Информатика

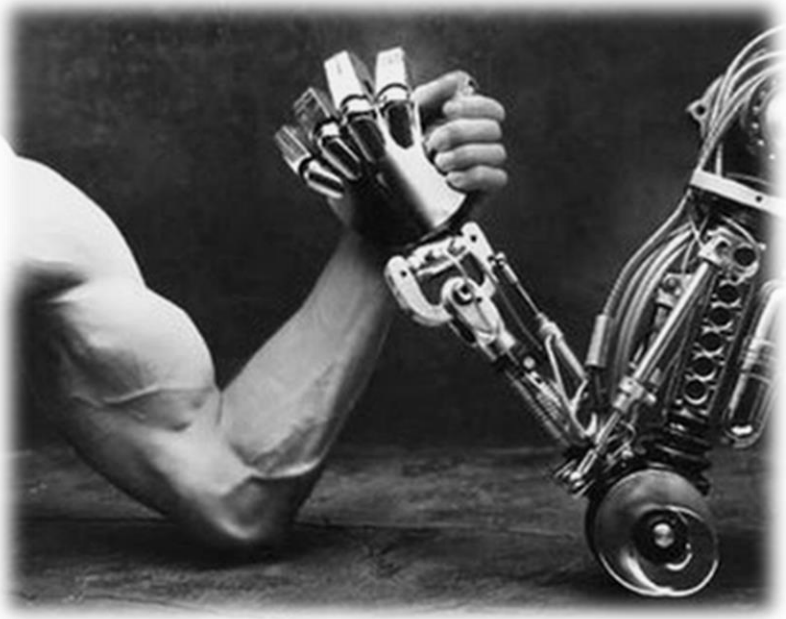
управление памятью

Управление памятью

- ***Память*** *– важнейший ресурс, требующий тщательного управления*
- От эффективности управления памятью зависит *производительность* приложения
- От правильности управления памятью зависит *корректность* приложения в целом

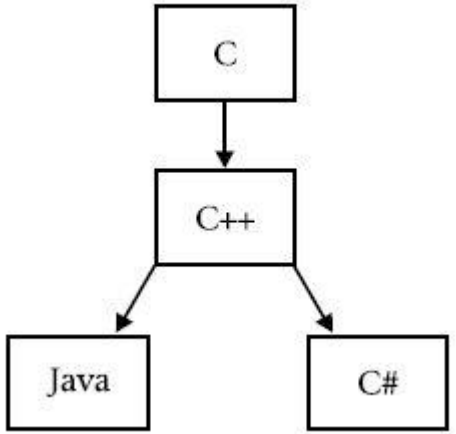
Кому доверить управление памятью?

- ***Программисту?***
  - \+ эффективнее расходуются ресурсы
  - – выше риск совершения ошибки, трудность поддержки
- ***Алгоритму?***
  - – не эффективны при больших объемах памяти, вносят нестабильность и требуют времени для работы
  - \+ человеческий фактор сходит на нет



Подходы к управлению памятью

- Рассмотрим подходы к управлению памятью от ручного и полуавтоматического к полностью автоматическому
- Обсудим идеи от C к C++ к C#
- Рассмотрим преимущества и недостатки



Повторение – мать учения

- Память под локальные переменные выделяется из стека и освобождается при завершении метода(функции)
  - *размер выделяемой памяти известен на этапе компиляции*
- Стек – мал, поэтому основные данные программы хранятся в куче
  - *выделяются динамически*

Особенности языка Си

- Относительная простота
- Процедурный подход
- Препроцессор и макросы
- Работа с памятью через указатели
- Структуры и объединения для составных данных

Хранение данных в Си

- В Си у программиста есть возможность контроля над расположением данных
- Можно работать с данными на стеке или размещать их в куче
  - утверждение справедливо и для массивов

Виртуальная память

- ОС использует виртуальную память для выделения памяти программам
- Каждый процесс считает что работает со всей физической памятью
- Согласовывается использование памяти программами
- Предоставляется возможность использовать больше памяти, чем физически установлено в системе

Виртуальная память

- Используемые программами ***виртуальные*** *адреса отображаются в* ***физические*** *адреса памяти* компьютера
  - оперативная и диск
- ОС хранит *таблицу соответствий* виртуальных адресов с физическими для обработки *запросов по адресу*

Ручное управление памятью

- Использование простых аллокаторов
  - поиск
  - выделение
  - освобождение
- С ростом сложности программы растет роль аллокатора

