FAGAMMON
FACULDADE PRESBITERIANA GAMMON
30 ANOS

Algoritmos e Estrutura de Dados II

Aula 10

Alocação Dinâmica

Claudiane Maria Oliveira
claudianeo@gmail.com

Introdução

- Limitação no uso de vetores
  - Ao declarar um vetor, precisamos nos preocupar com o tamanho dele.
  - Quando o tamanho só é conhecido em tempo de execução, geralmente utilizamos um código como o abaixo.

```
// pedimos o tamanho ao usuário
// e o usamos ao declarar o vetor
int n;
cin >> n;
int vetor[n];
```

Introdução

- Quando fazemos:

```
int n;
cin >> n;
int v[n];
```

- Estamos usando uma região de memória chamada pilha de execução (stack).

- O problema é que essa região possui limitação de espaço!

Alocação Estática de Memória

- Se declaramos:

  int x;

- Estamos reservando (alocando) espaço na memória para guardar um inteiro.

- Se declaramos:

  int x[10];

- Estamos reservando (alocando) espaço na memória para guardar dez inteiros.

Alocação Dinâmica de Memória

- **Alocação dinâmica**, significa que nós podemos, "manualmente", pedir o sistema para reservar mais memória durante a execução do programa.
- Para isso usamos a palavra-chave **new**
- Ao usar: **new int**
  - Vemos que o operador new espera um tipo de dado (int, no exemplo).
  - O sistema encontra (e reserva) um bloco de memória com tamanho suficiente para guardar um valor do tipo int.
  - É retornado o endereço da memória reservada (ou seja, a primeira posição dessa região de memória).

Alocação Dinâmica de Memória

- Exatamente! Um ponteiro é uma variável capaz de guardar um endereço de memória.

```
int *p;
```

- Se nós então guardarmos o endereço retornado pelo **new** em um ponteiro…

```
int *p = new int;
```

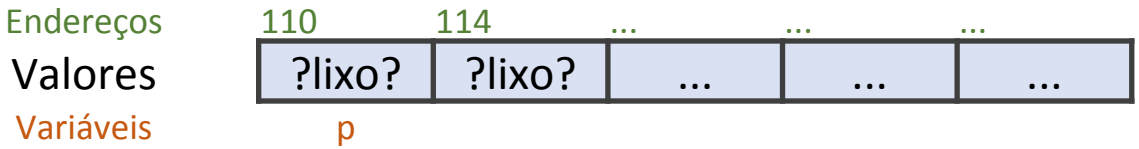
- Podemos depois guardar e consultar valores na região de memória alocada pelo new, usando o ponteiro (usando o operador de indireção *).

```
int *p = new int;
*p = 5;
```

Alocando Memória Dinamicamente

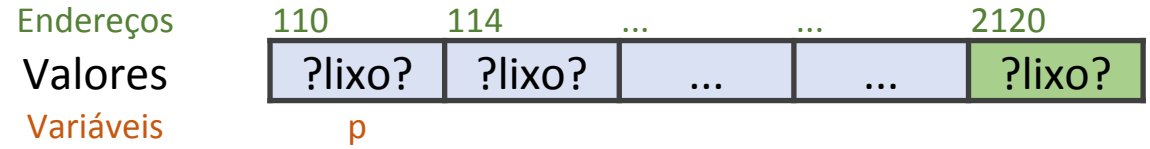
- Primeiro, se apenas declararmos um ponteiro p, é reservado um espaço de memória na pilha para essa variável (e como ela não foi inicializada ela tem "lixo").

```
int *p;
```



- Se executássemos apenas o comando new int, estaríamos alocando espaço na memória para guardar um inteiro.

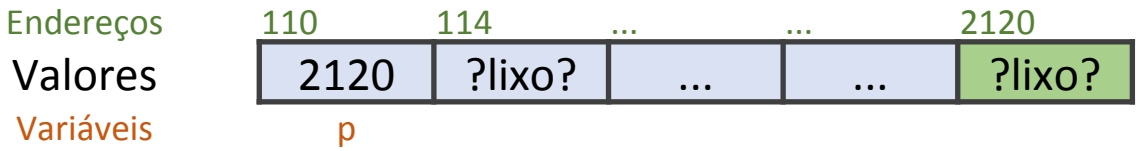
```
new int
```



Alocando Memória Dinamicamente

- Por fim, se executarmos o comando completo, declarando o ponteiro e guardando nele o endereço retornado por new int, teríamos:

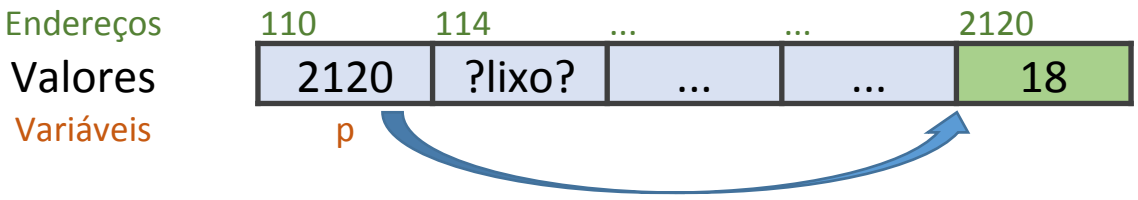
```
int *p = new int;
```



Alocando Memória Dinamicamente

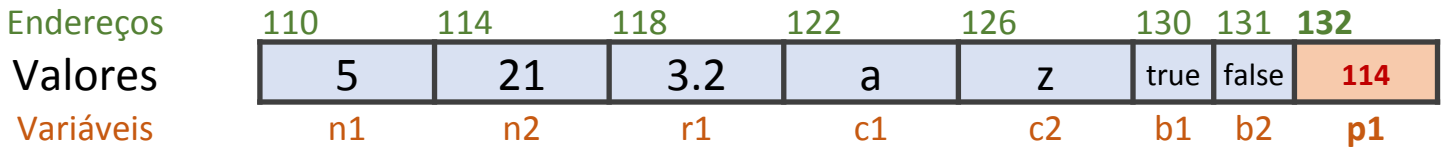
- Ele apenas agora está apontando uma região de memória que alocamos "manualmente" ao invés de apontar para uma variável.

```
int *p = new int;
*p = 18;
```



Desalocando memória

- Quando declaramos variáveis, o sistema reserva memória e o próprio sistema também libera a memória utilizada.



- Já quando usamos alocação dinâmica, nós é que "manualmente" reservamos a memória (através do comando new).

- Portanto, somos nós os responsáveis também por liberar (desalocar) a memória.

Desalocando memória

- Para desalocar memória usamos o comando **delete**.

- E para usar o comando delete precisamos passar para ele o endereço da região alocada:

```
delete endereco
```

```
 6  int main()
 7  {
 8      int *p = new int;
 9
10      // algum código
11
12      delete p;
13  }
```

- Dizemos que p1 agora aponta para n2

Cuidados na desalocação de memória

- Não desalocar memória alocada “manualmente” é um problema grave!

- Basta você lembrar que:
  Todo <u>new</u> executado tem que ter um <u>delete</u> associado.

Cuidados na desalocação de memória

- Às vezes nós alocamos memória dentro de um if ou de um else. Dessa forma, na parte final do programa podemos não ter certeza se a memória foi alocada ou não.
- É exatamente isso que fazemos. C++ possui a palavra-chave **NULL**, que significa nenhum endereço.
- Podemos então inicializar o ponteiro com esse valor. E depois testar se ele tem esse valor antes de desalocar a memória.

Retorna *true* se o ponteiro tiver um endereço válido (ou seja, diferente de NULL).

```
int *p = NULL;

// algum código

if (p)
    delete p;
```

Cuidados na desalocação de memória

- Sempre inicialize seus ponteiros com a palavra-chave NULL.
-
- E teste se o ponteiro é valido (usando if) antes de usá-lo (não só no caso do delete).
-
- Dessa forma, você evita tentativas de acessar equivocadamente endereços de memória que são lixo.

Cuidados na desalocação de memória

- Veja um exemplo de uso do NULL

```
 8      int vetor[10];
 9      for (int i = 0; i < 10; ++i) {
10          cin >> vetor[i];
11      }
12
13      int procurado, posicao;
14      int* ptPos = NULL;
15      cin >> procurado;
16      for (int i = 0; (i < tamanho) and (ptPos == NULL); ++i)
17      {
18          if (vetor[i] == procurado) {
19              posicao = i;
20              ptPos = &posicao;
21          }
22      }
23
24      if (ptPos != NULL)
25          cout << "Encontrado na posicao: " << *ptPos << "\n";
```

Endereços e Vetores

- Mas antes de usarmos os ponteiros vamos dar uma examinada em como uma variável composta (vetor) se comporta.
- O que imprime o código abaixo?

```
 8      int v[10];
 9      for (int i = 0; i < 10; i++) {
10          cin >> v[i];
11      }
12
13      cout << &v[0] << endl;
14      cout <<  v    << endl;
```

Endereços e Vetores

- Mas antes de usarmos os ponteiros vamos dar uma examinada em como uma variável composta (vetor) se comporta.
- O que imprime o código abaixo?

```
 8      int v[10];
 9      for (int i = 0; i < 10; i++) {
10          cin >> v[i];
11      }
12
13      cout << &v[0] << endl;
14      cout <<  v    << endl;
```

```
0x61fef4
0x61fef4
```

- Os dois cout's imprimiram o mesmo valor!!!
- Ou seja, uma variável vetor na verdade guarda um endereço de memória. Justamente o endereço de memória do primeiro elemento do vetor.

