Garbage Collection no *CLR*

- Sumário
  - Libertação manual de memória
  - Gestor de memória com libertação automática (*garbage collection)*
    - *Contagem de referências*
    - *Mark and Sweep*
    - *Mark and compact*
    - *Optimizações*
      - *Detecção com cópia*
      - *Algoritmos geracionais*
  - *Gestão de memória em .NET(modelo computacional)*
    - *Finalização*
    - *O pattern Dispose*
    - *GCHandles*
    - *Referências fracas (weak references)*
    - *Acesso programático*

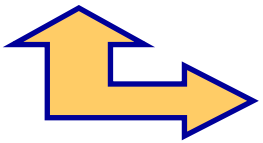
Alocação dinâmica com e sem *garbage collection*

C++ clássico

```
const int nblocos = 1024*1024;
const int size = 1000;

int main(int argc, char* argv[]) {
    try {
      char *ptr;
      for (int j=0; j <nblocos; ++j) {
        ptr = new char[size];
      }
    }
    catch (...) {  cout << "error allocating memory!" <<
    endl;  }
    return 0;

}
```

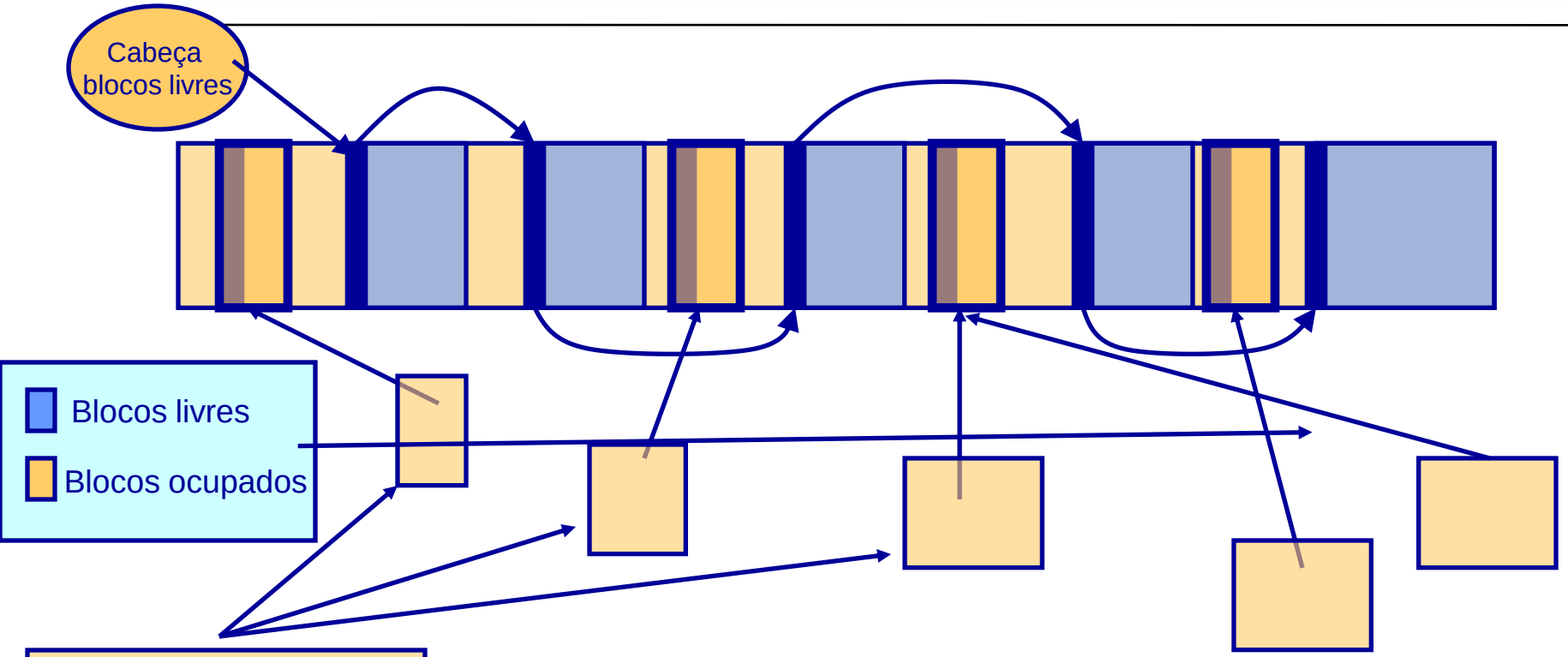


Máximo de memória alocada?

C#

```
public  class AllocChecker {
    readonly static int nblocos = 1024*1024;
    readonly static int size = 1000;
    public static void Main(string[] args) {
       try {
         for (int j=0; j < nblocos; ++j) {
            byte[] byteArray = new byte[size];
         }
       }
       catch (OutOfMemoryException e)  {
          Console.WriteLine(e.ToString());
       }
    }
}
```

Heap tradicional (malloc, free – new, delete)

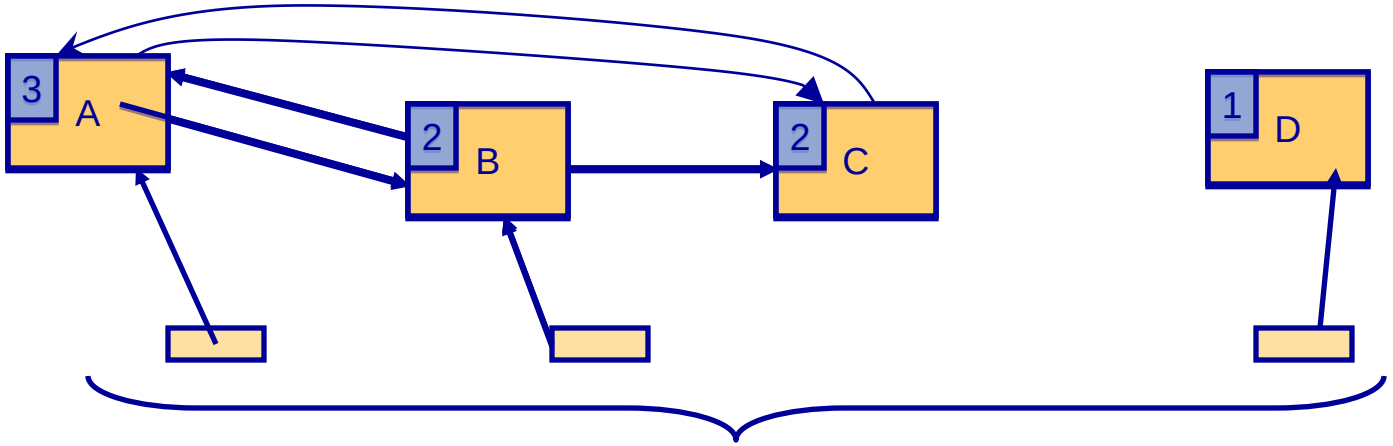


- Cria problemas de fragmentação (interna e externa) de memória
- Ineficiência no processo de alocação/libertação - implica pesquisa
- Potencia o aumento do espaço de endereçamento virtual ocupado pelos processos
- A libertação manual potencia o aparecimento de erros: *leaks* de memória e utilização de memória não alocada