Указатели

- Ручное управление памятью связано с работой с адресами
- **Указатель – тип, для хранения адресов**
- В x86 системах адресное пространство ограничено 2^32 байт, поэтому указатель хранит 32 битные адреса
- В x64 указатель хранит 64 битные адреса

Работа аллокатора

Аллокатор может:

- динамически выделять в памяти указанное количество байт и возвращать указатель на первый из них
- по указателю освободить ранее выделенную память

Динамическое выделение в C

Стандартная библиотека языка C содержит функции:

- void* malloc(long numbytes)
  - выделение памяти
- void free(void* firstbyte)
  - освобождение памяти
- Есть еще calloc (clear allocation) и realloc

Арифметика указателей

- Указатели соответствуют адресам в линейном пространстве памяти
- Адреса можно сравнивать
- Можно указывать смещение относительно текущего положения
  - пример итератора на указателях

Операции

- Взятие адреса
  int x=5; int * y = &x;
- Разыменование (dereferencing)
  *y = 7;
- Индексатор
  int * arr = new int[10]; arr[1]=5; *(arr+1)=5;
- Доступ к члену через указатель
  Student* s=new Student(…); … s->Name

Пара трюков

- Обращение к байтам числа типа int

```
int x = 5; byte* b=&x; //b[0]
```

- Доступ к байтам структуры
  (в C++ даже к private полям)

- Жадная матрица

```
int * arr=new int[100];
int** matrix=new int*[10];

for(int i=0; i<10; i++) matrix[i]=arr+i*10;
```

Нулевой указатель

- Константа, показывающая, что переменная-указатель не указывает ни на какой объект
- В C и C++ это 0 или макрос NULL
- В C++11 рекомендуется nullptr
- В Pascal, Ruby nil
- В C# и Java null

Проблемы

- *Указатель* хранит только адрес первого байта и *не имеет автоматической проверки границ*
- malloc возвращает указатель void*, программисту нужно самому привести его к нужному типу

Логические ошибки

- Двойное освобождение памяти
- Нарушение границ (переполнение буфера, чтение за границами буфера)
- Разыменование нулевого указателя
- Использование неинициализированного указателя

Утечки памяти

- Уменьшение объема свободной памяти, связанный с не освобожденными вовремя участками памяти
- Может возникать при потере значения указателя
- Дикий (подвисший) указатель (dangling pointer) – не ссылающийся на существующий объект

Дикие указатели

```
char* dp = NULL;
{
    char c;
    dp = &c;
}
```

c пропадает из области видимости,
dp становится диким указателем

- Другой пример – обращение к значению указателя после освобожнения памяти

Особенности языка C++

- Развивает идеи Си
- Работа с памятью с помощью указателей и ссылок
- Мультипарадигменный с упором на ООП (начался с идеи «Си с классами»)
- Как и в языке Си, в C++ программист может размещать объекты как в стеке, так и в куче

Динамическое выделение в C++

- В C++ введены два аллокатора в дополнение к malloc, free, realloc и calloc
- **new, delete** – выделение и освобожнение памяти любого указанного типа
- **new [], delete[]** – выделение и освобождение массива данных любого указанного типа

```
int * ptr_i = new int;          struct person *human = new struct person;
double *ptr_d = new double[10];
delete ptr_i;    delete[] ptr_d;    delete human;
```

Типобезопасность указателей

- В C++ указатели типизированы
- Легче выделять память, не нужно вызывать sizeof для типов
- Однако, нет никакой гарантии что указатель не будет преобразован к недопустимому типу

Путаница в способах выделения

- Использование несоответствующих друг-другу способов выделения и освобождения памяти (разных аллокаторов) – ошибка!
- По указателю не понятно каким аллокатором выделялась память
- Даже не понятно, указатель указывает на массив или на отдельный элемент

Ссылки и указатели

- В C++ появились ссылки для упрощения работы со значениями без необходимости их копирования и выполнения взятия адреса и разыменования указателей
- Ссылка в C++ обязательно должна быть проинициализирована

Примеры

- Использование с локальными переменными
  int x=5; int& rx = x; rx=7;
  Использование с параметрами и возвращаемыми значениями функций
  void swap<T> (T&x, T&y)…
  int x=5, y=3; swap(x,y);

Борьба с утечками памяти

- Уменьшить возможность совершения ошибки по неосторожности
- Автоматизировать и локализовать выделение и освобождение памяти, сохранив контроль

