этих механизмов управления памятью. Сегодня большинство языков программирования требуют от компиляторов использования всех трех механизмов управления памятью — обычно подразумевается применение самого "дешевого" из пригодных способов. В дальнейшем



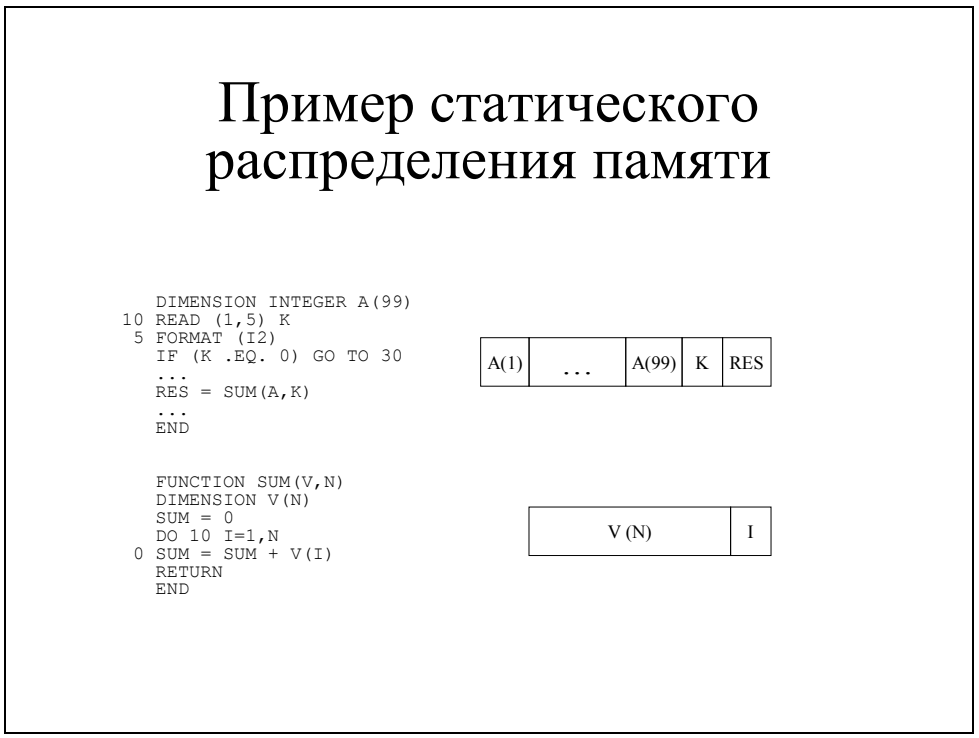
Статическое управление памятью

Статическое управление памятью представляет собой простейший способ распределения памяти и было достаточно распространено на заре развития языков программирования. Если структура языка позволяет обойтись статическим распределением памяти, то удается достичь максимальной эффективности программы в целом, так как во время выполнения не приходится выделять и освобождать память, делать какие-либо проверки и прочие дорогостоящие и часто возникающие операции.

Любопытно, что именно по этой причине "выжил" один из самых ранних языков программирования Фортран. Дело в том, что для Фортрана можно в среднем скомпилировать более эффективную программу, чем для языков с более сложной системой управления памятью, а именно скорость выполнения была главным параметром в основных областях применения этого языка (математические и прочие научные расчеты). Со временем эффективность работы программы стала менее важной, чем удобство программирования, но к тому моменту уже был накоплен огромный багаж работающих программ, и потому Фортран по-прежнему скорее жив, чем мертв.

Как мы уже говорили выше, желание свести все к статическому управлению памятью заставляет разработчиков отказаться от многих конструкций, привычных для программиста. Приведем некоторые примеры: рекурсивные процедуры (так как неизвестно, сколько раз будет вызвана процедура, и непонятно, как различать экземпляры процедуры), массивы с неконстантыми или изменяющимися границами и вложенные процедуры/подпрограммы.

Подводя краткие итоги, в распоряжении программиста оказываются только простые переменные, структуры и массивы фиксированного размера. Именно такой аскетичный набор данных предоставляют Фортран и Кобол. Сегодня это может показаться удивительным, но даже с таким ограниченным набором возможностей люди ухитрялись создавать крупные промышленные системы! Однако не во всех случаях это удобно, например, сложно представить себе процесс написания тех же компиляторов на Коболе.



