**Gerenciamento automático de memória**

Alocação explícita ou implítica

Desalocação automática

No Java é o coletor de lixo que trata a memória que está sendo utilizada.

No caso do C++ é misto.

**Controle de informações**

O *heap* é onde ocorre as alocações dinâmicas e instanciações de objetos. Vamos olhar o *heap* como um conjunto de células. Coleção de blocos pré-alocados.Em princípio, todo o *heap* será composto de vários blocos de mesmo tamanho de uma maneira geral.

As células serão diferenciadas de duas maneiras:

**Atômicas**: não contém ponteiros.

**Não Atômicas**: contém ponteiros para outras células. (Ex: struct nó)

Indicador de células:

*Indicador de atomicidade*

*Contador de referências*

*Ponteiros de ligação*

*Indicador de marcação*

*Tamanho de dados*

*etc...*

*Obs: todas as células vão conter todas essas informações para controle da cada célula, esse controle é feito pelo gerenciador e não pelo programador.*

Gerenciamento automático:

**Contador de referencia:** manter o controle de quantos elementos apontam para uma determinada célula.

Campo para registrar quantas referências que existem para aquela célula específica. Todo controle que houver vai gerenciar as referências. Se não houver ninguém referenciando a uma célula, significa que ela poderá ser liberada.

Raízes são os ponteiros chamados de referências externas, as variáveis de um programa devem ser controladas para saber para quem elas apontam, a linguagem deve ter como recurso para que isso funcione.

*Abordagem*, cada vez que é feito uma manipulação deve-se gerenciar as referências e atualizar o contador que fala quantas referências hã para uma determinada célula.

Se a célula for zerada, pode haver o desencadear de desalocação de uma lista com milhares de elementos, onde o próximo elemento acaba sendo zerado e desalocado gerando um efeito dominó de desalocação. O problema é que isso é muito ineficiente.

Há uma intenção de otimização onde quando uma célula é apontada por *zero* a célula não é desalocado imediatamente.

**Reumo:** Controle automática de memória :)

**Coletor de lixo:** manter um controle sobre quem é referenciado e quem não é. Se é possível chegar em um nó através de referência externa, o nó é mantido. Caso contrário, o mesmo será liberado.

A coleta de lixo só é realizada quando a memória passa a estar escassa. O coletor de lixo para o programa, limpa a memória e o programa volta para execução.

***Prós****:*

Evitam-se ponteiros inválidos! Não haverá ponteiros apontando para áreas inválidas. Perde-se a variável externa antes de liberar a memória.

Evita-se a liberação "dupla". Exemplo:

p1 = malloc(v);

p2 = p1;

*{ bloco de código }*

free(p1);

*{ bloco de código }*

*p3 = malloc(v2);*

free(p2);

Evitam-se os vazamentos de memória (leaks): esquecimento de liberar um bloco, o qual fica sem uso e sem possibilidade de realocação.

***Contras:***

Fazer o garbage collection consome processamento!

Sobrecarga de processamento;

Redução do desempenho;

Desempenho irregular.

Imprevisibilidade do momento da coleta:

Redução de desempenho ou parada no processamento;

Frequência desconhecida de execução da coleta;

Implicação em aplicações de tempo real.

Vulnerabilidade a código mal escrito (não é exclusivo de garbage collection! :P)

Instanciações desnecessárias ou em locais inadequados.

for(i=0; i < 5000; i++){

o = new Obj();

*{ bloco de código }*

}

*Em algumas situações pode ser interessante instanciar o objeto fora do laço, pois neste caso haverá uma quantidade "absurda" de objetos instanciados, o que pode gerar lentidão e muito mais trabalho para o garbage collection liberar todos os objetos instanciados no laço.*

**Marcação e troca (Algoritmo)**

"Passar a vassoura e limpar o que não está sendo utilizado"

Vai marcando todos os nós, se são atômicos ou não. Se houver duas referências para o mesmo nó não há a necessidade de verificar novamente. Marca todos que são acessíveis. Assim que todos forem marcados, quem não foi marcado é porque não há acesso externo, ou seja, podem ser removidos.

Fases:

Marcação (marking) - determinação de acessibilidade direta ou indireta

Regeneração (collection) - retorno das células não marcadas ao *heap*; compactação de memória.

**Marcar e trocar (*mark and sweep*)**

Algoritmo básico:

*- Para cada ponteiro de raiz*

*Visitar a célula indicada*

*- Para cada célula visitada*

*Marcar a célula*

*Para cada ligação, visitar a célula indicada*

Regeneração:

*- Busca sequencial pelas células não marcadas*

*- Criação de lista encadeada de células não marcadas (lista de disponíveis)*

*- Compactação do espaço de memória*

*Varredura de memória com deslocamento das células deixar a parte livre contínua*

*Envolve os ajustes dos ponteiros de referência*

*Obs:* A busca é realizada uma a uma, verificando quais células estão marcadas ou não, se estiver marcada, nada acontece, se não, ela é liberada.

<http://www.artima.com/insidejvm/applets/HeapOfFish.html>

Métodos incrementais:

Realiza o garbage collection aos poucos.

Considerações finais:

**Gerenciamento Automático**

- Contador de referências

Controle do número de referências a cada célula

- Coleta de lixo

Marcação e regeneração

Cópia

Incrementais

*Referências:*

Drozdek (2002). Capítulo 12: **Gerenciamento de memória**. Seções 12.1 e 12.3.

Langsam et al (1996), Capítulo 9: **Storage management**. Seção 9.2.