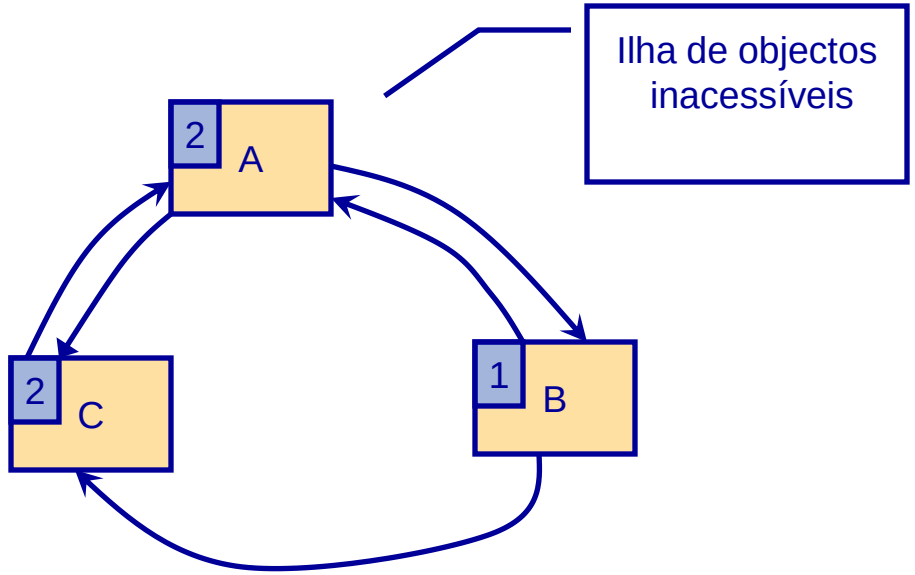
Gestão automática de memória

- Motivações
  - Aumento da robustez do código
    - A sequência de passos necessários para a criação/destruição de objectos é fonte comum de erros
  - Utilização mais eficiente da memória
- Fases
  - Alocação
  - Detecção
    - *Mark*
  - Libertação
    - *Sweep*
    - *Compact*
- Optimizações
  - Detecção com cópia
  - Algoritmos híbridos baseados em *gerações*

Libertação por contagem de referências

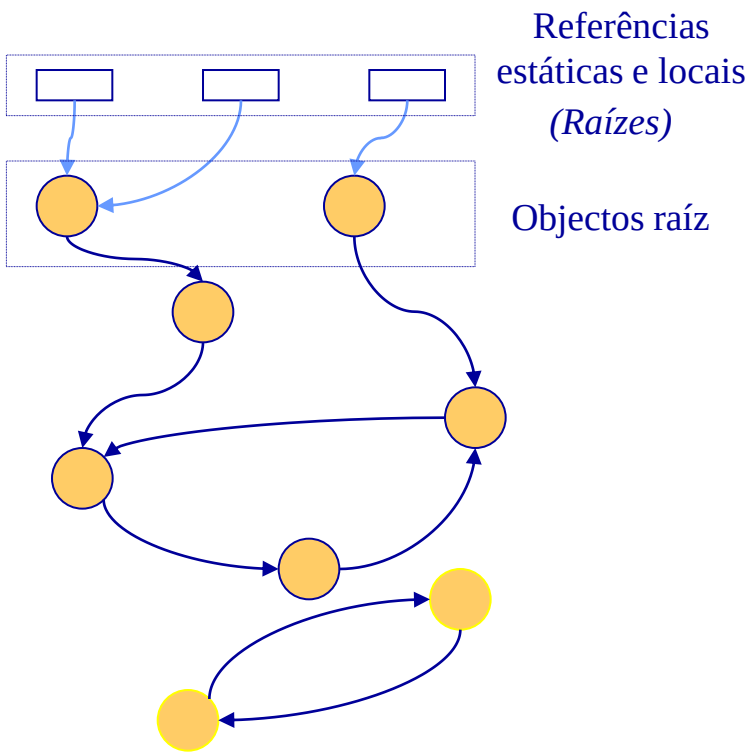


- **Vantagens**
  - Operação incremental
- **Desvantagens**
  - Custo computacional total
  - Espaço adicional por objecto
  - Não recolha de ciclos