Resource Acquisition Is Initialization

- Получение ресурса – инициализация
  Освобождение – уничтожение объекта
- Идея – оборачивание разделяемого объекта/ресурса в оболочку на стеке, чтобы с завершением работы функции, содержащей объект-оболочку, освободился ресурс

Конструкторы и деструкторы C++

- **Конструктор** – *функция, вызываемая при создании объекта*
  - при вызове new для создании объекта в куче
  - при вызове функции, содержащей объявление локального объекта
- **Деструктор** – *функция, вызываемая при удалении объекта*
  - при вызове delete
  - при завершении функции, выходе из области видимости

RAII в C++

Идея автоматических объектов

- При вызове конструктора захватывается/получатся доступ к ресурсу (инициализация)
- При вызове деструктора по выходу объекта из области видимости освобождается ресурс

Автоматизация управления памятью

- Идея автоматических объектов получила дальнейшее развитие
- В C++11 были стандартизированы инструменты, моделирующие работу классических указателей на автоматических объектах, исключающих необходимость ручного управления памятью

Умные указатели в C++

- **unique_ptr, shared_ptr** и **weak_ptr**
- Уменьшают вероятность ошибки при неправильном использовании указателей, сохраняя эффективность
- Предотвращают большинство ситуаций, приводящих к утечке памяти (но не все)

Unique pointer

- Автоматический объект, контролирующий исключительное владение указателем
- Предотвращает копирование (явно удалены конструктор копирования и оператор присваивания)
- Для передачи указателя другому объекту используется std::move

Unique pointer

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; //Compile error.
std::unique_ptr<int> p3 = std::move(p1);
//Transfers ownership.
p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

Shared pointer

- Автоматический объект, предоставляющий возможности одновременного владения нескольких объектов общим ресурсом-указателем
- Ведёт учёт владения, при отсутствии владельцев, освобождает память, на которую ссылается указатель

Автоматический подсчет ссылок

- Реализует учёт владения подсчётом количества владельцев
- Вызов конструктора увеличивает счётчик, деструктора – уменьшает
- Если при вызове деструктора счетчик обнуляется, освобождается память

Shared pointer

```
std::shared_ptr<int> p1(new int(5));
std::shared_ptr<int> p2 = p1;
//Both now own the memory.
p1.reset(); //Memory still exists, due to p2.
p2.reset();
//Deletes the memory, since no one else owns it
```

Циклические ссылки

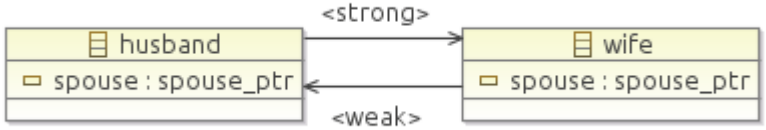
- *Circular reference*
- Если несколько объектов ссылаются друг на друга, автоматический подсчёт ссылок не сможет освободить память, выделенную по указателям (счётчики будут не нулевые)

ARC в других языках

Perl, PHP, Objective-C, Swift используют автоматический подсчёт ссылок для управления памятью

Борьба с циклическими ссылками

- Слабые ссылки (weak reference)



- Аналогичны разделяемым умным указателям (shared_ptr), но не увеличивающие счётчик ссылок
- Не препятствуют удалению ссылающихся друг на друга объектов

Сложность ARC

- Использование автоматического подсчёта ссылок для управления памятью требует качественной работки ссылок
- Циклические зависимости могут быть неочевидны (длинные циклы ссылок)
- Могут потребовать дополнительные инструменты для борьбы с утечками памяти

Сборка мусора

- John McCarthy 1959 упрощение управления памятью в Lisp
- Полностью автоматический подход к управлению памятью, решающий проблему циклических ссылок
- Освобождение памяти отдано на откуп сборщику мусора
- Пользователь совсем не заботится об освобождении ресурсов (но и не может проконтролировать процесс и его эффективность)

ARC – тоже форма сборки мусора

- ARC – децентрализованный подход к сборке мусора
- Каждый объект с помощью счетчика ссылок следит за использованием ресурса и сам принимает решение об освобождении

Трассирующий сборщик мусора

- Первая идея к сборке мусора
- Периодически анализируются все ссылки из регистров, стека и глобальных переменных и отмечаются все используемые данные. Неиспользуемые – удаляются
- Требует приостановки выполнения программы для сборки мусора
- В чистом виде не используется

