Universidade Federal do Rio Grande do Norte

Instituto Metrópole Digital

Linguagem de Programação I • IMD0030

◁ 2º Projeto de Programação ▷

GREMLINS - GeRENciador de Memória com LIsta eNcadeada Simples

11 de maio de 2016

Gremlin Stripe

Sumário

Introdução

O objetivo deste projeto de programação é motivar o uso de listas encadeadas em um contexto de aplicação real. A aplicação selecionada é um **gerenciador de memória** (GM). Implementar um GM não é uma tarefa trivial porém é importante em situações na qual é necessário ter total controle sobre a organização de memória e melhorar o desempenho de tempo associado à requisições de memória pela aplicação-alvo.

Um exemplo típico em que um GM pode fazer a diferença em termos de desempenho acontece em *motores de jogos*, usados para construir jogos digitais. Em geral jogos digitais são aplicações de alto desempenho em termos de tempo de execução, e uma das possíveis otimizações viáveis para esta categoria de aplicação é tomar controle sobre como a memória é gerenciada.

De maneira bem simplificada, um GM vai requisitar ao sistema operacional (SO) um grande bloco de memória e, a partir de então, vai se encarregar de entregar blocos menores de memória à aplicação, quando solicitado. Isso implica que as requisições de memória dinâmica pela aplicação não serão mais realizadas via `new` e `delete` (i.e. resolvidas pelo sistema operacional), mas sim através de novos métodos associados à classe GM. Em particular, este projeto estipula a implementação do sistema *memory pool* para gerenciamento de memória.

Para uma visão geral sobre o gerenciamento de memória, estratégias e algoritmos, visite o verbete na wikipédia `https://en.wikipedia.org/wiki/Memory_management` como ponto de partida.

1 Conceitos Básicos sobre Gerenciamento de Memória

Um *memory pool* é uma região de memória contígua organizada em blocos de memória não sobrepostos que são alocados para o usuário do sistema. Confira na Figura 1 uma possível representação gráfica para um *memory pool* com 128 Kbytes de memória com alguns blocos ocupados.

Uma *memory pool* suporta dois tipos de operações fundamentais:

- **Allocate**: A operação de *aquisição* localiza na *pool* uma região de memória contígua do tamanho solicitado e retorna um ponteiro para o início deste bloco, se houver memória livre. A região então é marcada como *reservada*, o que significa que ela está em uso pela aplicação-cliente. No caso de não ser possível encontrar um bloco de memória livre do tamanho solicitado, o sistema deve gerar uma exceção.

- **Free**: A operação de *liberação* retorna a região de memória reservada de volta para o *pool*. Depois de liberada, a mesma região (ou parte dela) pode ser reservada novamente via chamada subsequente da operação *aquisição*. Consiste um erro tentar liberar uma região de memória que não está reservada. O comportamento do gerenciador neste caso é indefinido.

2 A Classe `StoragePool`

Nesta seção introduzimos a classe `StoragePool` que deve definir os métodos básicos de manipulação de uma *memory pool*, apresentados na seção anterior. Eventualmente esta classe será estendida de

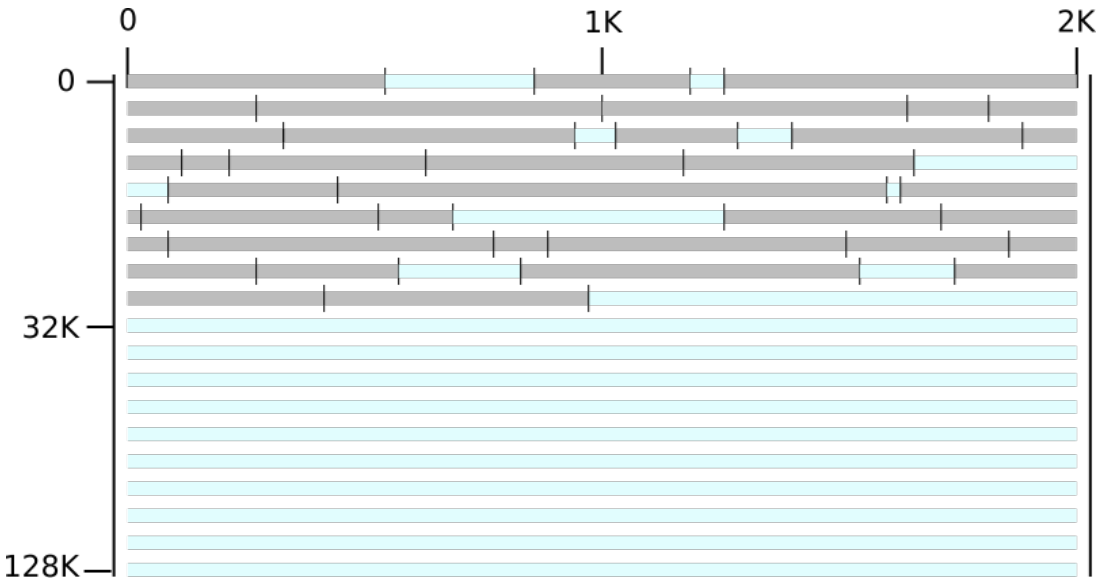


Figura 1: Exemplo de ocupação de 128K de memória via *memory pool*. As barras em cinza são blocos ocupados, enquanto que as barras claras são blocos desocupados e prontos pra uso. Os traços verticais representam os limites das alocações dentro de uma área ocupada. Note que alguns blocos ocupados são contíguos, outros não.

acordo com o tipo de implementação adotada, como por exemplo o uso de listas simplesmente encadeadas ou duplamente encadeadas para gerenciar a área livre de memória.

2.1 Visão Geral da Classe

O trecho de Código 1 apresenta a definição da classe abstrata `StoragePool` com as operações fundamentais discutidas anteriormente.

Código 1 Definição da classe abstrata `StoragePool`.

```
1 class StoragePool {
2     public:
3         virtual ~StoragePool();
4         virtual void* Allocate( size_t ) = 0;
5         virtual void Free( void * ) = 0;
6 };
```

A função `Allocate` requisita uma quantidade de memória expressa em bytes e passada como argumento. O retorno é um ponteiro para a região reservada.

A função `Free` recebe um ponteiro para uma região reservada e a libera para uso posterior pelo GM, no caso o *memory pool*.

A classe abstrata `StoragePool` define um conjunto mínimo de operações que outras classes *memory pool* devem implementar. Cada nova classe derivada deve adotar uma política distinta de organização e ocupação de memória, discutida ao longo deste documento.

2.2 Os Métodos `new` e `delete`

A forma de utilização de uma classe derivada de `StoragePool` seria algo como:

```
SomePool p;                      // Instanciando um GM derivado de StoragePool.
void *ptr = p.Allocate( 100 );   // Alocar 100 bytes de memória.
...                              // Usar a memória.
p.Free( ptr );                   // Liberar a memória.
```

O código acima requisita um bloco de 100 bytes de memória, para posteriormente o liberar.