Пример статического распределения памяти

Рассмотрим статическое управление памятью на небольшом примере на Фортране. В этом языке вся память может быть выделена статически и во время выполнения программы будут меняться только значения простых переменных и элементы массива. Для этого каждая функция транслируется в статически выделенную область памяти, которая содержит сам код и связанные с ним данные, а связь между подпрограммами осуществляется через блоки данных, общие для нескольких подпрограмм (COMMON) или путем передачи параметров и передачи управления при нерекурсивных вызовах. Типы данных могут быть только одного из пяти заранее заданных видов, а переменные никогда не освобождаются.

Все это позволяет прибегнуть к максимально простому способу представления данных в памяти — одномерному массиву переменных. При этом главная функция компилируется независимо от функции SUM и потому для них создаются два различных адресных пространства. Транслированные подпрограммы объединяются уже только во время загрузки.

Любопытно, что в Фортране зафиксирована даже схема представления в памяти многомерных массивов, причем порядок записи в каком-то смысле уникален, так как требует хранить двумерные массивы по столбцам, а не по строкам. Другая особенность Фортрана более неприятна: из-за того, что во время выполнения не производятся никакие проверки, могут быть пропущены серьезные ошибки (скажем, выход за границы массива или извлечение вещественного значения в целую переменную, наложенную на ту же память с помощью оператора EQUIVALENCE).

Практически во всех языках управление памятью включает в себя статическую компоненту, так как этот способ распределения памяти наиболее дешев и не требует накладных расходов во время исполнения. Статически можно распределять константы, переменные и фиксированные массивы. В более сложных языках программирования для распределения памяти нам придется отталкиваться от значений, которые станут известны только во время исполнения.

Стековое управление памятью

- Простейший метод распределения памяти времени выполнения
- Освобождение памяти в обратном порядке
- Задачи утилизации, уплотнения и повторного использования становятся тривиальными
- Используется для точек возврата из подпрограмм и локальных сред
- Подходит для языков со строго вложенной структурой (Pascal, Algol 68, Modula 2)

Стековое управление памятью

Так как статическое распределение памяти чрезмерно ограничивает программиста, проектировщикам языков программирвоания со временем пришлось перейти к более сложным средствам управления памятью, работающим во время выполнения программы. Самым простым из таких методов является *стековое управление памятью*. Его идея заключается 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. Понятно, что задачи утилизации и повторного использования становятся тривиальными, а проблемы уплотнения просто не существует.

При таком подходе все значения каждого блока или процедуры объединяются в единую часть стека, называемую *рамкой*. Для управления памятью нам потребуется *указатель стека*, показывающий на первый свободный элемент, и *указатель рамки*, хранящий адрес дна рамки (это потребуется при выходе из блока или процедуры).

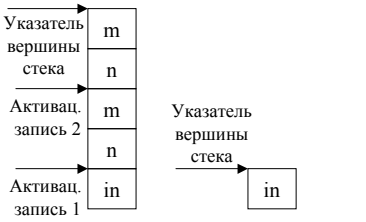
Стековое управление памятью особенно выгодно для языков со строгой вложенной структурой входов в процедуры и выходов из них. Дополнительно необходимо потребовать, чтобы структуры данных имели фиксированный размер, могли создаваться программистом только при входе в процедуру и обязательно уничтожались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

Таким образом, по сравнению со статическим распределением памяти нам удалось снять ограничения на вложенные процедуры и рекурсивные вызовы, но мы по-прежнему требуем от всех переменных фиксированных размеров. Зато при выходе из процедуры компилятор точно знает размер освободившегося блока (следовательно, сколько ячеек памяти надо снять с вершины стека). Языками, пригодными для стекового управления памятью, являются Паскаль, Алгол 68 и Модула 2.

Пример стекового управления памятью

```
var in : integer;
function Digits (integer n) : integer;
var m : integer;
begin
  if n < 10 then return n
  else begin
    m := n div 10;
    return n - m*10 + Digits (m);
  end;
end;

begin
  read (in);
  writeln(Digits(in));
end.
```



Состояние стека после второго вызова процедуры Digits

Состояние стека после возврата в главную процедуру программы

Пример стекового управления памятью памяти

Проиллюстрируем стековый механизм управления памятью с помощью небольшого примера на Паскале. На слайде приведена программа, использующая рекурсивную процедуру Digits для вычисления суммы цифр числа. При этом нам необходимо уметь различать экземпляры процедуры Digits, для этого необходимо создавать независимые активационные записи для различных экземпляров. На первом рисунке показан момент после первого рекурсивного вызова. После возврата в главную программу со стека снимаются не нужные более переменные и указатель вершины стека перемещается вниз (этот момент показан на втором рисунке).