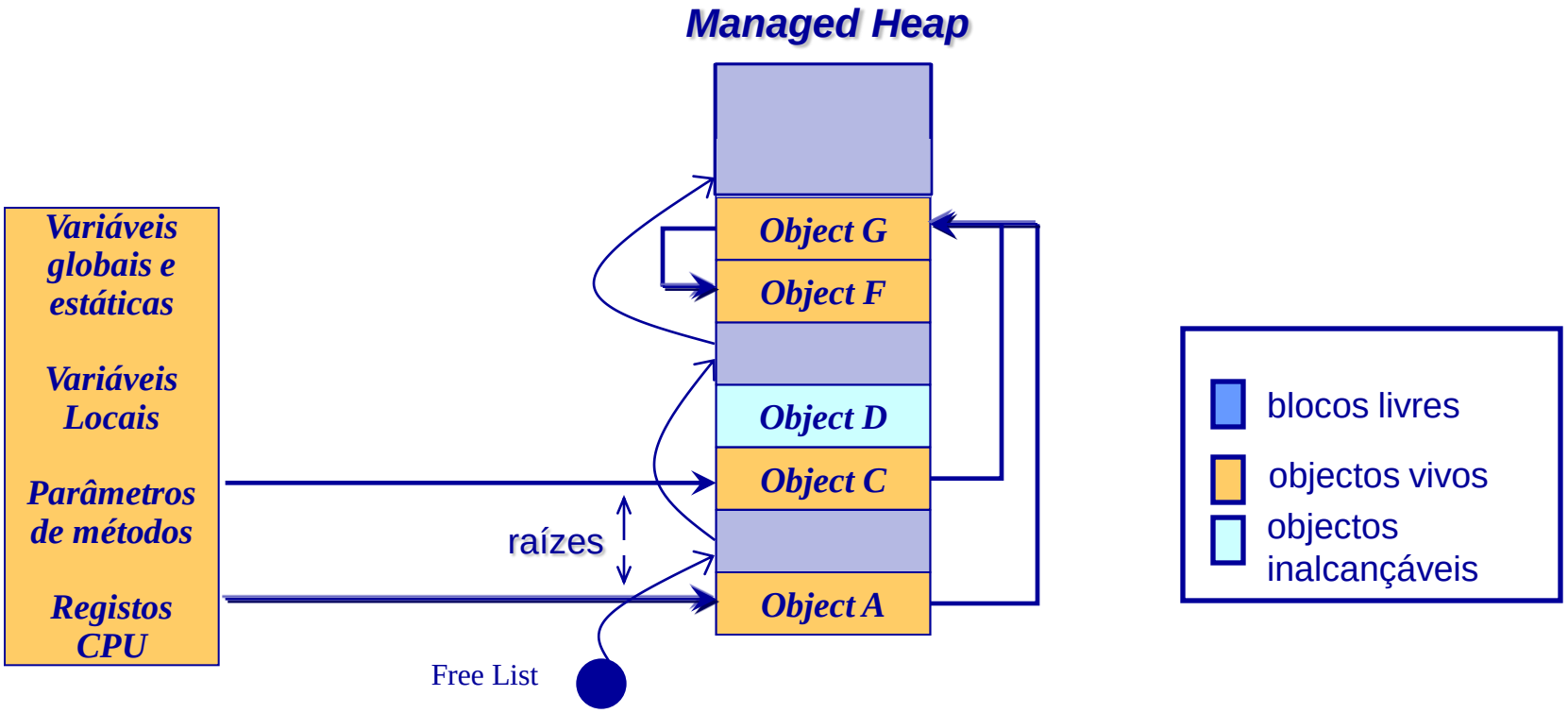
Grafo de objectos

- Grafo orientado
  - Vértices - objectos alojados explicitamente
  - Arestas - referências entre objectos
- *Objectos Raíz*
  - Conjunto de objectos alcançáveis directamente a partir das raízes
- *Adjacentes*(*A*)
  - Conjunto de objectos alcançáveis directamente a partir de *A*
- Critério: objecto *B* é *alcançável* (vivo)
  - *B* pertence ao conjunto de Objectos raíz
  - *A* é alcançável e *B* pertence a *Adjancentes*(*A*)



Detecção *(Mark)*

- Distinguir os objectos que estão a ser utilizados (vivos) dos que não estão (lixo)
- Inicialmente assume que todos os objectos são lixo
- Percorre o grafo de objectos que podem ser acedidos a partir das raízes, marcando-os como vivos (*depth-first ou breadth-first*)

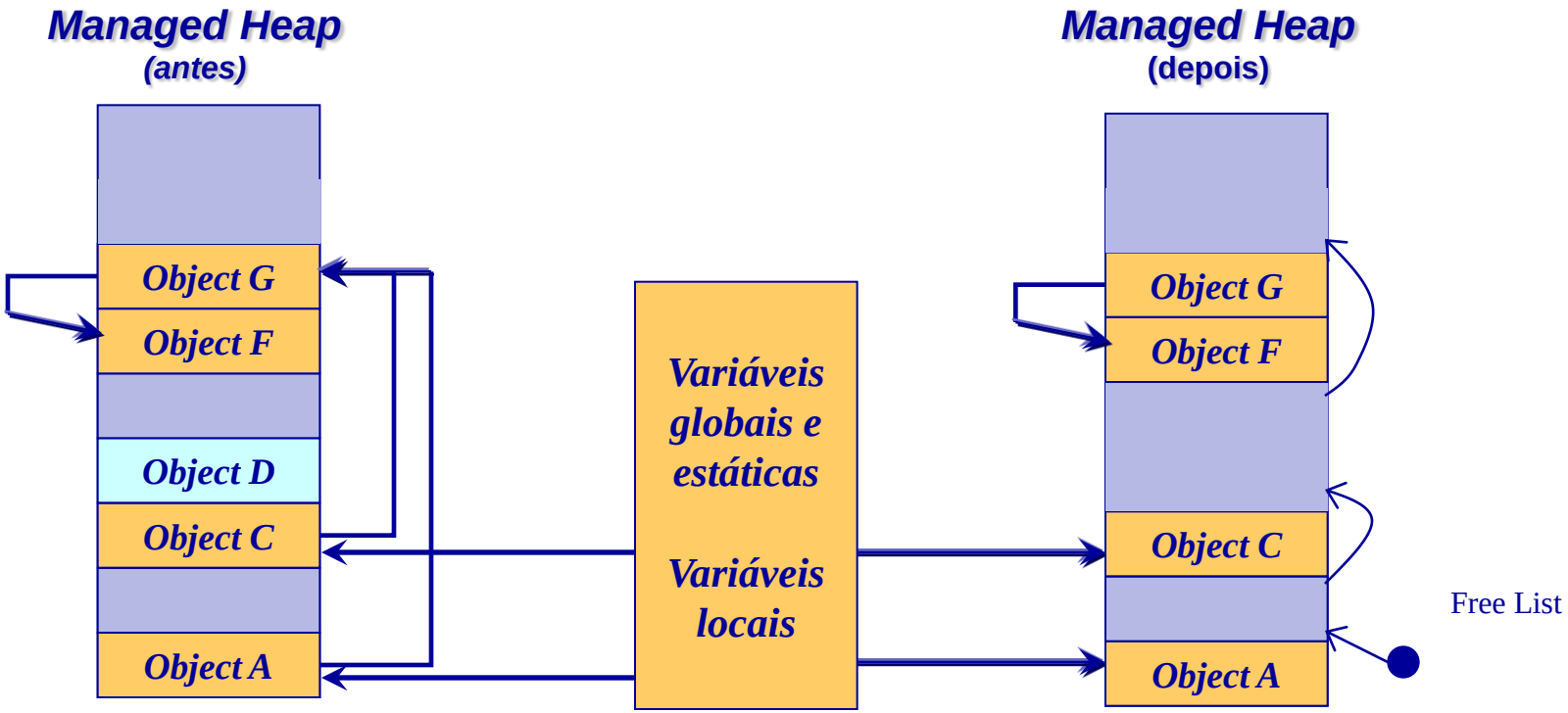


Recolha (sweep)

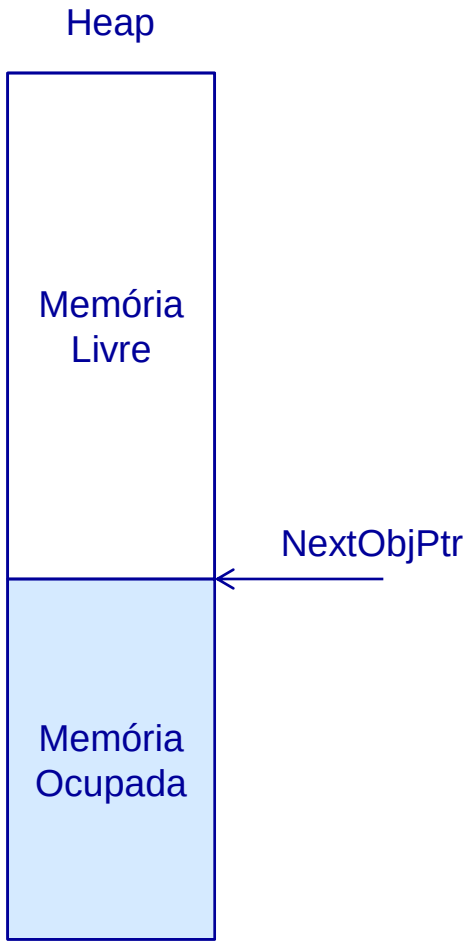
O processo de recolha (*sweep*) tem as seguintes desvantagens:

- Alocação pouco eficiente
- Fragmentação externa da memória
- Penaliza o *working set* do processo

A vantagem é não necessitar de ajustar as referências para os objectos vivos

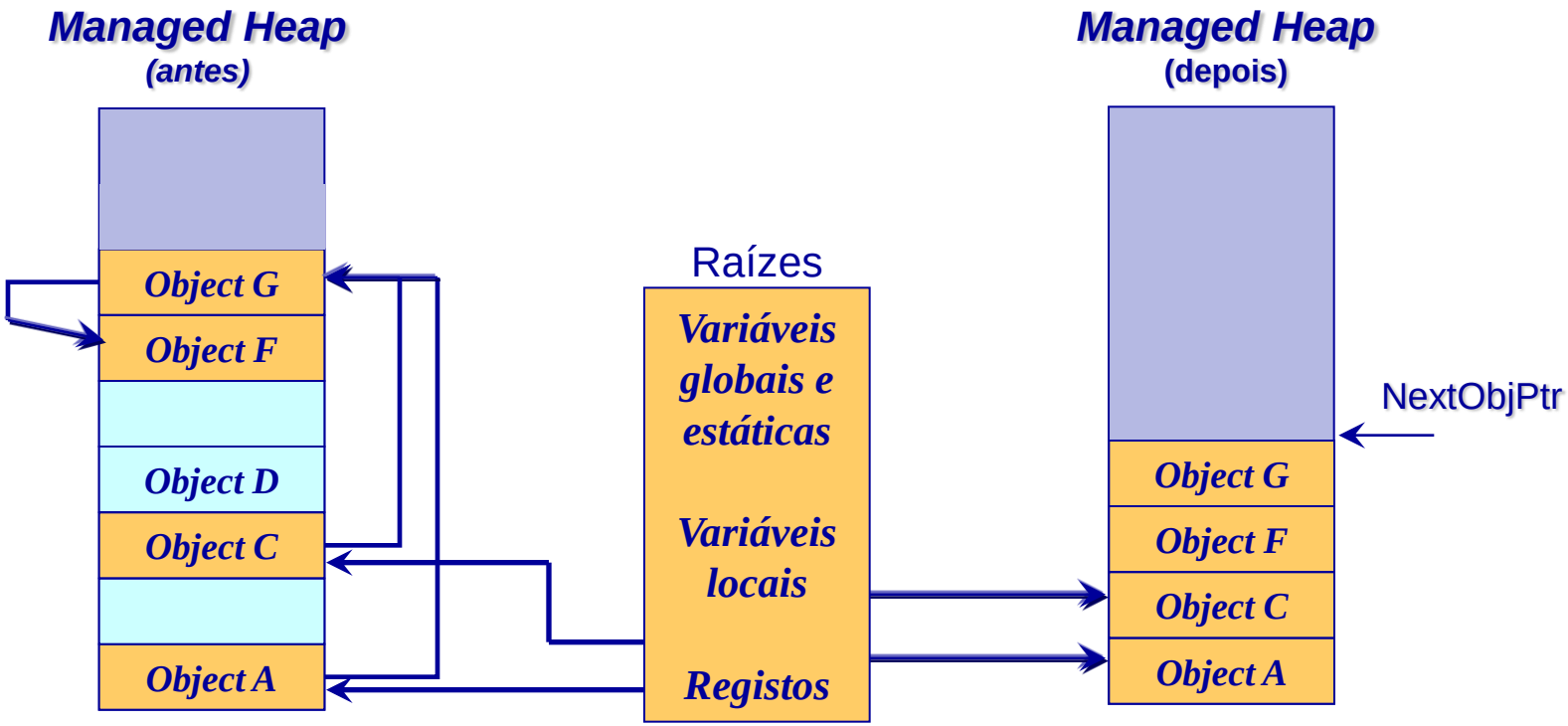


Alocação (ideal)



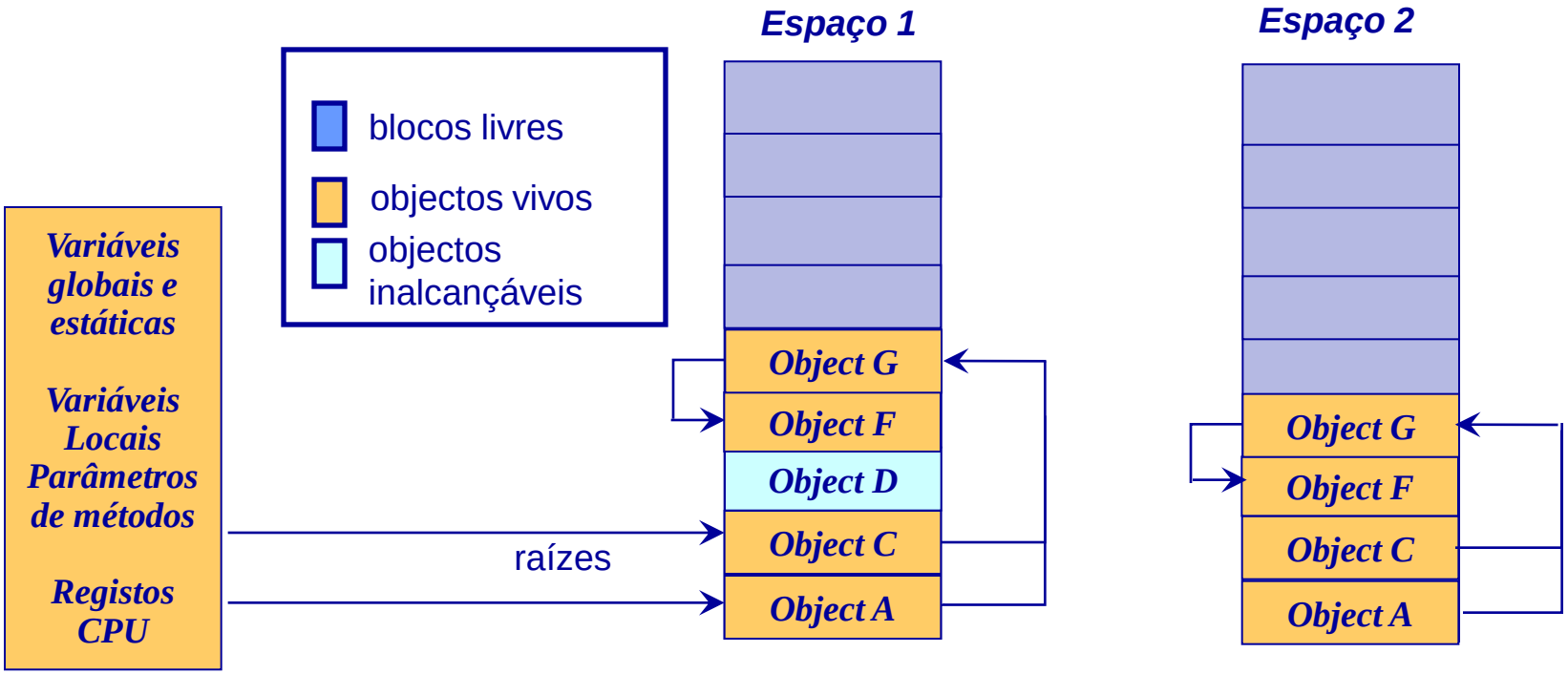
Mark and Compact - recolha com compactação