Em um típico programa C++ as alocações e liberações de memória são realizadas via `new` e `delete`, respectivamente. Um exemplo genérico seria

```
T * ptr = new T;
```

onde `T` é o nome de um tipo. Neste caso o `new` aloca uma instância de `T` e retorna um ponteiro para a instância. A chamada da função `new` envolve três passos:

1. Uma quantidade de memória suficiente para armazenar uma instância de `T` (`sizeof(T)` bytes) é solicitada;
2. O construtor de `T` é chamado (se não for um tipo básico da linguagem), e;
3. Um ponteiro para o objeto é retornado.

Os passos 1 e 3 logo acima são realizados pela função em C++

```
void * operator new ( size_t );
```

Similarmente, o comando `delete ptr` libera o objeto apontado por `ptr` e retorna sua memória para o GM. Esta chamada de função envolve dois passos:

1. O destrutor do objeto apontado por `ptr` é chamado, e;
2. A memória ocupado pelo objeto é liberada.

Os passos 1 e 2 logo acima são realizados pela função em C++

```
void operator delete ( void * );
```

Se um novo GM quiser assumir o controle, é necessário sobrescrever estas duas funções do C++, da seguinte forma:

```
1 SomePool p;
2
3 void * operator new( size_t bytes )
4     { return p.Allocate( bytes ); }
5
6 void operator delete( void * ptr )
7     { p.Release( ptr ); }
```

No exemplo acima o *memory pool* é uma variável global `p` e todo a requisição regular de memória será gerenciada por este objeto.

O comando `new` ainda oferece um mecanismo para que o programador controle qual o GM usado através de argumentos adicionais

```
T * ptr  = new (argument list) T;
```

Os argumentos são, então, passados para a função sobrecarregada correspondente de `new`. Por exemplo, uma possível implementação para `new` que permite indicar um GM específico seria

```
void * operator new (size_t bytes, StoragePool & p )
    { return p.Allocate( bytes ); }
```

o que viabiliza utilizar o código abaixo

```
SomePool p, q;
T * ptr1 = new (p) T; // Alocando T dentro do GM 'p'.
T * ptr2 = new (q) T; // Alocando T dentro do GM 'q'.
```

no qual existem dois GMs distintos, `p` e `q`, e cada um armazena uma instância diferente de `T`.

Infelizmente, contudo, não é possível fazer o mesmo tipo de passagem de parâmetro para a função `delete`. Isso implica que a única forma de liberar a memória no exemplo anterior seria

```
delete ptr1;
delete ptr2;
```

Note que isso provoca um problema: se (sobre)escrevêssemos uma nova versão para `delete` não teríamos como saber sobre qual objeto `p` ou `q` deveríamos invocar o método `Free`!

Uma forma de contornar esta limitação é manter um registro explícito sobre qual GM é o "dono" de um bloco de memória. Isso pode ser feito adicionando, para cada região de memória solicitada, um *tag* (marca) que contém um ponteiro para o *pool* (GM) ao qual o bloco pertence. Veja no Código 2 um exemplo de como adicionar o *tag* ao uma região de memória.

Note que uma "marca" (`struct Tag`) é adicionada a todo espaço de memória solicitada a uma GM, e esta marca contém um ponteiro para a GM que é proprietária da memória "emprestada". Perceba também que os espaços de memória solicitados diretamente ao SO possuem uma marca com ponteiro nulo. Desta forma, agora é possível durante a operação `delete` identificar a quem pertence o espaço de memória que se deseja liberar (confira o condicional da Linha 22 do Código 2).

Por consequência da adição do `Tag`, sempre que o programa solicitar n bytes a uma GM na verdade serão alocados n bytes + `sizeof(Tag)` bytes, sendo que o `Tag` ocupa os primeiros bytes. É por este motivo que o `new` sobrescrito deve retornar um ponteiro para o início do bloco de dados, "saltando" o `Tag`.

O código abaixo ilustra como solicitar e liberar memória a partir de várias GMs do tipo `SomePool` (exemplo didático apenas) que estende `StoragePool`.

```
1 SomePool p, q;
2 T *ptr0 = new T;      // Instanciando 'T' diretamente com o Sistema Operacional.
3 T *ptr1 = new (p) T; // Instanciando 'T' no GM 'p'.
4 T *ptr2 = new (q) T; // Instanciando 'T' no GM 'q'.
5 ...
6 delete ptr0;          // Because the tag is nullptr, we know 'ptr1' is from the SO.
7 delete ptr1;          // Because of the tag, we know 'ptr1' belongs to 'p'.
8 delete ptr2;          // Because of the tag, we know 'ptr0' belongs to 'q'.
```

Código 2 Sobrecarregando `new` e `delete` e adicionando um *tag* em cada bloco.

```
1  struct Tag {  StoragePool * pool; };
2
3  void * operator new ( size_t bytes, StoragePool & p ) {
4      Tag* const tag = reinterpret_cast<Tag *> ( p.Allocate(bytes + sizeof(Tag)) );
5      tag->pool = &p;
6      // skip sizeof tag to get the raw data-block.
7      return ( reinterpret_cast<void *>( tag + 1U ) );
8  }
9
10 void * operator new ( size_t bytes ) {  // Regular new
11     Tag* const tag = reinterpret_cast<Tag *>( std::malloc(bytes + sizeof(Tag)) );
12     tag->pool = nullptr;
13     // skip sizeof tag to get the raw data-block.
14     return ( reinterpret_cast<void *>( tag + 1U ) );
15 }
16
17 void operator delete ( void * arg ) noexcept {
18     // We need to subtract 1U (in fact, pointer arithmetics) because arg
19     // points to the raw data (second block of information).
20     // The pool id (tag) is located 'sizeof(Tag)' bytes before.
21     Tag * const tag = reinterpret_cast<Tag *>( arg ) - 1U;
22     if ( nullptr != tag->pool ) // Memory block belongs to a particular GM.
23         tag->pool->Release( tag );
24     else
25         std::free( tag ); // Memory block belongs to the operational system.
26 }
```

Em termos de organização de implementação, sugere-se que a sobrecarga dos métodos `new` e `delete` sejam agrupados em um arquivo cabeçalho `mempool_common.h` o qual deve ser incluído por qualquer código cliente que queira usar o GM proposto. Além deste arquivo cabeçalho, será necessário incluir o cabeçalho de definição da classe que de fato implementa o GM, apresentado nas próximas seções.

3 Gerenciamento de Blocos Livres com Lista Encadeada Simples

Existem várias formas de implementar um GM. Em todos os casos o objetivo é sempre o de acelerar o tempo de execução dos métodos básicos `Allocate` e `Free`, sendo que o ideal seria se estas operações apresentassem complexidade temporal constante.