Отметим, что в языках, в которых присутствует динамическая информация, в каждой рамке вначале распределяется вся статически известная память, а все остальные переменные размещаются над статической частью. Таким образом, в момент компиляции мы еще не знаем адреса рамок, но, по крайней мере, мы сможем распределять статические адреса относительно начала определенной рамки. Другой подход к решению проблемы размещения динамической информации связан с размещением в статической части только информации, известной во время компиляции, и указателей на сами переменные в динамическую часть рамки стека или в кучу.

Еще одно замечание связано с вопросом параллелизма: при наличии параллелизма невозможно гарантировать последовательность выходов из процедур и потому чисто стековый механизм управления памятью недостаточен (хотя эту проблему можно решить путем разветвления стека в момент распараллеливания процесса).

Управление кучей

- Куча - это блок памяти, части которого выделяются и освобождаются способом, не подчиняющимся какой-либо структуре
- Куча требуется в тех языках, где выделение и освобождение памяти требуется в произвольных местах программы
- Серьезные проблемы выделения, утилизации, уплотнения и повторного использования памяти
- Самая сложная часть управления кучей – это сборка мусора

Управление кучей

Наконец, последним механизмом управления памятью является *управление кучей*. Этот механизм предназначен для работы со всеми структурами данных, которые по тем или иным причинам не пригодны для статического или стекового распределения памяти. Потребность в куче возникает всякий раз, когда выделение и освобождение памяти может потребоваться в непредсказуемый момент времени. Более того, большинство современных объектно-ориентированных языков программирования попросту не могут обойтись без динамического выделения и уничтожения объектов. Так что хорошо это или плохо, но управление кучей становится одним из основных механизмов управления памятью в современных системах.

При управлении кучей проблемы утилизации и уплотнения памяти становятся весьма серьезными: моменты возврата памяти не всегда очевидны, порядок возврата памяти просто непредсказуем, а память, отведенная для распределения под новые объекты, обычно напоминает решето. Решение этих проблем возлагается на механизм *сборки мусора*. В этом процессе выделенная ранее память пересматривается с целью обнаружения неиспользуемых фрагментов, а высвобожденная память передается для повторного использования. Сборка мусора впервые появилась в трансляторах с языка LISP, в котором концепция управления кучей была практически единственным механизмом управления памятью.

Наверное, самой критичной проблемой управления кучей является отслеживание активных элементов в памяти: как только возникнет потребность в освобождении дополнительной памяти, процесс сборки мусора должен будет утилизировать все неиспользуемые более фрагменты памяти. Определить, используется ли данный момент в программе, можно несколькими способами. Наиболее распространенными из них являются счетчики ссылок и различные алгоритмы разметки памяти.

Отслеживание свободной памяти с помощью подсчета ссылок

Суть метода:

- В каждом элементе памяти заводится место для счетчика ссылок на данный элемент.
- При создании новых ассоциаций с данным фрагментом памяти значение счетчика увеличивается на единицу
- По окончании времени жизни переменных, указывающих на данный фрагмент памяти, счетчик ссылок уменьшается на единицу

Проблемы счетчиков ссылок:

- Проблема циклических списков
- Большие накладные расходы (подсчет ссылок происходит постоянно, даже если не возникает потребности в сборке мусора)

Отслеживание свободной памяти с помощью подсчета ссылок

Самый простой способ отслеживания свободной памяти заключается в приписывании каждому объекту в памяти специального *счетчика ссылок*, показывающего количество "живых" переменных, использующих данный объект. При первичном выделении памяти в программе счетчику присваивается значение, равное единице; при создании новых указателей на данный фрагмент памяти, значение счетчика увеличивается на единицу. Если какая-то из переменных, указывающих на данный фрагмент памяти, перестает существовать, значение счетчика ссылок уменьшается на единицу. При достижении счетчиком нуля фрагмент памяти считается более не используемым и может быть утилизирован. Отметим также, что в эту схему легко вписывается и явное управление памятью: оператор free приводит к простому уменьшению счетчика ссылок на единицу.