- Compactar objectos sobreviventes. O processo de compactação tem as seguintes vantagens:
  - Evita a fragmentação da memória
  - Alocações consecutivas resultam em endereços consecutivos
- A desvantagem é a necessidade de ajustar as referências para os objectos vivos



Detecção com cópia (cópia dos objectos vivos para outro espaço)

- Percorre o grafo de objectos que podem ser acedidos a partir das raízes, copiando-os para um novo espaço. No final os espaços trocam de papel
  - Evita a fragmentação da memória
  - Alocações consecutivas resultam em endereços consecutivo
  - Ajuste das referências para objectos já copiados pode ser feito na mesma passagem (memorizando no objecto o seu novo endereço)
- A desvantagem é a necessidade de mais espaço

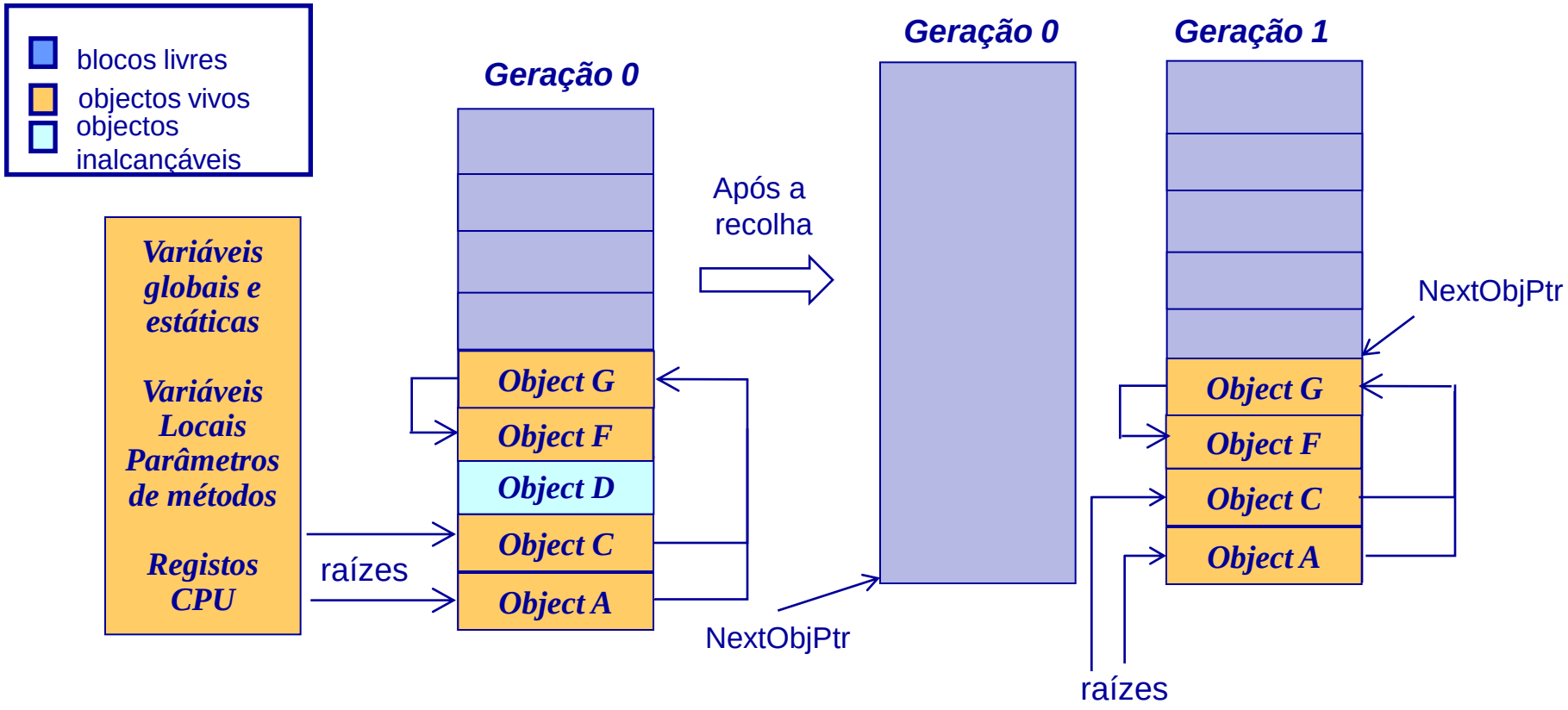


Algoritmos híbridos - GC geracional

- Algoritmo de GC que divide a memória em várias zonas, designadas gerações
  - Os objectos residem na zona de memória “adequada à sua idade”
  - A idade de um objecto é determinada função do número de GCs a que sobreviveu
- Parte do seguinte pressuposto:
  - A esperança de vida de um objecto depende do tempo que já viveu
    - a maioria dos objectos morre antes da primeira operação de recolha após a sua criação.
    - quanto mais antigo for o objecto, maior será o seu tempo de vida.
- Tem como objectivo reduzir o número de cópias e de ajustes de referências necessários

Recolha em GC com gerações

- Percorre o grafo de objectos que podem ser acedidos a partir das raízes, copiando-os para um novo espaço. No final os espaços trocam de papel
  - Evita a fragmentação da memória
  - Alocações consecutivas resultam em endereços consecutivo
  - Ajuste das referências para objectos já copiados pode ser feito na mesma passagem (memorizando no objecto o seu novo endereço)

