IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 10 – Gerenciamento de Memória
(material baseado nas notas de aula do Prof. Silvio Sampaio)

Objetivo

- Introduzir os conceitos de alocação dinâmica e gerenciamento de memória em **C++**
- Para isso, estudaremos:
  - Alocação estática x alocação dinâmica
  - Comandos básicos de gerenciamento de memória
- Ao final da aula espera-se que o aluno seja capaz de:
  - Distinguir a alocação estática da alocação dinâmica
  - Desenvolver programas capazes de gerenciar dinamicamente a memória do computador

Alocação estática x dinâmica

- As linguagens de programação **C** e **C++** permitem dois tipos de alocação de memória:
  - Estática
  - Dinâmica
- Na alocação estática, o espaço de memória para as variáveis é reservado no início da execução, não podendo ser alterado depois

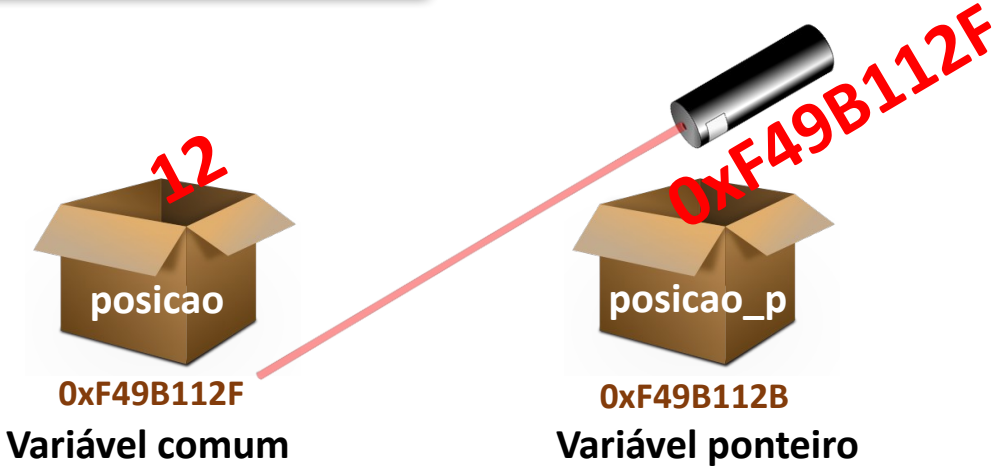
```
1
2       int numero;
3       int pontuacao[20];
4
5
```

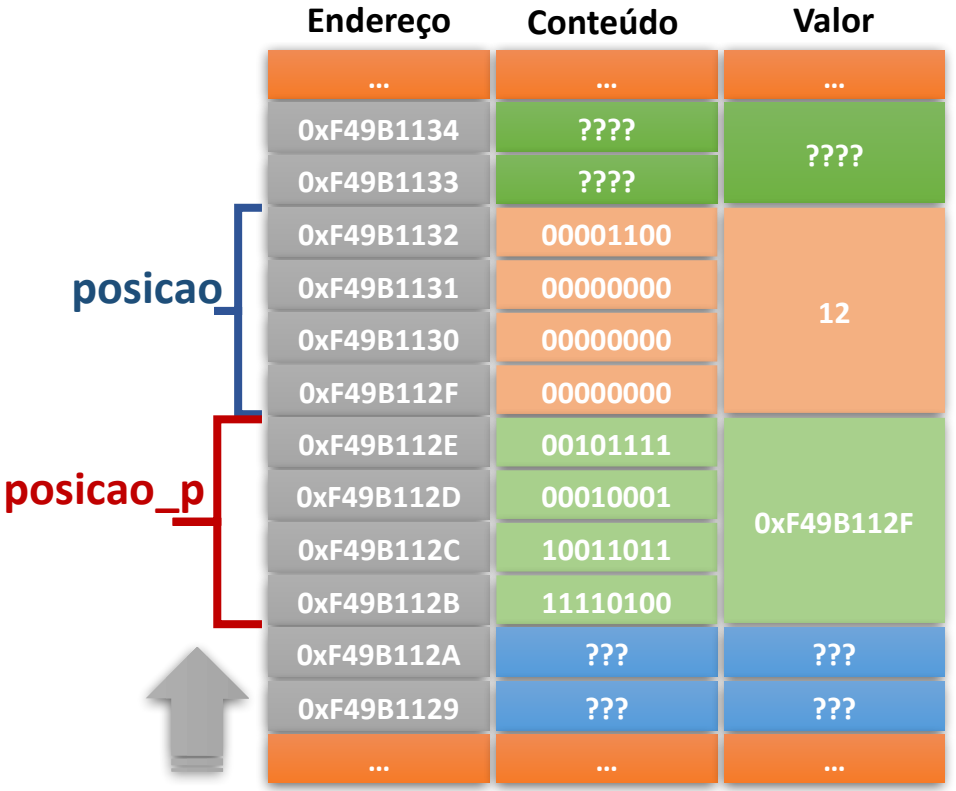
- Na alocação dinâmica, o espaço de memória para as variáveis pode ser alocado dinamicamente durante a execução do programa
  - **Ponteiros se fazem necessários**