Копирующий сборщик мусора

- Используются две кучи: одна используется, другая в резерве
- При заполнении первой, данные анализируются и копируются во вторую с уплотнением, после чего кучи меняются местами
- Эффективное расположение данных в куче
- Накладные расходы по памяти, необходимость модификации указателей при уплотнении, необходимость приостановки выполнения программы при сборке мусора

Трассирующий и уплотняющий

- Данные хранятся в одной куче
- Когда кончается место, обходятся ссылки, данные перемещаются, ссылки модифицируются
- Среднее между копирующим и трассирующим сборщиком
- Более сложный алгоритм сборки, требует эвристик и оптимизаций
(не уплотнять при фрагментации до 25%)

Сборка по поколениям

- Обычно в ООП большинство объектов перестают использоваться вскоре после создания
- Объекты разделяются по поколениям: новые – первое, пережившие первую сборку – второе, дальше – третье и т.д.
- Старшие поколения проверяются реже

Сложности со сборкой по поколениям

- При сборке проверяются только объекты нужного поколения, более старые пропускаются
- Но объекты могут изменяться
- Объект старшего поколения может сослаться на объект младшего
- Нужно отслеживать это и переносить молодой объект в корневое множество старшего поколения, чтобы исключить ошибочную сборку

Гибридные решения

- На практике обычно комбинируют подходы
- Получается, например, уплотняющий трассирующий сборщик мусора по поколениям, осуществляющий сборку параллельно
- Некоторые языки предлагают на выбор несколько вариантов сборки мусора (более экономные по памяти или быстрее производящие сборку)

Недостатки GC

- GC требует дополнительных ресурсов (времени и памяти)
- Может приводить к непрогнозируемым задержкам исполнения (недопустимым в окружениях реального времени и обработке транзаций)
- Несовместим с ручным управлением
- Освобождает ресурсы недетерминированно
- Несовместим с классическим RAII

Управление памятью в C#

- В обычных ситуациях программисту не нужно специально заботиться об управлении памятью
- Выделение памяти происходит при вызове оператора new
- Освобождается память исключительно сборщиком мусора

Типы значения в C#

- У программиста есть возможность хранить данных в месте объявления (на стеке), используя пользовательские типы значения
- Пользовательские типы значения – структуры
- В некоторых ситуациях эта техника позволяет повысить производительность

Сборщик мусора в C#

- CLR использует трассирующую, сжимающую кучу сборку по поколениям
- GC начинает сборку при выделении памяти (new), после достижения определённого порога памяти или в другое время для стирания «следов» приложения в памяти

Корни

- При сборке мусора, GC просматривает **корни** – *то, что держит объект «в живых»*
  - локальная переменная или параметр
  - статическое поле
  - объект в очереди на финализацию (далее)
- Корни напрямую или косвенно ссылаются на объект и препятствуют сборке
- Объекты, на которые не ссылаются корни – *недостижимые* и подлежат сборке

Large & small object heap

- GC использует отдельную кучу для больших объектов (>85KB)
- По умолчанию, большие объекты не участвуют в сжатии кучи из-за дороговизны операции перемещения

Фрагментация кучи

- Большие объекты приводят к фрагментации кучи – при удалении создают свободные места, которые в дальнейшем трудно заполнять
  - к фрагментации также приводят фиксированные в куче объекты (fixed, pinned)
- Фрагментация может замедлить выделение памяти

OutOfMemoryException

- Если при создании объекта недостаточно места в памяти, будет инициирована сборка мусора
- Если даже после сборки операционная система не может выделить необходимое количество памяти, выбрасывается исключение

Сборка по поколениям в C#

- Основная техника оптимизации GC, сокращающая время сборки
- GC делит управляемую кучу на три поколения: Gen0, Gen1 и Gen2
- Gen0 и Gen1 – поколения с коротким сроком жизни (эфемерные)
  - поэтому, такой сборщик называют эфемерным

Молодое поколение Gen0

- Gen0 – недавно созданные объекты, ещё «не видавшие» сборки мусора
- Gen0 небольшое
  - обычно от нескольких сотен KB до нескольних MB
  - максимум 256MB в x64 CLR
- При заполнении Gen0, начинается его сборка
  - сборка Gen0 происходит относительно часто
- Сборка Gen0 быстрая (~1ms)