Neste projeto de programação você deve implementar uma GM da categoria *storage pool* que usa uma lista simplesmente encadeada para manter uma lista de **blocos** de memória livres. Ou seja, você vai criar uma classe `SLPool` derivada de `StoragePool`, introduzida no Código 1.

A seguir apresentaremos algumas definições para ajudar e guiar a implementação do GM em questão.

3.1 Organização Interna do GM `SLPool`

Quando um GM é instanciado no código cliente, é necessário informar seu tamanho em número de bytes. O GM então deve solicitar ao SO a quantidade de memória requisitada, a qual é organizada como um arranjo `Pool` de **blocos** (`struct Block`). Um bloco é a unidade básica que pode ser concedida ao cliente, portanto pode ser classificado como **reservado**, i.e. "emprestado" ao cliente, ou **livre**, i.e. sob o controle do GM. Uma sequência de blocos consecutivos e contíguos no arranjo `Pool` é denominado de **área**.

Em linhas gerais, um GM deve realizar as seguintes tarefas: manter informações sobre quais os blocos de memória que foram alocados e manter uma lista de áreas livres ou não-alocadas. Vamos considerar uma chamada chamada de `Allocate` pelo cliente; o GM deve localizar no arranjo `Pool` uma área de blocos livres cujo tamanho, expresso em quantidade de blocos, seja o suficiente para satisfazer a solicitação em bytes recebida. Conforme indicado no início desta seção, a abordagem ora sugerida é a de manter uma *lista simplesmente encadeada* das áreas livres com seus respectivos tamanhos.

Além de manter uma lista de áreas livres, é necessário manter informações acerca do tamanho de área de blocos reservada ou alocada ao cliente. Precisamos desta informação visto que o comando `Free` recebe apenas um ponteiro para a área a ser liberada; i.e. o tamanho da área não é informado como argumento para a operação `Free`! Portanto esta informação deve, de algum modo, estar codificada na própria área de blocos cedida ao cliente.

Como fazer para manter todas estas metainformações? A resposta está em utilizar o próprio espaço do bloco para guardar metainformações, da seguinte forma: (1) o tamanho (comprimento em número de blocos) de cada área é armazenado no *primeiro campo* do primeiro bloco de cada área, e; (2) os ponteiros responsáveis pelas ligações da lista encadeada de áreas livres serão definidos como o *segundo campo* dentro do primeiro bloco de cada área livre.

Para entender melhor como os blocos são utilizados para armazenar metainformações, precisamos entender a estrutura básica de um bloco. A Figura 2 apresenta uma representação abstrata de um bloco, com destaque para seus campos internos. Já o Código 3 apresenta o esqueleto básico da classe `SLPool`, com destaque para a organização do bloco em dois campos: `Header` e um segundo campo definido por meio de `union` para podermos interpretá-lo tanto como um simples ponteiro para outro bloco, `mp_Next`, quanto como o espaço de dados cedido ao cliente, `mc_RawArea`.

Em geral o bloco é definido como tendo entre 8 e 16 bytes, mas este é um número arbitrário com um custo-benefício associado. Se o bloco for definido com um tamanho S muito grande, podemos ter desperdício de memória, caso o montante solicitado pelo cliente não seja um múltiplo exato de S. Por outro lado, se S for muito pequeno, é possível que o GM fique com a memória muito *fragmentada* após múltiplas operação de `Allocate` e `Free`.

3.2 Exemplo de um GM em Ação

Para contextualizar os conceitos apresentado até aqui, apresentamos um exemplo didático no qual o código cliente solicita 220 bytes ao GM, sendo que cada bloco do GM tem 16 bytes. Vamos

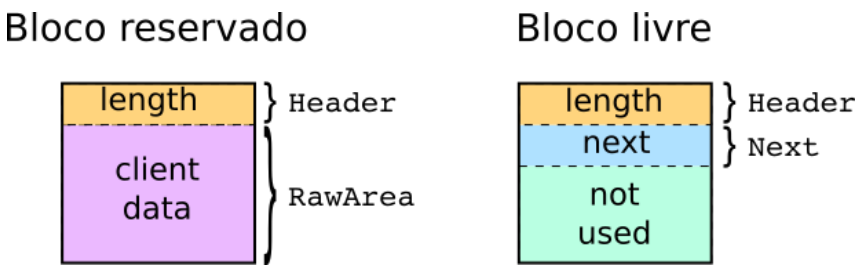


Figura 2: Estrutura geral de um **bloco** (`struct Block`), que é a unidade básica do nosso GM. Um bloco possui dois campos, `header` e um segundo campo que pode ser interpretado como um ponteiro para outro bloco ou o espaço de memória cedido ao cliente. As figuras ilustram a interpretação da organização interna do *primeiro* bloco de uma área reservada (esquerda) e de uma área livre (direita).

Código 3 Esqueleto da classe `SLPool` no qual é possível identificar os campos de metadados `Header` e a organização do bloco que pode ser interpretado como um ponteiro da lista, `mp_Next`, ou uma área crua de dados para o cliente, `mc_RawArea`.

```
1  class SLPool : public StoragePool {
2      public:
3          struct Header {
4              unsigned int mui_Length;
5              Header() : mui_Length(0u) { /* Empty */ };
6          };
7
8         struct Block: public Header {
9             enum { BlockSize = 16 }; // Each block has 16 bytes.
10            union {
11                Block *mp_Next;
12                char mc_RawArea[ BlockSize - sizeof( Header ) ];
13            };
14            Block() : Header(), mp_Next( nullptr ) { /* Empty */ };
15        };
16
17     private:
18         unsigned int mui_NumberOfBlocks;
19         Block *mp_Pool;         //!< Head of list.
20         Block &mr_Sentinel;     //!< End of the list.
21
22     public:
23         explicit SLPool( size_t );
24         ~SLPool();
25         void * Allocate( size_t );
26         void Free( void * );
27 };
```

acompanhar o mapa de memória do GM ao longo das três operações fundamentais: criação, alocação e liberação.

3.2.1 Criação do GM

Inicialmente, a organização interna do GM corresponde a Figura 3, na qual definimos um `Pool` com 15 blocos: 14 blocos (14×16 bytes $= 224$ bytes de capacidade) $+ 1$ bloco *extra* para servir como **Sentinela**[1] da lista encadeada. Inicialmente existe apenas 1 área livre (i.e. 1 nó na lista) que compreende todos os 14 blocos de `Pool`; por isso o meta-campo `next` do primeiro bloco aponta para `NULL`. Note também que o último bloco é designado para ser o Sentinela da lista e não pode ser liberado para o cliente, pois serve apenas para apontar para o primeiro nó da lista de áreas livres—encare o Sentinela como o nó cabeça da lista de áreas livres, cujo objetivo é facilitar as operações de manipulação da lista.