Ponteiros

- Tipo especial de variável que armazena endereços de memória e permite acessá-los diretamente

```
1
2       ...
3       int posicao = 12;
4       int *posicao_p;
5       posicao_p = &posicao;
6       ...
7
```





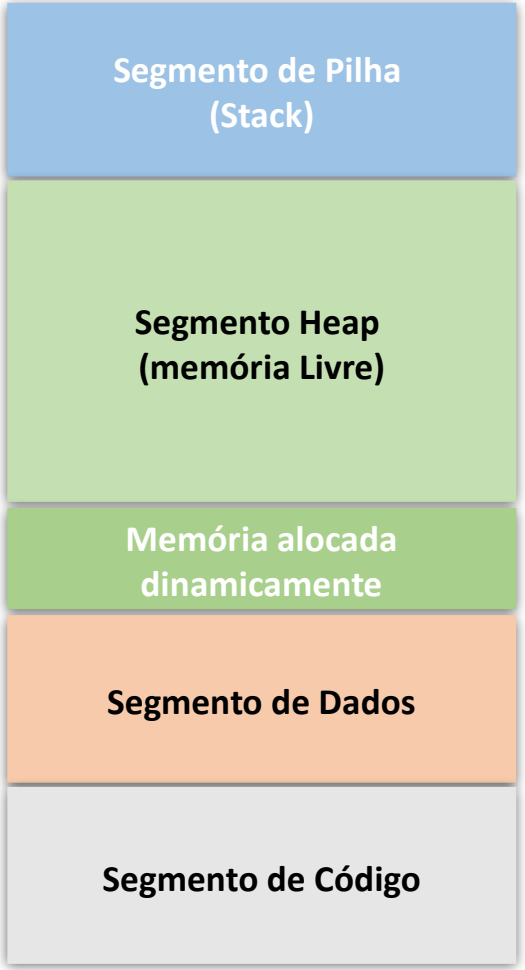
Ponteiros

- Operadores utilizados para manipular ponteiros
  - Operador de acesso à memória **&** (**referenciamento**)
    - retorna o endereço de uma variável
  - Operador de indireção * (**desreferenciamento**)
    - retorna o conteúdo do endereço de uma variável apontada
    - * de indireção ≠ * de multiplicação ≠ * do tipo ponteiro

```
1
2       ...
3       int posicao = 12;
4       int *posicao_p = &posicao;
5       int nova_posicao = *posicao_p;
6       ...
7
```