Этот метод прост, понятен и легко реализуется, но, к сожалению, у этого метода есть серьезные недостатки. Во-первых, приведенный выше алгоритм в некоторых случаях не справляется с определением свободной памяти. Например, для циклического списка уничтожение внешнего указателя на эту структуру делает ее мусором, и в то же время счетчики ссылок циклического списка не становятся равными нулю, т.к. элементы списка указывают друг на друга.

Во-вторых, использование механизма счетчиков ссылок связано со значительной потерей эффективности во время исполнения, так как при каждом присваивании в программе необходимо производить соответствующие арифметические операции. Все эти действия могут оказаться совершенно бесполезными, если программе хватит исходного объема памяти, а это противоречит одному из принципов трансляции, согласно которому программы не должны терять в производительности из-за языковых или компиляторных механизмов, которыми они не пользуются.

Из-за этих проблем механизм счетчиков ссылок не получил широкого распространения. Сегодня он используется только в редких случаях и обычно для крупных объектов, т.к. для них дополнительные затраты памяти не так заметны, а проблема цикличности неактуальна (см., например, счетчики ссылок COM-объектов на платформе Windows).

Отслеживание свободной памяти с помощью разметки

- Суть метода:
  - При возникновении необходимости в свободной памяти мы отмечаем все заведомо живые переменные и затем проводим транзитивное замыкание, помечая все, что связано с ними
- Проблемы разметки памяти:
  - Потери производительности возникают только в тех программах, которые действительно не могут обойтись без сборки мусора, но разметка памяти – крайне медленный процесс

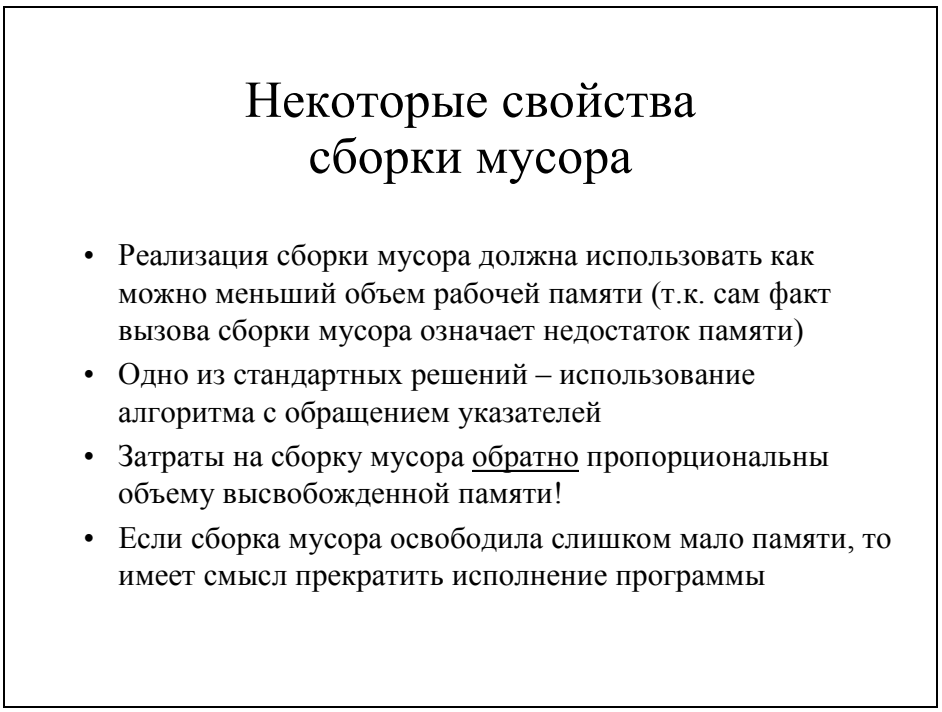
Отслеживание свободной памяти с помощью разметки

Другим методом отслеживания свободной памяти является механизм *разметки памяти*. При этом подходе все действия по поиску неиспользуемых переменных откладываются до возникновения недостатка памяти, т.е. до того момента, когда программа требует выделить фрагмент слишком большого размера. В этот момент стартует процесс сборки мусора, начинающийся именно с разметки памяти.