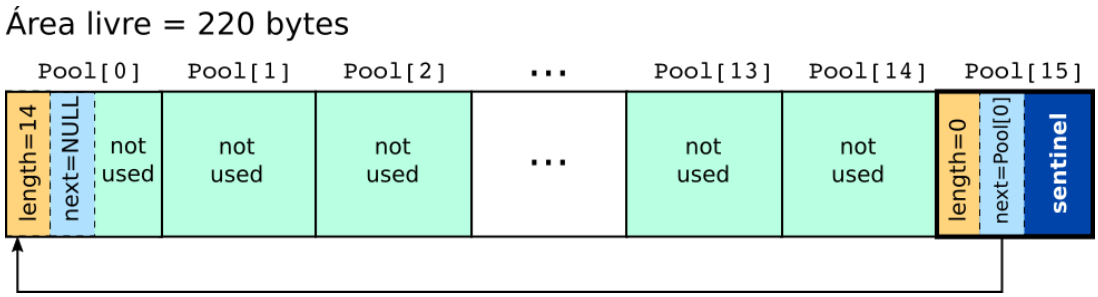


Figura 3: Representação inicial do GM, com `Pool` de 15 blocos, todos inicialmente livres. Note que o primeiro bloco da área livre contém a metainformação `length` $=14$, indicando que temos 14 blocos livres de forma contígua, e `next=NULL` indicando que não existe outro nó na lista de áreas livres. Nota também que o último (extra) bloco é marcado como sendo o *Sentinela* da lista encadeada; este nó não é pode ser liberado para o cliente e funciona como o nó-cabeça da lista de áreas livres, que no exemplo possui apenas 1 nó.

3.2.2 Alocação de Memória

Quando o cliente solicita uma quantidade de memória (em bytes), o GM deve calcular qual o número mínimo de blocos necessário para satisfazer a solicitação:

$$Nblocks = \lceil (bytes + sizeof(Header))/sizeof(Block)) \rceil$$

ou seja, memória suficiente para acomodar os bytes solicitados *mais* um `Header` contendo a quantidade de blocos para da área reservada e que vai ser retornada ao cliente.

O método `Allocate` então deve percorrer a lista de áreas livres em busca da primeira ocorrência de uma região livre que seja grande o suficiente para satisfazer a solicitação do código cliente. Esta estratégia de ação é denominada de *First-fit*.

[1] Normalmente este nó seria denominado de **cabeça** ou *head* em Inglês; porém para evitar confusão com o campo `Header` (cabeçalho), resolvemos denominá-lo de **Sentinela**.

Uma abordagem alternativa é a *Best-fit*. A estratégia *Best-fit* procura achar uma área livre cujo tamanho seja o mais próximo possível que a quantidade de memória requisitada pelo código cliente. Uma das vantagens desta estratégia é a de minimizar a fragmentação de memória. Por outro lado, esta estratégia requer que o método `Allocate` atravesse *toda* a lista de áreas livres para identificar a área com o melhor "encaixe".

Em qualquer uma das estratégias, existem três possíveis resultados, a saber:

- Não existe uma área livre capaz de satisfazer o pedido do código cliente. Neste caso é necessário lançar a exceção `std::bad_alloc`, definida no cabeçalho `<new>`;
- Foi encontrado uma área livre cujo tamanho é *exatamente* o espaço solicitado. Neste caso a área encontrada é removida da lista de áreas livres e um ponteiro para a área de dados do primeiro bloco é retornado, e;
- Foi encontrado uma área livre cujo tamanho é *maior* do que o espaço solicitado. Neste caso a área é dividida em duas áreas; o tamanho da *primeira área* é definida como sendo o número de blocos solicitado e o tamanho da *segunda área* é igual ao número de blocos que sobraram da área original. Por fim, a segunda área (com o novo tamanho reduzido) é inserida na lista de áreas livres e o método retorna um ponteiro para área de dados do primeiro bloco da primeira área.

Nos últimos dois casos é necessário "saltar" o `Header` do primeiro bloco, visto que ele contém a metainformação sobre quantos blocos constituem a área livre que será retornada ao código cliente. Esta informação é fundamental no momento da operação `Free`, para que o GM possa saber quantos blocos estão sendo devolvidos pelo código cliente.

Suponha agora que o código cliente deseja alocar 44 bytes de memória. Isso representa uma área de 3 blocos de memória: 3×16 bytes $= 48$ bytes $- 4$ bytes (do `Header`) $= 44$ bytes. A Figura 4 apresenta o mapa de memória do GM depois da realização de uma operação `Allocate` correspondente, cuja solicitação se caracteriza como o terceiro caso descrito acima.

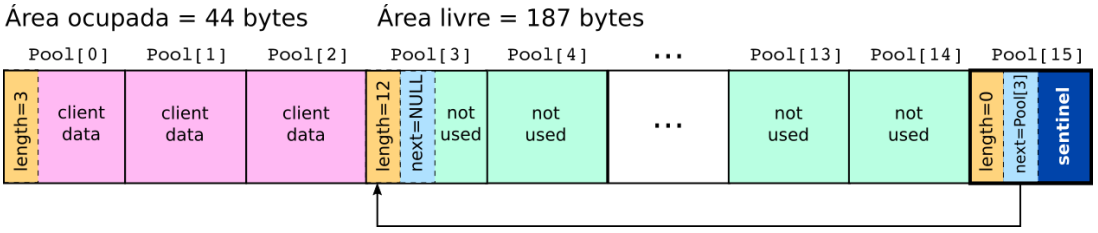


Figura 4: Mapa de memória do GM após uma alocação de 44 bytes. Repare que o nó Sentinela foi ajustado para apontar para o novo início da lista de áreas livres. Por enquanto, esta lista possui apenas 1 nó, com uma área de 12 blocos livres. No início do `Pool`, os três primeiros blocos foram cedidos ao código cliente; repare que no início desta área ocupada o campo `Header` (inacessível ao cliente) indica que a área é formada por 3 blocos.

Supondo que várias alocações e liberações de memória foram realizas, o mapa de memória do GM poderia, por exemplo, estar como da Figura 5. Existem duas áreas ocupadas, uma iniciando

no `Pool[3]` com 2 blocos de comprimento e outra iniciando no `Pool[9]` com 4 blocos de comprimento; e três áreas livres na lista que inicia em `Pool[5]` (4 blocos), segue para `Pool[13]` (2 blocos) e termina em `Pool[0]` (3 blocos).