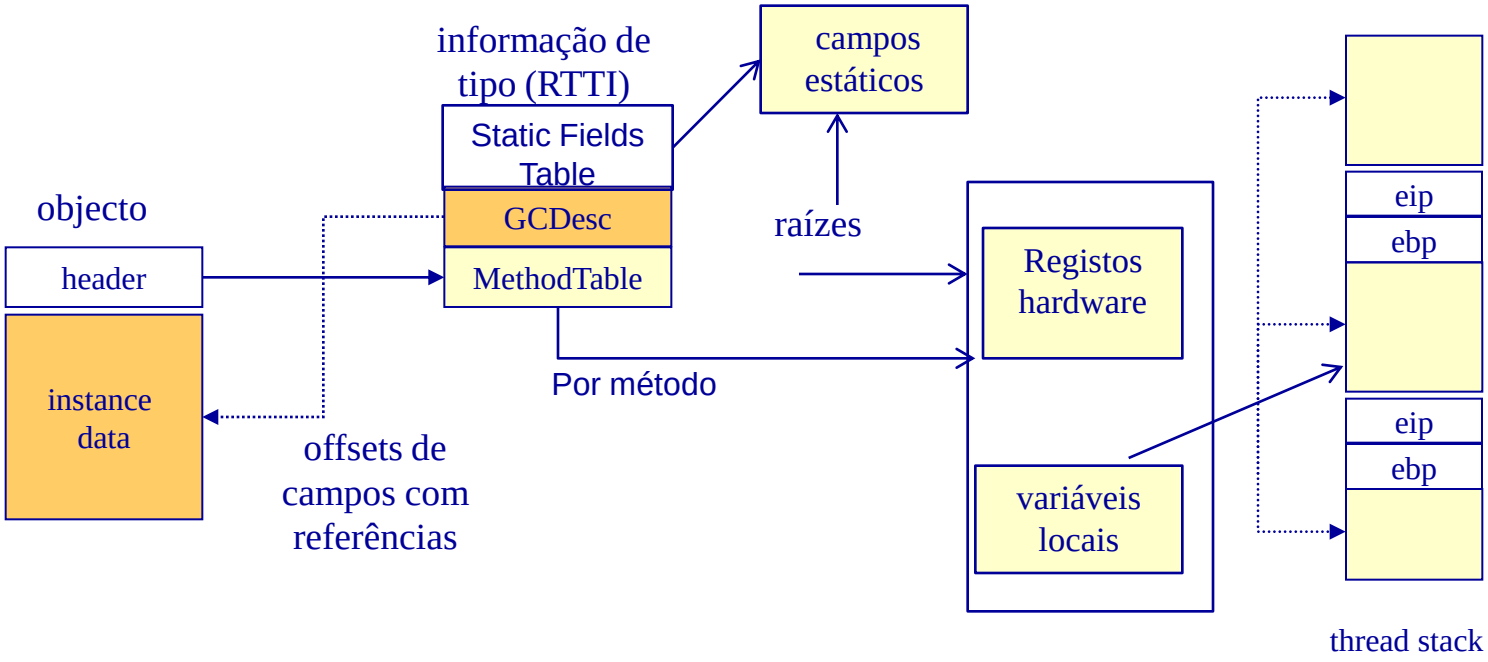


Exemplo de implementação de gerações
(Shared Source CLI)

- *Garbage Collector* baseado em duas gerações:
  - Geração 0
    - Criação de novos objectos
    - Ciclo de recolha: percurso no grafo copia os objectos sobreviventes para a geração 1
  - Geração 1
    - Objectos que sobreviveram à geração 0
    - Algoritmo de marcação e recolha (*mark-sweep*)
    - Lista de objectos livres com estratégia *first-fit*
- *Heap* dedicado para *objectos grandes* (≥ 85000 *bytes*)

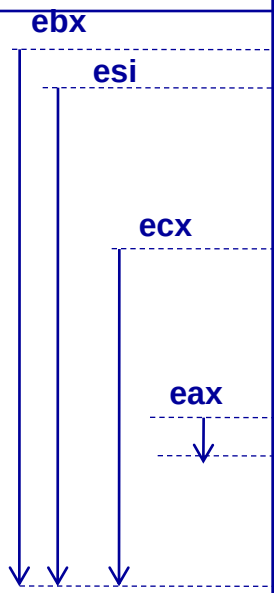
Tabelas de suporte o GC

- A máquina virtual fornece mecanismos para percorrer *stacks* e identificar referências vivas em *stack frames*. Em tempo de compilação JIT são geradas estruturas de dados de suporte a*:*
  - Informação sobre os campos dos objectos com referências
  - Enumeração de referências no registo de activação (para cada método, é criada uma tabela de regiões de endereços que especifica, para cada *região dentro do método, as referências vivas*



Regiões de código com raízes

```
internal sealed class SomeType {
  private TextWriter m_textWriter;
  public SomeType(TextWriter tw) { m_textWriter = tw;  }
  public void WriteBytes(Byte[] bytes) {
    for (Int32 x = 0; x < bytes.Length; x++) {
       m_textWriter.Write(bytes[x]);
    }
  }
}
```




```
00000000  push edi
00000001  push esi
00000002  push ebx
00000003  mov ebx,ecx                    // ebx = this (argument)
00000005  mov esi,edx                    // esi = bytes array (argument)
00000007  xor edi,edi                    // edi = x (a value type)
00000009  cmp dword ptr [esi+4],0        // compare bytes.Length with 0
0000000d   jle 0000002A                  // if bytes.Length <=0, go to 2a
0000000f  mov ecx,dword ptr [ebx+4]      // ecx = m_textWriter (field)
00000012  cmp edi,dword ptr [esi+4]      // compare x with bytes.Length
00000015  jae 0000002E                   // if x >= bytes.Length, go to 2e
00000017  movzx edx,byte ptr [esi+edi+8] // edx = bytes[x]
0000001c  mov eax,dword ptr [ecx]        // eax = m_textWriter's type object
0000001e  call dword ptr [eax+000000BCh] // Call m_textWriter's write method
00000024  inc edi                        // x++
00000025  cmp dword ptr [esi+4],edi      // compare bytes.Length with x
00000028  jg 0000000F                    // if bytes.Length > x, go to f
0000002a  pop ebx
0000002b  pop esi
0000002c  pop edi
0000002d  ret                            // return to caller
```

Referências inter-geracionais – *card table*

- Raízes para o percurso no grafo de objectos da geração 0
  - Referências:
    - Em *stack e registos*
    - Campos estáticos
    - *Outros …*
  - **Referências em objectos da geração 1**
  - **Referências no *large object heap***

- Optimizar a detecção das referências inter-gerações
  - ***Card Table*** – *bitmap* com 1 *bit* associado a 128 *bytes de memória* do heap de geração 1 e do *large object heap. Estão a 1 os bits que representam endereços de objectos que contêm referências para objectos de geração 0.*

Instrumentação de código para suportar o GC

- Em tempo de compilação JIT
  - O código é instrumentado para:
    - verificar, em ponto seguros do código (final de métodos e durante a execução de ciclos) a necessidade/pedido de colecta por parte do GC. A *thread* auto suspende-se em caso afirmativo
    - Registar, no *card table,* as escritas, em objectos de geração 1, de referências para objectos da geração 0.
- Em tempo de execução:
  - GC anuncia a necessidade de suspender a execução da aplicação
  - As threads da aplicação auto suspendem-se nos pontos seguros inseridos pelo JIT

Interface Programática (excerto da classe GC)

```
public static class GC {

      // Methods
      public static void Collect();
      public static void Collect(int generation);
      public static void KeepAlive(object obj);
      public static int CollectionCount(int generation);  // V2.0
      public static int GetGeneration(object obj);
      public static long GetTotalMemory(bool forceFullCollection);

      // ….

      // Properties
      public static int MaxGeneration { get; }
}
```