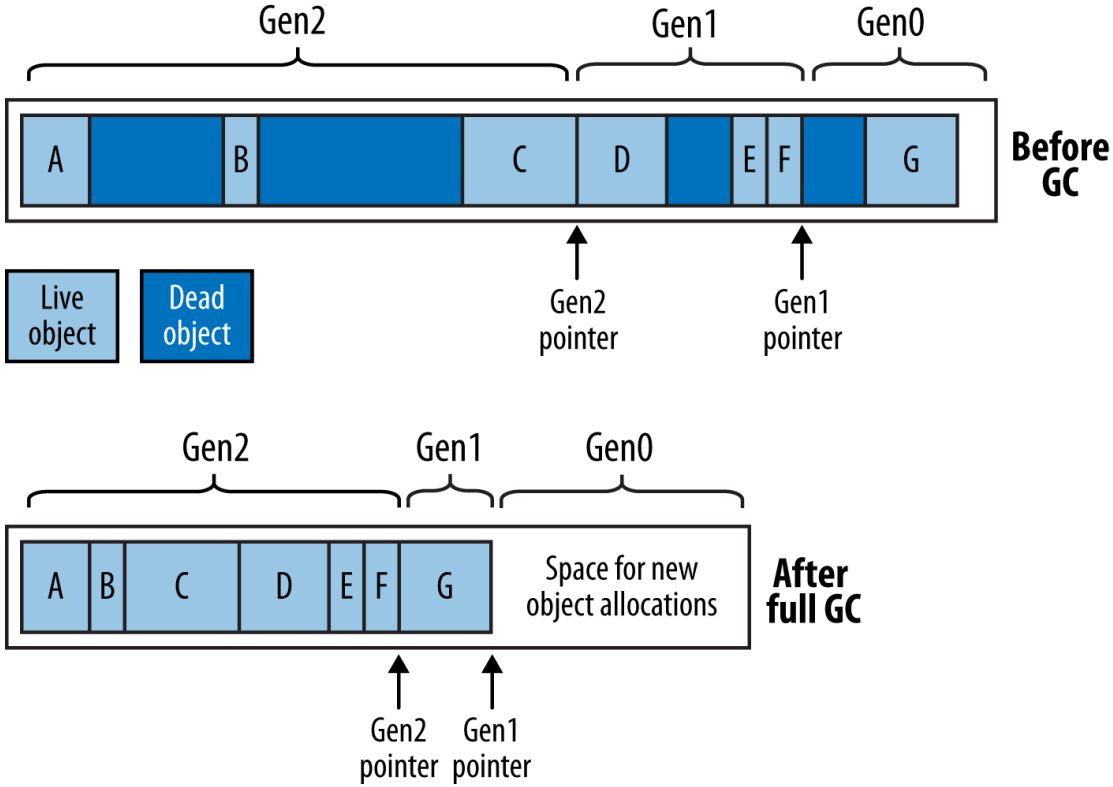
Поколение Gen1

- Gen1 – поколение объектов, переживших одну сборку
- Размер Gen1 схож с Gen0
- Сборка Gen1 тоже происходит относительно часто

Gen2 и полная сборка

- Gen2 – поколение объектов-долгожителей
  - и больших объектов
- Размер Gen2 не ограничен
- GC собирает Gen2 при полной сборке мусора, если сборки эфемерных поколений было недостаточно
- Полная сборка мусора в программе с большим графом объектов занимает ~100ms

Сборка мусора и поколения



Dispose и Finalize

- В некоторых случаях необходимо детерминированное управление ресурсами (нельзя полагаться на GC)
- Используются идеи RAII
- Интерфейс Idisposable
- Применяется как с управляемыми, так и с неуправляемыми ресурсами

IDisposable

- Вызов метода Dispose освобождает ресурс
- При выходе из блока using, вызывается Dispose

```
// Открываем файл внутри блока using
using (FileStream file = File.OpenRead("foo.txt"))
{
    // Выходим из функции при выполнении некоторого условия
    if (someCondition) return;
    // Файл будет закрыт автоматически при выходе из блока using
}
// А что, если кто-то откроет файла вне блока using?
FileStream file2 = File.OpenRead("foo.txt");
```

Недетерминированное освобождение

- Можно в дополнение к детерминированному освобождению ресурса с помощью Dispose, дать возможность описать освобождение в финализаторе (выглядит как деструктор в C++)

Dispose pattern

```
class DataContainer : IDisposable
{
  ...
  public void Dispose()
  {
     Dipose(true);
  }
  ~DataContainer()
  {
       Dispose(false);
  }
```

```
protected virtual void Dispose
(bool disposing)
  {
      if (m_isDisposed)
           return;
      if (disposing)
          m_managedData.Dispose();
      DataProvider.DeleteUnmanagedData
      (m_unmanagedData);
      m_isDisposed = true;
  }
  private bool m_disposed = false;
  private IntPtr m_unmanagedData;
  private IDisposable m_managedData;
}
```