Разметка памяти начинается с обнаружения всех заведомо живых элементов программы. К таким причисляются все объекты за пределами кучи (на стеке, в регистрах процессора и т.д.), а также все объекты в куче, на которые они указывают. Все эти элементы помечаются как используемые. Затем мы перебираем все используемые элементы и помечаем все прочие объекты, на которые они ссылаются. Этот процесс повторяется рекурсивно до тех пор, пока мы не перестаем находить новые используемые элементы.

Следующий просмотр сборки мусора утилизирует все элементы, не помеченные как живые, а также уплотняет все живые элементы, сдвигая их в начало кучи. Очевидно, что затем сборщику мусора придется поменять все значения указателей, используемых в программе; в связи с этим на время сборки мусора все остальные процессы приостанавливаются.

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 скорее всего, все процессы будут заморожены на некоторое время, которое, скорее всего, будет заметно даже пользователю.



Некоторые свойства сборки мусора

Сборка мусора обладает рядом интересных свойств. Начнем с того, что реализация сборки мусора всегда нетривиальна, так как к ней одновременно предъявляются требования эффективности (иначе процесс сборки мусора будет заметно тормозить исполнение программы) и использования минимально возможного объема памяти (так как сам факт вызова сборки мусора уже означает недостаток памяти). Прямолинейный обход всех достижимых элементов потребует специальной дополнительной памяти под стек значений, что нежелательно, так как эта память будет потеряна для обычных приложений.

Для решения этой проблемы Шорром и Уэйтом еще в 1968 году был предложен алгоритм с *обращением указателей*: во время маркировки каждого следующего элемента мы будем обращать указатель на него, а при достижении последнего достижимого элемента вернемся по списку в обратном направлении, опять-таки обращая указатели. При такой схеме обхода достаточно всего двух переменных для хранения рабочих указателей и одного дополнительного поля в каждом обрабатываемом элементе (для хранения отметки, обработан этот элемент или нет). Однако этот алгоритм работает заметно медленней. Современные методы маркировки обычно используют какое-то сочетание приведенных выше методов. Более подробно эта тема освещена в книге Д. Кнута "Искусство программирования", том 1, раздел 2.3.5 "Списки и сборка мусора".

Другая интересная особенность сборки мусора заключается в том, что затраты на ее исполнение *обратно* пропорциональны объему высвобожденной в результате памяти. Это связано с тем, что основные затраты во время сборки мусора приходятся именно на фазу маркировки и, следовательно, чем больше активных элементов в куче, тем процесс дороже. В предельных случаях — когда в результате сборки мусора освобождается совсем мало памяти — программа практически прекращает полезную деятельность, так как для продолжения работы ей снова и снова приходится прибегать к сборке мусора, но скорее всего, никакого улучшени при этом не происходит (в таких случаях говорят, что программа "жужжит"). По этой причине многие алгоритмы сборки мусора ставят для себя специальную нижнюю границу: если алгоритму не удалось освободить больше некоторого заранее заданного объема памяти, то программа тут же завершается.

Поколения объектов

Исследования показали, что для большинства программ верны следующие предположения:

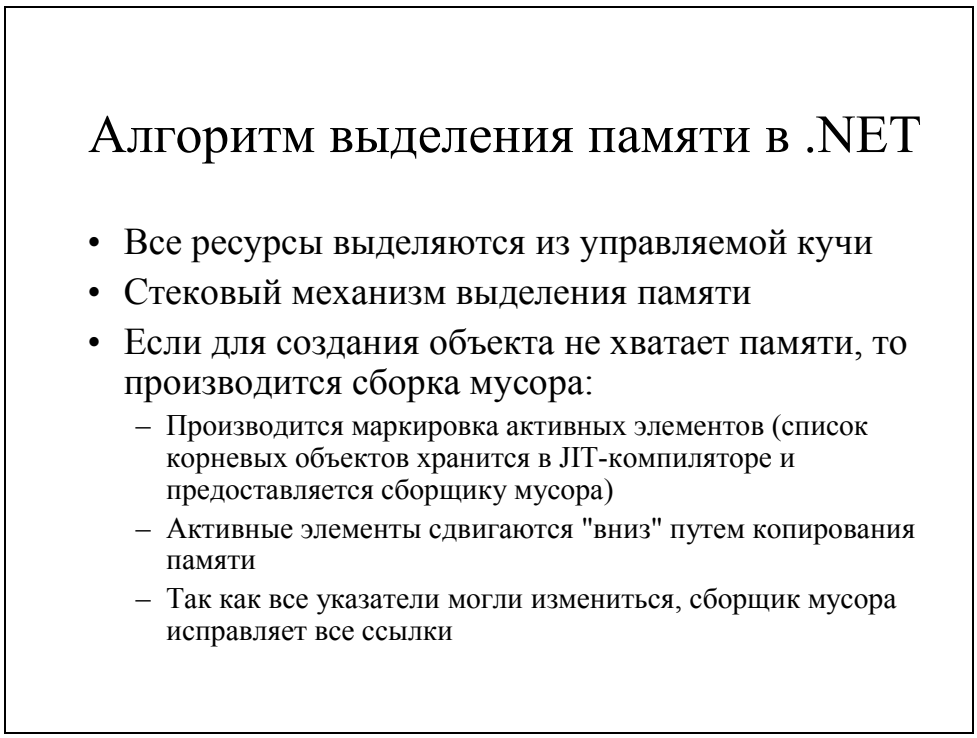
- Чем моложе объект, тем меньше его ожидаемое время жизни
- Чем старше объект, тем больше его ожидаемое время жизни
- Молодые объекты зачастую сильно связаны друг с другом и обычно используются почти одновременно
- Сжатие части кучи обычно быстрее, чем сжатие всей кучи

Таким образом, можно оптимизировать алгоритм сборки мусора путем использования методики поколений

Поколения объектов

Упомянем еще одну распространенную оптимизацию, применяемую при сборке мусора: учет поколений объектов. Практические исследования показали, что в современных языках программирования активность объекта зависит от его возраста, причем нетривиальным образом: чем старше объект, тем больше его ожидаемое время жизни. Подавляющее большинство элементов программы используются сугубо локально, чаще всего все время жизни переменной в программе (от описания до последнего использования) умещается в пределы одного блока, процедуры или метода. Кроме того, типичный объект связан, в основном, со своими "сверстниками" и чаще всего умирает вместе с ними. Исходя из этого, логично предположить, что сборка мусора будет наиболее эффективной при обработке недавно созданных объектов. Действительно, большинство объектов утилизируются при первом же вызове сборки мусора.

Поэтому в целях уменьшения времени работы современные алгоритмы сборки мусора специально помечают объекты, пережившие сборку мусора. Такие объекты объединяются в *поколения*: к нулевому объекту относят объекты, не пережившие еще ни одной сборки мусора, к первому поколению – объекты, пережившие одну сборку мусора и т.д. В дальнейшем сборщик мусора постарается ограничиться обработкой только нулевого поколения (например, во время разметки памяти сборщик мусора не перебирает указатели из "старых" объектов). Если при этом высвобождено достаточно памяти для дальнейшей работы, то сборка мусора на том прекращается, если же нет, то производится сборка мусора в первом поколении, затем при необходимости во втором и т.д.



Алгоритм выделения памяти в .NET

Основными механизмом работы с памятью в .NET являются неявное управления памятью, стековый механизм выделения памяти и сборка мусора, использующая поколения и механизм "разметки-и-уплотнения" (mark-and-compact). Сразу оговоримся, что платформа .NET содержит также средства, позволяющие обойти эти ограничения, например, C# содержит специальный оператор fixed и некоторые средства управления механизмом сборки мусора, но в данной лекции мы их проигнорируем.

Для начального выделения памяти простой стековый механизм: имеется один указатель на следующее свободное место в куче, который после помещения в кучу очередного объекта увеличивается на его размер. Понятно, что в какой-то момент указатель кучи может выйти за пределы доступного адресного пространства — в этот момент начинает работу алгоритм сборки мусора. В целях оптимизации процесса сборка мусора чаще всего ограничивается обходом нулевого поколения – чаще всего этого оказывается достаточно.

Для сборки мусора производится маркировка активных элементов; она начинается с так называемых корневых объектов, список которых хранится в JIT-компиляторе .NET и предоставляется сборщику мусора. По окончании маркировки все активные элементы сдвигаются к началу кучи путем простого копирования памяти. Так как эта операция компрометирует все указатели, сборщик мусора также исправляет все ссылки, используемые программой.