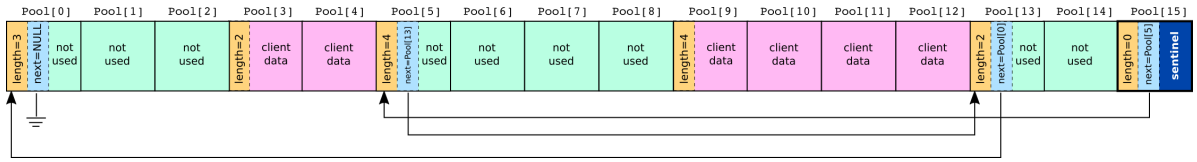


Figura 5: Mapa de memória do GM após diversões operações `Allocate` e `Free`. No exemplo da figura a lista encadeada de áreas livres *não* está ordenada por endereço de cada bloco.

3.2.3 Liberação de Memória

Em teoria, a liberação de uma área previamente ocupada pelo código cliente deveria ser uma operação trivial: inserir a área livre na cabeça da lista, o que representa uma operação de complexidade temporal *constante*.

Porém, existe um problema com esta abordagem. É necessário recordar que o método `Allocate` ocasionalmente divide uma área livre em duas partes, quando o tamanho da área livre é maior do que o solicitado pelo código cliente. Portanto, ao liberar uma área precisamos **combiná-la** com uma eventual área livre adjacente (i.e. vizinha). Se não combinarmos áreas livres adjacentes, produziremos uma lista de áreas livres degenerada, ou seja, fragmentada com um grande número de áreas livres pequenas e adjacentes. Pra piorar, possivelmente as áreas livres seriam tão pequenas que sequer satisfariam os pedidos de memória subsequentes—apesar de existir memória contígua livre suficiente.

Consequentemente, o método `Release` precisa verificar se uma área a ser liberada possui áreas adjacentes livres. Contudo, como saber onde estão as áreas adjacentes livres de maneira eficiente? A solução sugerida é a de manter a lista de áreas livres *ordenadas pelo endereço do início da área*.

Sendo assim, uma área a ser liberada deve ser inserida no local apropriado da lista de áreas livres, de maneira a manter a propriedade de ordenação mencionada. No ponto correto de inserção, verificamos se a área a ser liberada possui alguma área livre adjacente com quem precisa se unir.

O método `Free` recebe como argumento um ponteiro `ptReserved` para a área reservada que será liberada. Determinamos o tamanho do bloco l da área a ser liberada através da metainformação contida no campo `Header` do primeiro bloco apontado por `ptReserved`.

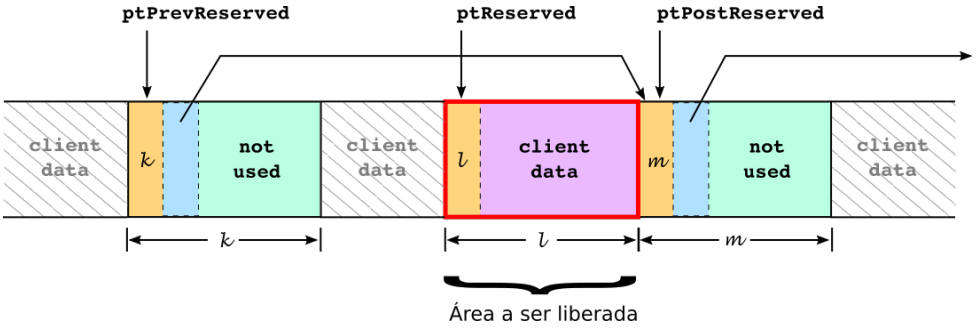
A partir deste ponto, a estratégia sugerida é a de percorrer a lista de áreas livres (recorde que a cabeça da lista é o Sentinela) com dois ponteiros, `ptPrevReserved` e `ptPostReserved`, que inicialmente apontam para o nó cabeça da lista e para o primeiro nó da lista de áreas livres, respectivamente. A condição de parada do laço ocorre quando o `ptPostReserved` alcança o fim da lista (`nullptr`) ou encontra uma área cujo *endereço seja maior do que o endereço apontado por* `ptReserved`. Portanto, a pós-condição do laço é a de que `ptPrevReserved` aponta para o nó anterior ao local de inserção de `ptReserved` e que `ptPostReserved` aponta para o nó posterior ao local de inserção de `ptReserved`.

A partir desta pós-condição, com relação a área a ser liberada existem quatro possibilidades a serem consideradas:

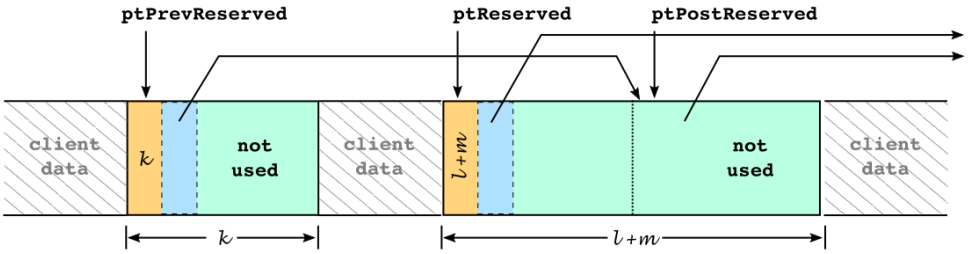
1. As áreas imediatamente anterior e posterior são livres. Neste caso as três áreas são combinadas em uma única área, i.e. os dois nós originais da lista de áreas livres são combinados em um único nó cujo tamanho da área corresponde a soma das três áreas.
2. As áreas imediatamente anterior e posterior são reservadas. Neste caso a área a ser liberada simplesmente é adicionada à lista—é o único caso em que não é necessário combinar nós.
3. A área imediatamente anterior é reservada e a posterior é livre. Neste caso a área a ser liberada é combinada com a área posterior.
4. A área imediatamente anterior é livre e a posterior é reservada. Neste caso a área a ser liberada é combinada com a área anterior.

Podemos tratar as 4 situações anteriores em duas etapas consecutivas:

- Se a área a ser liberada *imediatamente precede* uma área livre (ver Figura 6a), as duas áreas são combinadas em uma só (ver Figura 6b). Caso contrário (ver Figura 7a), a área a ser liberada é inserida na lista *antes* da área apontada por `ptPostReserved` (ver Figura 7b).
- Similarmente, se a área a ser liberada *imediatamente segue* uma área livre (ver Figura 8a), as duas áreas são combinadas (ver Figura 8b). Caso contrário (ver Figura 7a), a área a ser liberada é inserida na lista *depois* da área apontada por `ptPrevReserved` (ver Figura 7c).

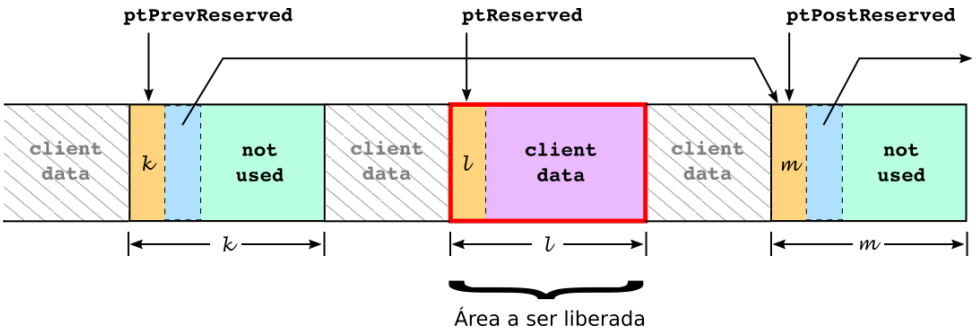


(a) Área a ser liberada imediatamente precede uma *área livre*.