Alocação dinâmica

- A alocação dinâmica de memória é um mecanismo bastante útil na solução de problemas que exigem grandes conjuntos de dados
  - Meio pelo qual um programa pode obter memória enquanto está em execução, sendo gerenciado pelo próprio programador
- Ela pode oferecer grandes benefícios em termos de desempenho e de utilização de recursos
  - A memória alocada dinamicamente é obtida através do segmento **HEAP**, onde apenas o espaço de memória necessário em um dado momento é efetivamente utilizado



Alocação Dinâmica

- As linguagens **C** e **C++** permitem que o programador tenha um alto grau de controle sobre a máquina através da alocação dinâmica
- Elas possuem ambas dois comandos básicos para gerenciamento de memória
  - Comandos da linguagem C++:
    - **new** aloca memória
    - **delete** libera memória alocada
  - Comandos da linguagem C:
    - **malloc** aloca memória
    - **free** libera memória alocada

Alocação dinâmica em C++

- Operador **new**
  - Aloca uma área de memória do tamanho correspondente à representação do tipo declarado
  - Retorna um ponteiro do tipo declarado apontando para o início da área alocada, ou **NULL** caso não seja possível alocar a memória requisitada
- Sintaxe para alocação de uma variável ponteiro do tipo **T**

```
1
2
3
4
5
```

```
...
T *p = new T;
...
```

```
1
2
3
4
5
6
7
```

```
...
int *p = new int;
float *q = new float;
Ponto *umPonto = new Ponto;
...
```

Alocação dinâmica em C++

- Exemplo:

```cpp
#include <iostream>

int main()
{
    int *p = new int;  // alocação de variável ponteiro do tipo inteiro

    /* IMPORTANTE: convém sempre verificar se a alocação ocorreu corretamente,
       ou seja, se o retorno do operador new é diferente de NULL */

    if( p )
    {
        std::cout << "Memoria alocada" << std::endl;
        std::cout <<  p << std::endl;  // imprime o endereço de p
        std::cout << *p << std::endl;  // imprime o conteúdo de p
        *p = 10;                       // inicializa o conteúdo de p com 10
        std::cout << *p << std::endl;  // imprime o conteúdo de p
    }
    else
        std::cout << "Alocacao impossivel" << std::endl;
    return 0;
}

```

Alocação dinâmica em C++

- Operador **delete**
  - Libera a área de memória previamente alocada no sistema utilizando o seu endereço inicial como parâmetro
  - O sistema operacional se encarrega de gerenciar lacunas do **heap**
- Exemplo:

```
1   #include<iostream>
2
3   int main()
4   {
5       int *p = new int;  // alocação de variável ponteiro do tipo inteiro
6       if( p )            // verifica se a alocação ocorreu corretamente
7           delete p;      // libera a memória alocada
8       return 0;
9   }
10
```

Erros comuns da alocação dinâmica

- Não alocar memória antes de acessar o conteúdo do ponteiro
  - Para acessar o conteúdo, sempre deve ser verificado se o ponteiro é válido
- Copiar o conteúdo do ponteiro ao invés do conteúdo da variável apontada
- Não liberar memória alocada previamente quando ela passar a ser desnecessária
- Tentar acessar o conteúdo de um ponteiro depois da sua memória já ter sido liberada
- O valor nulo deve ser sempre atribuído ao ponteiro após à sua liberação de memória

Exercício

- O que está errado neste programa?

```
1   #include <iostream>
2
3   int main()
4   {
5       int a, b, *p;
6       a = 2;
7       *p = 3;
8       b = a + (*p);
9       std::cout << a << std::endl;
10      return 0;
11  }
12
```

Exercício

- O que está errado neste programa?

```
1   #include <iostream>
2
3   int main()
4   {
5       int a, b, *p;
6       a = 2;
7       *p = 3;
8       b = a + (*p);
9       std::cout << a << std::endl;
10      return 0;
11  }
12
```

  - **Resposta:** A variável **p** (tipo ponteiro para inteiro) foi criada, porém não inicializada, ou seja, não aponta para nenhuma posição de memória válida (pode estar a apontar para “algum lugar” na memória, por causa do “lixo” deixado na posição onde **p** foi alocada). Assim, a instrução da **linha 7** poderá causar um erro de acesso à memória!!!