Реально алгоритм сборки мусора, используемый в .NET, существенно сложнее, так как включает в себя такие оптимизации как слабые ссылки, отдельную кучу для крупных объектов, сборку мусора в многопоточных приложениях и т.д. Подробное описание алгоритма сборки мусора .NET можно найти в статье Дж. Рихтера "Automatic Memory Management in the Microsoft .NET Framework", MSDN Magazine, Nov/Dec 2000.

Пример на сборку мусора

```
using System;
using System.Collections;

class GarbageClass {
    public static int Main(String[] args) {

      // ArrayList object created in heap, myArray is now a root
      ArrayList myArray = new ArrayList();

      // Create 10000 objects in the heap
      for (int x = 0; x < 10000; x++) {
         myArray.Add(x.ToString()); //Object created in heap
      }

      foreach (object currValue in myArray)
      {
         string s = (string)currValue;
         Console.WriteLine (s);
      }

      System.GC.Collect (GC.GetGeneration(myArray[0]));
      return 0;
   }
}
```

Пример на сборку мусора

Проиллюстрируем основные концепции сборки мусора на простом примере. На слайде приведен класс GarbageClass, главный метод которого создает список myArray, а затем в цикле добавляет в этот список 10 000 строковых объектов. Все эти объекты создаются в куче; время их жизни распространяется до конца метода Main.

Затем все элементы myArray перебираются в цикле foreach, и значение каждого элемента печатается на экран. При этом на каждой итерации цикла происходит создание объекта в куче (это, конечно же, жутко неэффективно – по-хорошему надо было бы передавать currValue непосредственно в метод WriteLine, но мы как раз хотим продемонстрировать пример, в котором *много* мусора).

Наконец, непосредственно перед выходом мы явным образом вызываем сборщик мусора, причем уточняем, что хотим произвести сборку только в том поколении объектов, к которому принадлежит начальный элемент myArray. В нашем случае это не очень важно, но в приложениях с большим объемом вычислений это может оказаться удобным.

Интерфейс компилятора со сборщиком мусора

- Компилятор должен предоставить сборщику мусора информацию, необходимую для его корректной работы (например, сборщик мусора должен иметь возожность отличить в сгенерированном коде обычное целое число от указателя)
- Но в .NET необходимости в таком интерфейсе не возникает, т.к. сборщик мусора может получить всю необходимую ему информацию во время исполнения

Интерфейс компилятора со сборщиком мусора

В общем случае, между компилятором и сборщиком мусора должен существовать некоторый интерфейс, который обеспечивал бы передачу всей информации, необходимой для работы сборщика мусора. К такой информации можно отнести, например, адреса корневых объектов, схема размещения и выравнивания записей в куче, информация о переменных, содержащих указатели и т.п.

Однако в .NET необходимости в таком интерфейсе не возникает, так как сборщик мусора может самостоятельно получить всю необходимую ему информацию во время исполнения программы от библиотек динамической поддержки .NET. Это становится возможным благодаря тому, что .NET использует единую систему типов (Common Type System, см. лекцию 1) и запрещает потенциально опасные преобразования данных. Единственная трудность, связанная с использованием такого решения, может возникнуть при реализации компиляторов с языков, обладающих системой типов данных, отличной от .NET. В этом случае типы данных, специфические для реализуемого языка придется эмулировать с помощью типов .NET или путем создания специального класса, реализующего необходимую функциональность.

С другой стороны, в компиляторах, генерирующих ассемблерный код, интерфейс между компилятором и сборщиком мусора практически неизбежен, так как даже в простейших случаях невозможно провести различие между некоторыми типами данных – например, непонятно, как можно определить по 4-байтовому значению, является ли оно простым целым числом или указателем. Информация такого рода теряется в процессе компиляции и потому должна специально сохраняться для сборщика мусора.

Литература к лекции

- Д. Кнут "Искусство программирования", Вильямс, 2000
- J. Richter "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework", Parts 1 & 2, MSDN Magazine, Nov. 2000/Dec. 2000
- Microsoft C# Language Specification, Microsoft Press, 2001

Литература к лекции

• Д. Кнут "Искусство программирования", Вильямс, 2000
• J. Richter "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework", Parts 1 & 2, MSDN Magazine, November 2000 / December 2000
• Microsoft C# Language Specification, Microsoft Press, 2001