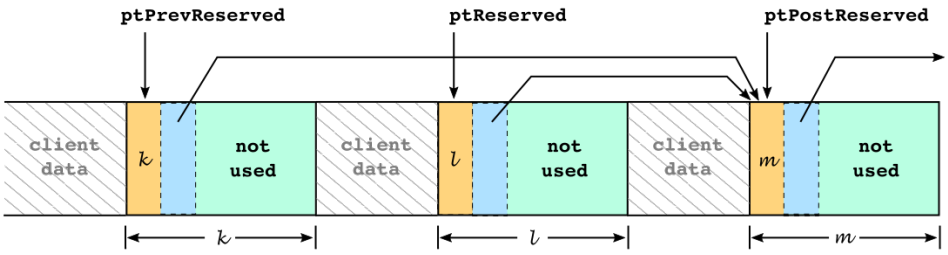


(b) Neste caso as duas áreas são combinadas: $l+m$. O campo `ptReserved->mp_Next` passa a apontar para o mesmo endereço que `ptPostReserved->mp_Next` aponta. Note que o campo `ptPrevReserved->mp_Next` ainda será ajustado na próxima etapa, de maneira a apontar (corretamente) para `ptReserved`.

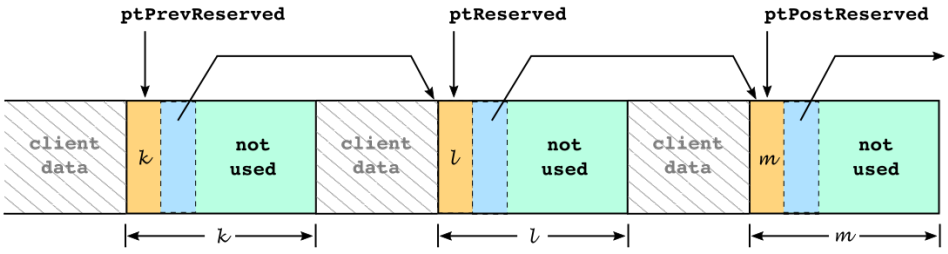
Figura 6: Inserção de área a ser liberada na lista de áreas livres que imediatamente **precede** uma **área livre**.



(a) Área a ser liberada imediatamente precede uma *área reservada*. O mesmo desenho também ilustra o caso em que a área a ser liberada segue uma *área reservada*.

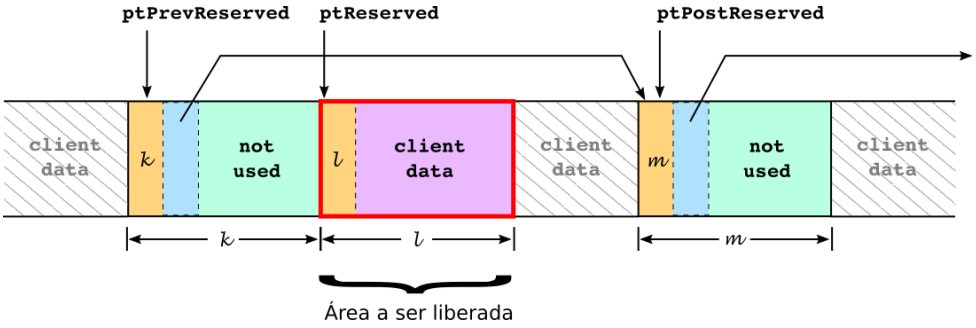


(b) Neste caso a área a ser liberada é inserida antes do nó `ptPostReserd`. A lista será ajustada posteriormente na segunda etapa.

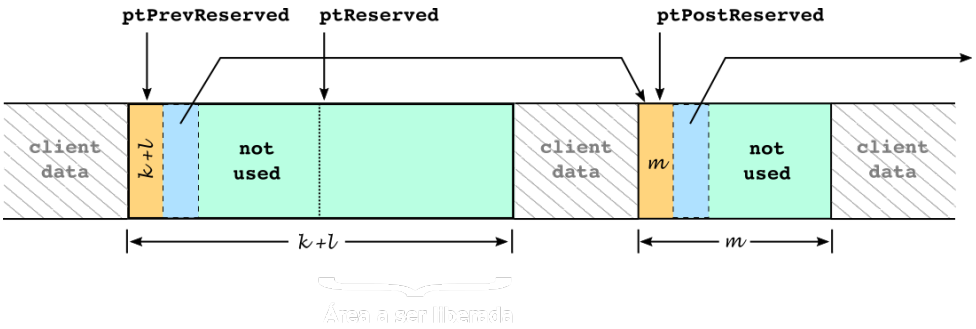


(c) Neste caso a área é inserida depois do nó `ptPrevReserved`.

Figura 7: Inserção de área a ser liberada na lista de áreas livres que imediatamente **precede** uma **área reservada** e/ou imediatamente **segue** uma **área reservada**.



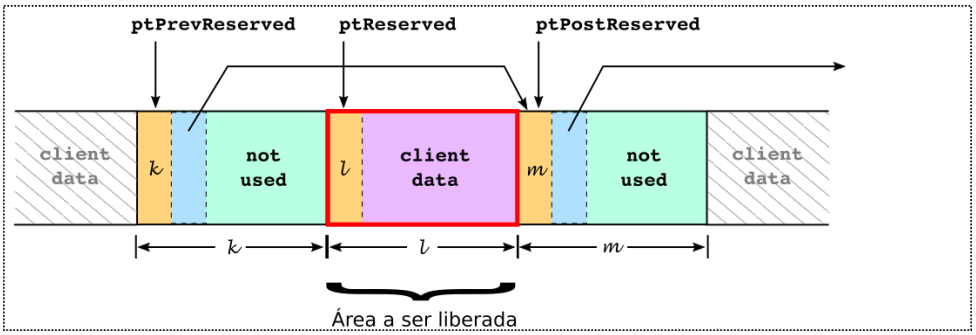
(a) Área a ser liberada imediatamente segue uma *área livre*.