Упрощённая версия паттерна

```
class SomethingWithManagedResources : IDisposable
{
    public void Dispose()
    {
        // Никаких Dispose(true) и GC.SuppressFinalize()
        DisposeManagedResources();
    }

    // Никаких параметров, этот метод должен освобождать
    // только неуправляемые ресурсы
    protected virtual void DisposeManagedResources() { }
}
```

Финализатор и GC

- Объекты с финализатором не могут быть очищены сразу, поскольку нужно вызвать финализатор (который может работать с полями)
- Объекты с финализатором должны пережить сборку
- После сборки CLR в высокоприоритетном finalizer thread запускаяе финализаторы
- После этого, объекты помещаются в очередь на удаление

Одновременная и фоновая сборка

- GC может блокировать потоки исполнения во время сборки
  - блокировка длится всё время сборки Gen0 и Gen1
  - во время сборки Gen2 GC позволяет потокам работать
- Это достигается в workstation CLR за счет использования фоновой сборки (начиная с .NET 4)
  - ранее сборка называлась одновременной и содержала ограничения на по-настоящему одновременную сборку Gen0 во время сборки Gen2

Форсирование сборки

- GC.Collect может инициировать сборку мусора
- Вызов GC.Collect без параметров инициирует полную сборку
- Можно указывать в параметре номер поколения (GC.Collect(0))

Не нужно зря форсировать GC

- Наилучшая производительность достигается при автоматической сборке
- Форсирование может помешать автонастройкам (пороговых значений поколений) и оптимизациям GC

Когда нужно форсирование

- В работающей в фоновом режиме службе, которая запускается по таймеру (System.Timers.Timer) каждые 24 часа (например)
- После завершения ежедневной работы, выполнение прекращается и GC не может выполнить работу
- Решение – форсировать сборку после выполнения работы

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Утечки управляемой памяти

- Причина – неиспользуемые или забытые ссылки
- Обычно источники утечки – статические объекты и обработчики событий

Утечка из-за обработчиков событий

```
class Host
{
    public event EventHandler Click;
}
class Client
{
    Host _host;
    public Client(Host host)
    {
        _host = host;
        _host.Click += HostClicked;
    }
    void HostClicked
    (object sender, EventArgs e) {...}
}
```

```
class Test
{
    static Host _host = new Host();
    public static void CreateClients()
    {
      Client[] clients =
        Enumerable.Range(0, 1000)
        .Select(i =>new Client(_host))
        .ToArray();
      // Do something with clients ...
    }
}
```

В чем проблема?

- Мы ожидаем, что 1000 клиентов будут доступны для сборки после выполнения CreateClients
- Однако, _host в событии Click содержит ссылки на все экземпляры Client

```
public void Dispose() { _host.Click -= HostClicked; }

Array.ForEach(clients, c => c.Dispose());
```

Утечка из-за таймеров

Экземпляры Foo никогда не соберутся GC

Timer хранит ссылку на активных членов, чтобы вызывать события Elapsed

Но Timer : IDisposable
Решение:

```
class Foo : IDisposable
{
    public void Dispose()
    {_timer.Dispose();}
}
```

```
class Foo
{
    Timer _timer;
    Foo()
    {
        _timer = new System.Timers.Timer
        { Interval = 1000 };
        _timer.Elapsed += tmr_Elapsed;
         _timer.Start();
    }
    void tmr_Elapsed(object sender,
                     ElapsedEventArgs e)
    { ... }
```

Слабые ссылки

- Способ решения проблемы с утечками памяти из-за невозможности GC собрать мусор
- WeakReference не мешает сборщику удалить объект

```
var weak = new WeakReference(new StringBuilder("weak"));
Console.WriteLine(weak.Target); // weak
GC.Collect();
Console.WriteLine(weak.Target); // (nothing)
```

- С помощью слабых ссылок можно описать свой делегат и использовать его для реализации событийной модели без утечек памяти

Дополнительная информация

- В C# есть возможность работы не только с управляемыми объектами
- Можно работать также с указателями (в небезопасном **unsafe** контексте)
- Можно взаимодействовать с неуправляемым кодом, настраивать связи с помощью маршалирования
- Можно вручную управлять памятью и даже выделять массивы на стеке (stackalloc)

Вопросы?

e-mail: marchenko@it.kfu.ru