Demo

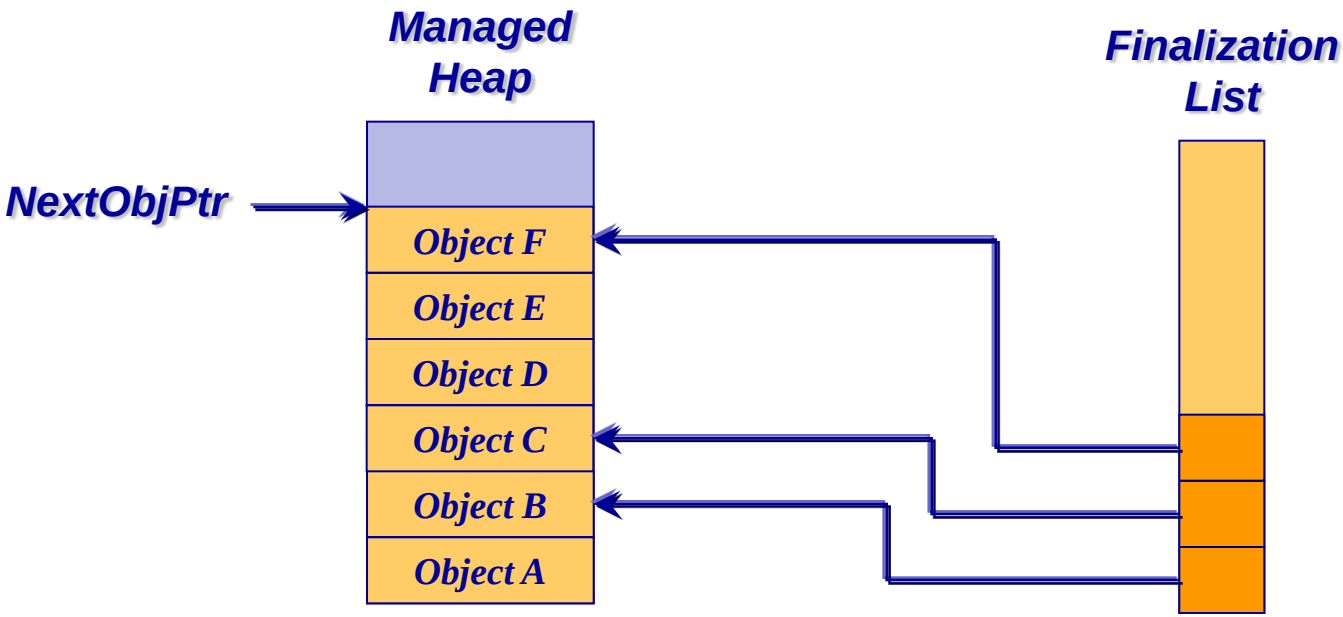
- Demo UsingMemory

Finalização

- Para tipos que necessitam de libertar recursos não geridos pelo GC
  - Redefinindo o método `Finalize` de `System.Object`
- A finalização de objectos ocorre:
  - Como consequência da execução do algoritmo de recolha, que pode acontecer por:
    - ***A geração 0 está completa***. Este evento é a forma mais comum do método *Finalize* ser invocado, porque ocorre naturalmente à medida que o código da aplicação executa, alojando novos objectos.
    - ***Chamada explicita ao método System.GC.Collect***.
  - Ou ***O CLR faz shutdown***. Quando o processo termina normalmente, tenta também fazer um *shutdown* normal ao CLR. Nestas circunstâncias, o CLR considera que deixam de existir raízes e chama o método *Finalize* para todos os objectos alojados no *managed heap* que pertençam a tipos~onde o métofo Finalize tiver sido redefinido

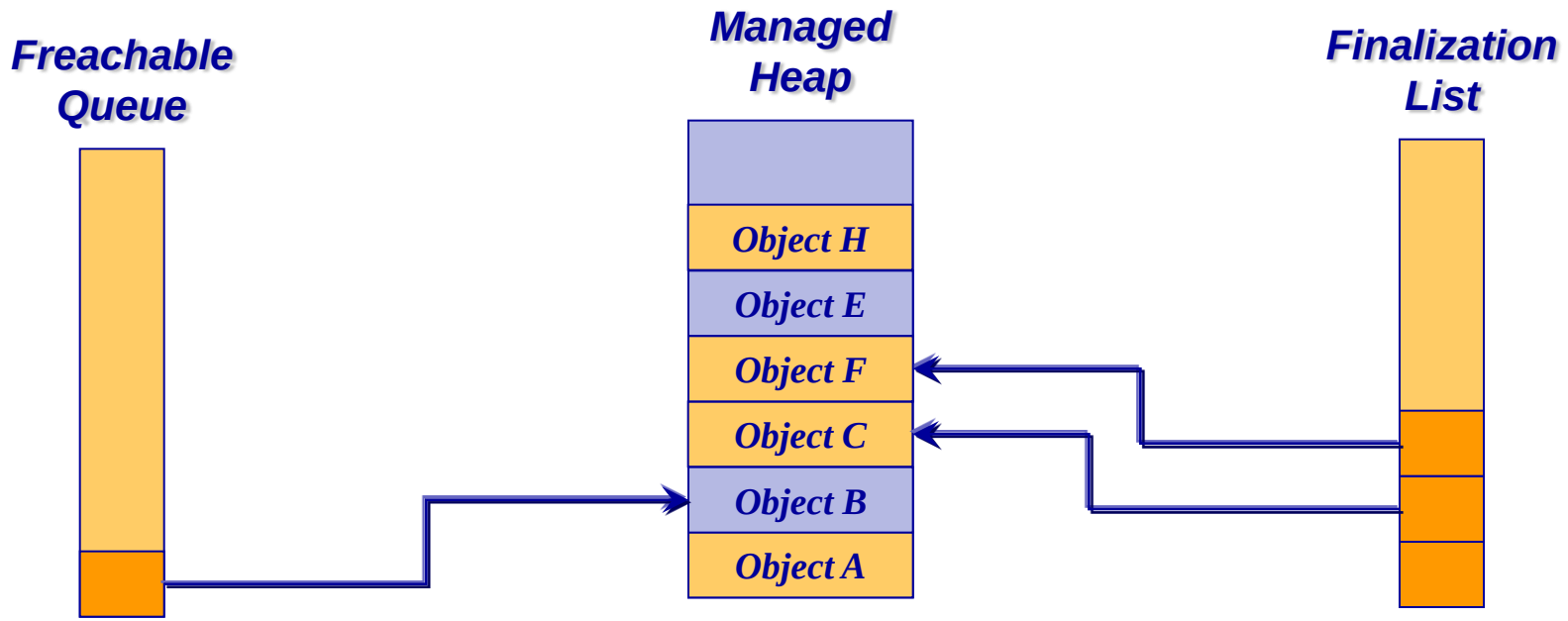
Finalização: implementação (1)

- Na fase de alojamento de memória, as instâncias de tipos que redefinam ou herdem uma redefinição do método `Finalize` são colocadas na *Finalization List*



Finalização: implementação (2)