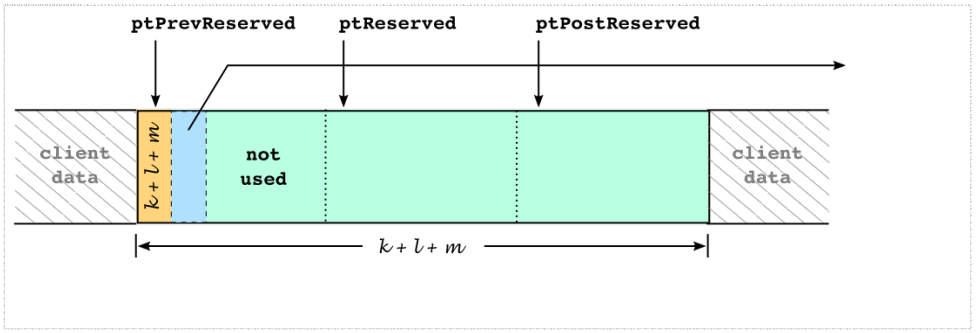
(b) Neste caso as duas áreas são combinadas: $k + l$. As ligações da lista de áreas livres permanece inalterada.

Figura 8: Inserção de área a ser liberada na lista de áreas livres que imediatamente **segue** uma **área livre**.

Ao aplicar as duas etapas consecutivas, automaticamente são resolvidos casos como o da Figura 9, no qual a área a ser liberada está imediatamente entre duas áreas livres.



(a) Área a ser liberada imediatamente segue e precede *áreas livres*.



(b) Neste caso a área a ser liberada é adicionada (em dois passos) à área livre que precede, cujo tamanho passa a ser $k+l+m$ e o ponteiro `ptPrevReserved->mp_Next` aponta para o mesmo nó que `ptPostReserved->mp_Next` apontava.

Figura 9: Inserção de área a ser liberada na lista de áreas livres que, simultaneamente **segue** e **antecede** uma **área reservada**.

Uma vez que é necessário manter a lista de áreas livres ordenada, a complexidade temporal do método `Free` é determinado pelo número de iterações na lista necessárias para encontrar a posição onde a inserção deve acontecer. No pior caso, a complexidade é $O(n)$, onde n corresponde ao número de blocos do arranjo `Pool`.

3.3 Considerações Sobre a Estratégia First-fit

A combinação das duas decisões de implementação—manter a lista de áreas livres ordenada pelo endereço e a estratégia de alocação de usar a primeira área que satisfaz o pedido de memória do código cliente ou *First-fit*—em algumas situações pode levar a uma degradação de desempenho. A queda de desempenho ocorre porque as área livres menores tendem a aparecer perto da cabeça da lista de áreas livres, enquanto que as áreas livres maiores tendem a se concentrar na calda da lista.

Esta "separação" de áreas pequenas no início da lista e áreas grandes no final ocorre devido a política *First-fit*, na qual a solicitação de memória é sempre respondida com a primeira área que seja grande o suficiente. Com isso, quando a área é grande demais ela é dividida em duas áreas e a

(menor) que sobra é inserida de volta na lista. Isso faz com que, eventualmente, muitas áreas perto da cabeça da lista sejam pequenas demais para satisfazer a maioria das solicitações subsequentes. Mesmo assim, a GM precisa visitar estas áreas cada vez que a lista é percorrida.

O uso de um bloco de memória de tamanho mínimo ameniza parcialmente esta tendência. Ou seja, o tamanho mínimo do bloco determina o limite inferior além do qual uma área não é mais dividida. Por exemplo, na implementação sugerida o tamanho do bloco é 16 bytes. Consequentemente, qualquer solicitação de memória até 12 bytes pode ser satisfeita em tempo constante porque a primeira áreas na lista de áreas livres é garantida de ter, pelo menos, 1 bloco de comprimento.

4 Avaliando o Desempenho do GM `SLPool`

Um dos problemas inerentes ao desenvolvimento de um GM é que é difícil avaliar seu desempenho ou prever como será seu funcionamento em uma aplicação real. Isto porque cada aplicação apresenta uma sequência própria de operações `Allocate` e `Free`, invocadas com frequências diferentes, e com tamanhos de memória variados. Assim, o mesmo GM pode apresentar um fraco desempenho para uma certa aplicação e um ótimo desempenho para outra aplicação.

No caso deste projeto de programação, recomenda-se o desenvolvimento de um sistema de *carga de simulação* e visualização do GM, para que seja possível avaliar o GM e depurar seu funcionamento, respectivamente.

Uma sistema de carga de simulação nada mais é do que uma sequência de operações `Allocate` e `Free` que são geradas por um programa cujo comportamento tenta simular o comportamento de uma aplicação real. Uma aplicação real periodicamente faz chamadas a estas funções para trabalhar com memória dinâmica. A aplicação utiliza esta memória por um certo tempo e depois chama `Free` para retornar a memória ao GM.

Para implementar este comportamento, o programa-simulação deve ter a seguinte estrutura geral:

```
1 for ( time t = 0 ; t < timeLimit ; ++t )
2     \\ Executar aqui a operação associada ao instante t_i.
```

A cada passo do tempo, a aplicação (1) libera toda memória que foi programada para ser liberada no instante t_i, e; (2) solicita α bytes de memória e programa sua liberação s unidades de tempo no futuro. Os valores α e s são gerados randomicamente dentro de um intervalo definido. Por exemplo, $\alpha \in [100, 2000]$ bytes e $s \in [1, 100]$ passos de tempo.

Para mantermos controle sobre as operações pendentes, basicamente é necessário associar a operação de alocação/liberação com uma marca de tempo (*time stamp*). Estes pares de associação operação-tempo são armazenados em uma estrutura de dados *lista de prioridade*[2](*priority queue*), por exemplo. A cada momento a simulação recupera qual a operação que tem o menor *time stamp* e o compara o instante atual t_i para saber se é hora de executá-la.

Veja um exemplo geral para uma rotina de teste:

[2]Uma lista de prioridade é uma coleção de elementos chave-informação no qual é possível recuperar o elemento que tem a maior (ou menor) chave em tempo constante. No caso do uso sugerido neste trabalho, estamos interessado em recuperar a operação que possui o menor *time stamp*, ou seja, a operação que precisa ser executada mais brevemente.

```
1  void StoragePoolTest( const StoragePool& _pool, std::time_t _timeLimit ) {
2      // [1] Setup random numbers generator for memory size, say [100,2000] bytes.
3      // [2] Setup random numbers generator for time intervals, say [1,100] units.
4      // [3] Create the pirority queue std::priority_queue 'pq' and
5      //     insert some events comprising the simulation.
6      // Assuming there is a class Event that creates a pair address/time-stamp.
7
8      // Run simulation fot the time set by the client.
9      for ( std::time_t t( 0 ) ; t < _timeLimit ; ++t )
10     {
11         while( !pq.empty() ) { // Run while we have events pending or time to run.
12             Event ev = pq.top(); // Access the event with the smallest time-stamp.
13             if ( event.getTimeStamp() > t ) break; // Still some time left....
14
15             // When we got here, the top event has run out of time.
16             pq.pop() // Remove event from priority queue.
17             _pool.Free( ev.getMemoryPtr() ); // Calling free operator.
18             delete ev;
19         }
20         auto memSize = getRandomForSize();
21         void* const add = _p.Allocate( memSize );
22         auto elapsedTime = getRandomTimeInterval();
23         std::time_t releaseTime = t + elapsedTime; // Set time stamp some time from now.
24         pq.push( new Event( add, releaseTime ) ); // Creating a new simulation event.
25         // Here you might want to show the memory map on the screen,
26         // or write it to a log file, for debugging purpose.
27     }
```