Exercício

- O que será impresso no seguinte programa?

```
1  #include <iostream>
2  int main()
3  {
4      double a, *p, *q;
5      a = 3.14;
6      std::cout << a << std::endl;
7      p = &a;
8      *p = 2.718;
9      std::cout << a << std::endl;
10     a = 5;
11     std::cout << *p << std::endl;
12     p = NULL;
13     p = new double;
14     *p = 20;
15     q = p;
16     std::cout << *p << std::endl;
17     std::cout << a << std::endl;
18     delete p;
19     std::cout << *q << std::endl;
20     return 0;
21 }
22
```

Exercício

- O que será impresso no seguinte programa?

```
1  #include <iostream>
2  int main()
3  {
4      double a, *p, *q;
5      a = 3.14;
6      std::cout << a << std::endl;
7      p = &a;
8      *p = 2.718;
9      std::cout << a << std::endl;
10     a = 5;
11     std::cout << *p << std::endl;
12     p = NULL;
13     p = new double;
14     *p = 20;
15     q = p;
16     std::cout << *p << std::endl;
17     std::cout << a << std::endl;
18     delete p;
19     std::cout << *q << std::endl;
20     return 0;
21 }
22
```

Resposta:

#> 3.14
#> 2.718
#> 5
#> 20
#> 5
#> LIXO

Alocação dinâmica de vetores

- **Vetores são ponteiros** em linguagem **C/C++**
- A liberação de memória de vetores deve ser efetuada com o comando **delete [ ]**

```cpp
#include <iostream>

int main()
{
    int a[10] = {1,2,3,4,5,6,7,8,9,0};
    int *b;
    b = a;

    b[5] = 100;
    std::cout << a[5] << std::endl;
    std::cout << b[5];
    return 0;
}
```

Alocação Estática

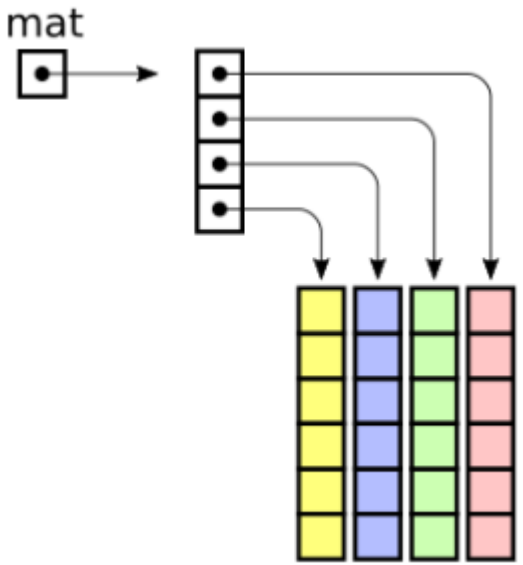
```cpp
#include <iostream>

int main()
{   // b = a não é permitido aqui
    int a[10] = {1,2,3,4,5,6,7,8,9,0};
    int *b;
    b = new int[10];

    b[5] = 100;
    std::cout << a[5] << std::endl;
    std::cout << b[5];
    delete[] b;
    return 0;
}
```

Alocação Dinâmica

Alocação dinâmica de matrizes

- Alocação de matrizes se faz da mesma forma que para vetores, incrementada do conceito de indireção múltipla (ponteiro de ponteiros)
- Ponteiro de ponteiros se aplica a qualquer dimensão desejada

```
1  #include <iostream>
2  int main()
3  {
4      float **matriz;    // ponteiro de ponteiros para a matriz
5      int linhas = 10, colunas = 15;
6      matriz = new float*[linhas];  // aloca as linhas da matriz
7
8      if( matriz != NULL )
9          for( int i = 0; i < linhas; i++ )
10         {
11             matriz[i] = new float[colunas]; // aloca as colunas da matriz
12             if( matriz[i] == NULL ) {
13                 std::cout << "Memoria Insuficiente" << std::endl;
14                 break;
15             }
16         }
17     return 0;
18 }
19
```