- Na fase de reclamação de memória os objectos não utilizados, e que estão na *Finalization List*, são colocados na *FReachable Queue* (e promovidos à geração 1)
- Uma thread dedicada percorre a *FReachable Queue* invocando o *finalizer* de cada objecto aí presente (e removendo o objecto da *FReachable Queue*).
- No próximo ciclo de recolha da geração 1 a memória é libertada



Ressurreição de objectos

```
public class BaseObj {
    protected override void Finalize() {
        Application.ObjHolder = this;
        GC.ReRegisterForFinalize(this);
    }
}

class Application {
    static public Object ObjHolder;    // Defaults to null
     •••
}
```

finalizador C#

C#

```
public class SomeType
{
    …
    ~SomeType()
    {
        // Cleanup code
    }
    …
}
```

Código equivalente gerado pelo compilador

```
public class SomeType
{
    …
    protected override void Finalize()
    {
        try { /* Cleanup code */ }
        finally { base.Finalize(); }
    }
    …
}
```

Problemas da Finalização

a) **O processo de finalização prolonga o tempo de vida dos objectos finalizáveis pois, após a sua morte natural, estes (e todos os objectos referenciados por estes) têm de ser mantidos em memória até à execução do *finalizer*.**

b) **A criação dos objectos finalizáveis é mais demorada.**

c) **Não há forma de controlar a altura da execução do método *finalize*.**
   - Pode ser executado em qualquer instante desde que o objecto é considerado "lixo" até ao fim da aplicação

d) **Não há garantias que os objectos finalizáveis sejam efectivamente finalizados, pois existe um tempo limite para a execução dos finalizadores durante a terminação de uma aplicação**
   - O tempo de execução do método, durante o *shutdown*, não pode ser superior a 2 segundos
   - O somatório do tempo de execução de todos os métodos, a chamar durante o *shutdown*, não pode exceder 40 segundos
   - (valores dependentes da versão do runtime)

e) **Os finalizadores são executados por uma ordem arbitrária.**

Finalização determinística-Padrão *Disposable*

```
public class Disposable : IDisposable {
        private bool disposed=false;

          protected virtual void Dispose(bool disposing) {
            if(disposing) {
               // Acesso permitido a membros managed
            }
            // Cleanup unmanaged resources
          }

          public void Dispose() {
              if (!disposed) {
                  disposed=true;
                  GC.SuppressFinalize(this);
                  Dispose(true);
              }
          }

          ~ Disposable() {
             Dispose(false);
          }
}
```

```
public interface IDisposable {
        // Methods
        void Dispose();
}
```

C# using

C#

```
public class SomeType : IDisposable {}
…
using (SomeType s = new SomeType()) {
  // Do something
}
…
```

Código equivalente gerado pelo compilador

```
…
SomeType s = new SomeType();
try { /* Do something */ }
finally { ((IDisposable)s).Dispose(); }
…
```

Exemplo

```
Byte[] bytesToWrite = new Byte[]{1, 2, 3, 4, 5 };
// Create the file
using (FileStream fs = new FileStream("Temp.dat", FileMode.Create)){
        // Write the bytes to the file.
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);
}
```

Demos

- Demo Files

GCHandle

```
public struct GCHandle  {

    public static GCHandle Alloc(object value);
    public static GCHandle Alloc(object value, GCHandleType type);

    public void Free();

    public object Target {   get;   set; }


    public IntPtr AddrOfPinnedObject();
    public bool IsAllocated { get; }

    public static explicit operator GCHandle(IntPtr value);
    public static explicit operator IntPtr(GCHandle value);

}
```

Métodos fábrica

A classe `WeakReference` é um wrapper para `GCHandles` de tipo `GCHandleType.Weak` ou `GCHandleType.WeakTrackResurrection`.

```
public enum GCHandleType  {
    Weak,
    WeakTrackResurrection,
    Normal,
    Pinned
}
```

GCHandle *types*

- **Normal**
  - **This flag allows you to *control the lifetime of an object. Specifically, you are*** telling the garbage collector that this object must remain in memory even though there may be no variables (***roots***) in the application that refer to this object. When a garbage collection runs, the memory for this object can be compacted (moved).
- **Pinned**
  - **This flag allows you to *control the lifetime of an object. Specifically, you are*** telling the garbage collector that this object must remain in memory even though there might be no variables (roots) in the application that refer to this object. When a garbage collection runs, the memory for this object cannot be compacted (moved).
- **Weak**
  - **This flag allows you to *monitor the lifetime of an object. Specifically, you can*** detect when the garbage collector has determined this object to be unreachable from application code. Note that the object's **Finalize method may or may not have executed** yet and therefore, the object may still be in memory.
- **WeakTrackResurrection**
  - **This flag allows you to *monitor the lifetime of an object.*** Specifically, you can detect when the garbage collector has determined that this object is unreachable from application code. Note that the object's **Finalize method (if it exists)** has definitely executed, and the object's memory has been reclaimed.

Tabela de GCHandles

- O GC constrói o grafo de todos os objectos acessíveis
- O GC percorre a ***short weak reference table.*** *As entradas na tabela que são objectos inacessíveis, são marcadas a null*
- O GC executa o processo de finalização
- O GC percorre a **long weak reference** table. *As entradas na tabela que são objectos inacessíveis, são marcadas a null*

Referências Fracas

As Weak References são apenas *wrappers* para instâncias do tipo GCHandle que facilitam a sua utilização

```
public class WeakReference {
  //propriedades
  public  Boolean  IsAlive;              // read/only
  public Boolean TrackResurrection;      // read/only
  public Object Target;                  // read/write
  // construtor
  public WeakReference(object target, bool weakLong);
}
```

```
void SomeMethod() {
   // Create a strong reference to a new object.
   Object o = new Object();
   // Create a strong reference to a short WeakReference object.
   // The WeakReference object tracks the Object's lifetime.
   WeakReference wr = new WeakReference(o);
   o = null; // Remove the strong reference to the object.
   o = wr.Target;
   if (o == null) {
     // A garbage collection occurred and Object's was reclaimed
   } else {
      // A garbage collection did not occur and I can successfully access
      // the object using o.
    }
}
```

Cache com Weak References

```
public class FancyObjects {
  private string name;
  private FancyObjects(string name) { this.name = name; }

  // private static cache of well-known objects
  static IDictionary table = new Hashtable();

  // public accessor function
  public static FancyObject Get(string name) {

  // check cache
      FancyObject result =
     (FancyObject)table[name];
   // create and cache if not there already
      if (result == null) {
        result = new FancyObject(name);
        table[name] = result;
      }
      return result;
    }
}
```

Com Weak References

```
public static FancyObject Get(string name) {

// check cache for weak reference
      WeakReference weak =
(WeakReference)table[name];
// try to dereference weak ref
      FancyObject result = null;
      if (weak != null)
        result = (FancyObject)weak.Target;

// create and cache if not there already or has been
GCed
      if (result == null ) {
        result = new FancyObject(name);
// cache weak reference only!
        table[name] = new WeakReference(result);
      }
      return result;
    }
}
```