Um teste empírico mais simples com a finalidade de comparação de tempo de execução poderia ser montado da seguinte forma: *i)* Definir manualmente em um programa uma sequência de comandos `new` e `delete`; *ii)* Medir o tempo de execução do programa utilizando o `SLPool`; *iii)* Medir o tempo de execução da mesmo programa utilizando o SO, e; *iv)* Repetir este teste várias vezes e calcular a média dos tempos para ambos os casos. Com isso será possível determinar qual o gerenciador de memória mais eficiente, considerando a massa de testes preparada.

5 Tarefas e Avaliação do Projeto

Para a implementação deste projeto **é obrigatório** a implementação da sua própria lista encadeada. Não serão aceitas soluções que utilizem as estruturas de dados da biblioteca externas, como STL (e.g. `list`, `stack`, `vector`, etc.) ou `boost`, por exemplo[3].

O programa completo deverá ser entregue sem erros de compilação, testado e totalmente documentado. O projeto será avaliado sob os seguintes critérios:-

1. Sobrescreve corretamente os métodos `new` e `delete` (5%);
2. Implementa corretamente a classe `StoragePool` e `SLPool` (5%);

[3] Com exceção da classe `std::priority_queue` que pode ser usada.

3. Implementa corretamente a inicialização da classe `SLPool` (5%);
4. Implementa corretamente o método `Allocate` (10%);
5. Implementa corretamente o método `Free` (15%);
6. Desenvolveu código cliente capaz de provar que a classe `SLPool` funciona corretamente (15%);
7. Desenvolveu código para testar o desempenho de `SLPool`, de maneira comparativa com o SO (15%);
8. Desenvolveu algum tipo de visualização textual do mapa de memória do `SLPool` para fins de depuração (15%);
9. Estende a classe `SLPool` de maneira a dar suporte a política de alocação *Best-fit*, além da política *First-fit* (ver Seção 3.2.2) (15%).

A pontuação acima não é definitiva e imutável. Ela serve apenas como um guia de como o trabalho será avaliado em linhas gerais. É possível a realização de ajustes nas pontuações indicadas visando adequar a pontuação ao nível de dificuldade dos itens solicitados.

Os itens abaixo correspondem à descontos, ou seja, pontos que podem ser retirados da pontuação total obtida com os itens anteriores:-

- Presença de erros de compilação e/ou execução (até −20%)
- Falta de documentação do programa com estilo Doxygen (até −10%)
- Vazamento de memória (até −10%)
- Falta de um arquivo texto `README` contendo, entre outras coisas, identificação da dupla de desenvolvedores; instruções de como compilar e executar o programa; lista dos erros que o programa trata; e limitações e/ou problemas que o programa possui/apresenta, se for o caso (até −20%).

Pontos extras poderão ser conquistados se o trabalho original for realizado **completamente** e foram incluídos:

- Uma visualização gráfica do funcionamento do GM, e;
- Implementar a classe `DLPool` derivada de `StoragePool` que utiliza uma *lista duplamente encadeada* para gerenciar as áreas livres do GM.

Boas práticas de programação

Recomenda-se fortemente o uso das seguintes ferramentas:-

- Doxygen: para a documentação de código e das classes;
- Git: para o controle de versões e desenvolvimento colaborativo;
- Valgrind: para verificação de vazamento de memória;
- gdb: para depuração do código; e
- Makefile: para gerenciar o processo de compilação do projeto.

Procure organizar seu código em várias pastas, conforme vários exemplos apresentados em sala de aula, com pastas como `src` (arquivos `.cpp`), `include` (arquivos `.h`), `bin` (arquivos `.o` e executável) e `data` (arquivos de entrada e saída de dados ou de testes).

Para a implementação eficiente deste projeto é importante visualizar constantemente o que está acontecendo no mapa de memória do GM à medida que operação `new` e `delete` são realizadas pelo código cliente. Além disso, é importante garantir que o GM vai manter a integridade dos dados do código cliente, a medida que o GM começa a ficar cheio e/ou várias operações de alocação e liberação foram realizadas.

Note também que o GM não possui um mecanismo para verificar *segmentation fault* ou falha de segmentação no uso da memória cedida ao código cliente. Como seria possível melhorar o GM para que ele possa detectar *memory access violation*?

6 Autoria e Política de Colaboração

O trabalho pode ser realizado **individualmente** ou em **duplas**, sendo que no último caso é importante, dentro do possível, dividir as tarefas igualmente entre os componentes.

Qualquer equipe pode ser convocada para uma entrevista. O objetivo da entrevista é duplo: confirmar a autoria do trabalho e determinar a contribuição real de cada componente em relação ao trabalho. Durante a entrevista os membros da equipe devem ser capazes de explicar, com desenvoltura, qualquer trecho do trabalho, mesmo que o código tenha sido desenvolvido pelo outro membro da equipe. Portanto, é possível que, após a entrevista, ocorra redução da nota geral do trabalho ou ajustes nas notas individuais, de maneira a refletir a verdadeira contribuição de cada membro, conforme determinado na entrevista.

O trabalho em cooperação entre alunos da turma é estimulado. É aceitável a discussão de ideias e estratégias. Note, contudo, que esta interação **não** deve ser entendida como permissão para utilização de código ou parte de código de outras equipes, o que pode caracterizar a situação de plágio. Em resumo, tenha o cuidado de escrever seus próprios programas.

Trabalhos plagiados receberão nota **zero** automaticamente, independente de quem seja o verdadeiro autor dos trabalhos infratores. Fazer uso de qualquer assistência sem reconhecer os créditos apropriados é considerado **plagiarismo**. Quando submeter seu trabalho, forneça a citação e reconhecimentos necessários. Isso pode ser feito pontualmente nos comentários no início do código, ou, de maneira mais abrangente, no arquivo texto `README`. Além disso, no caso de receber assistência, certifique-se de que ela lhe é dada de maneira genérica, ou seja, de forma que não envolva alguém tendo que escrever código por você.

7 Entrega

Você deve submeter um único arquivo com a compactação da pasta do seu projeto. Se for o caso, forneça também o link Git para o seu projeto. O arquivo compactado deve ser enviado **apenas** através da opção Tarefas da turma Virtual do Sigaa, em data divulgada no sistema.

◄ FIM ►