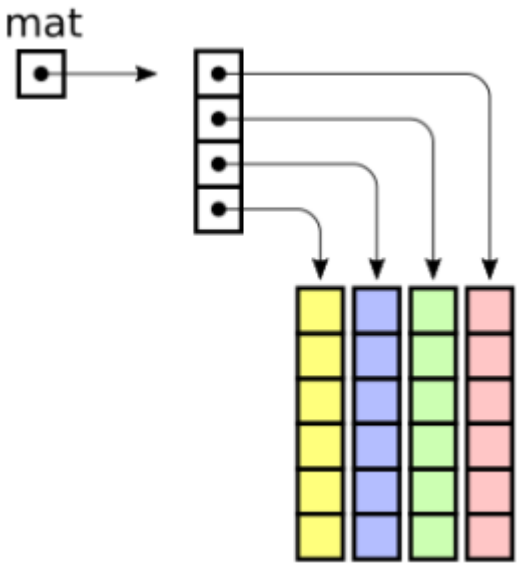


inf.ufpr.br/roberto/ci067/14_alocmat.html

Alocação dinâmica de matrizes

- A liberação de memória das matrizes deve ser efetuada para todos os ponteiros da indireção múltipla

```
1  #include <iostream>
2
3  int main()
4  {
5      float **matriz;  // ponteiro de ponteiros para a matriz
6      int linhas = 10, colunas = 15;
7
8      ... // Considerando a alocação de memória efetuada
9      if( matriz != NULL )
10     {
11         for( int i = 0; i < linhas; i++ )
12             delete [] matriz[i];  // libera as colunas da matriz
13         delete [] matriz;         // libera as linhas da matriz
14     }
15     return 0;
16 }
17
```



Use *nullptr* ao invés de *NULL*

```
#include <iostream>


int main(int argc, char const *argv[])
{
    int * x = nullptr;

    int a = 89;

    x = &a;

    std::cout << "Endereco de x = " << &x << std::endl;
    std::cout << "Valor de x = " << x << std::endl;
    std::cout << "Endereco de a = " << &a << std::endl;
    std::cout << "Valor apontado por x = " << *x << std::endl;
    return 0;
}
```

- Em C++11, variáveis do tipo ponteiro devem ser inicializados com o valor especial ***nullptr***
- A palavra reservada ***nullptr*** foi introduzida no C++ para representar o endereço 0
- Exemplo de declaração e inicialização de um ponteiro com o valor ***nullptr***:

Ponteiros para constantes

- Se quisermos armazenar o endereço de uma constante em um ponteiro, precisamos usar um ponteiro para constante
- Exemplo:

```
const int SIZE = 6;
const double taxas[SIZE] = { 18.55, 17.45, 12.85, 14.97, 10.35, 18.89 };

void mostraTaxas(const double *taxas, int size) {
    for (int count = 0; count < size; ++count) {
        cout << "Taxa " << (count + 1) << " é " << *(taxas + count) << "%." << endl;
    }
}
```

Trabalho – Unidade 1

- Implementar a classe “Imaginario” (números imaginários) baseado no código fornecido.
- Obrigatório
  - Código deve compilar com “make”. Portanto, criem corretamente o makefile.
  - Implementar os métodos e sobrecargas dos operadores (no código está como “Definição base”)
- A nota será dada fazendo diversos casos de teste utilizando todos os métodos obrigatórios
- Nota 0 em casos de:
  - Plágio
  - Código não compila

Trabalho – Unidade 1

$$a+bi \qquad\qquad i=\sqrt{-1}$$