Ponteiros e Vetores

- Se uma variável vetor na verdade guarda um endereço de memória, se eu quiser usar um ponteiro para apontar para um vetor...

```
int v[10];
int *p = v;
```

- Veja que a gente não pegou o endereço da variável vetor em si (com operador &), mas sim o próprio valor guardado pela variável.

Ponteiros e Vetores

- O que acontece se eu usar o código abaixo?

```
int v[10];
int *p = v;

for (int i = 0; i < 10; i++) {
    cin >> p[i];
}

cout << "1o elemento: " << p[0] << endl;
```

- O código funciona da mesma forma que se usasse o vetor v.
- Ou seja, um ponteiro pode ser usado para acessar os elementos de um vetor.

Ponteiros e Vetores

- Um vetor pode ser visto como um ponteiro, mas um ponteiro constante!

```
int v[10];
int *p;
p = v;
v = p;
```

Portanto:
Essa linha é válida.
Mas essa não!

Alocando vetores dinamicamente

- Um dos grandes usos de ponteiros é para alocação dinâmica de espaços maiores de memória.
- Ou seja, podemos **alocar dinamicamente um vetor** (ou uma matriz) e usar um ponteiro para apontar essa região.
- Podemos fazer isso dessa forma:

```
int *p = new int[10];
```

Com esse comando, estamos alocando dinamicamente espaço na memória para guardar 10 inteiros e guardando o endereço dessa região no ponteiro *p*.

Como nós acabamos de ver que podemos usar um ponteiro como um vetor, isso significa que podemos ler os dados para essa região, guardar os dados nela, etc., exatamente como fazíamos com uma variável vetor.

Alocando vetores dinamicamente

- Quando fazemos:

```
int n;
cin >> n;
int v[n];
```

- Estamos usando uma região de memória chamada pilha de execução (stack).
- O problema é que essa região possui limitação de espaço!
- Mas usando ponteiros e alocação dinâmica não temos esse problema!
- Com o operador new reservamos outra região de memória que não temos essa limitação de espaço.

Alocando vetores dinamicamente

- Vamos então criar nossa função (nesse exemplo gerando um vetor de tamanho aleatório).

Repare que o tamanho do vetor é retornado por referência.

```
int* criarVetor(int &tam)
{
    // define tamanho do vetor aleatoriamente
    // como um valor qualquer entre 10 e 1000
    srand(time(NULL));
    tam = 10 + rand() % 990;

    // aloca dinamicamente um vetor com o
    // tamanho sorteado
    int *p = new int[tam];

    // retorna o ponteiro para o vetor
    return p;
}
```

Retornando vetores usando ponteiros

```
int* criarVetor(int &tam)
{
    // define tamanho do vetor aleatoriamente
    // como um valor qualquer entre 10 e 1000
    srand(time(NULL));
    tam = 10 + rand() % 990;

    // aloca dinamicamente um vetor com o
    // tamanho sorteado
    int *p = new int[tam];

    // retorna o ponteiro para o vet
    return p;
}
```

```
int main()
{
    int n;

    int *v = criarVetor(n);

    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    // algum código

    delete [] v;
}
```

Usando ponteiros para Registros

- Quando os ponteiros apontam para registros, o acesso aos campos desses registros pode ser realizado por meio do operador ->

```
struct Ponto {
    int x;
    int y;
};

int main() {
    Ponto ponto;
    cin >> ponto.x >> ponto.y;

    Ponto *p;
    p = &ponto;

    cout << (*p).x << " - " << (*p).y << endl;
    cout <<   p->x << " - " <<   p->y << endl;
}
```

As duas linhas estão fazendo exatamente a mesma coisa.

Alocando Matrizes dinamicamente

- Matrizes podem ser vistas como vetores de vetores.

```
int m, n;
int **matriz;

cin >> m >> n;

matriz =  new int*[m];

for (int i = 0; i < m; i++) {
    matriz[i] = new int[n];
}

// algum código que usa a matriz

for (int i = 0; i < m; i++){
    // aqui se desalocam as colunas de cada linha
    delete[] matriz[i];
}

delete[] matriz; // aqui se desalocam as linhas
```

Exercícios

Faça um programa que contenha:

a) Um subprograma que receba um valor N e crie dinamicamente um vetor de N elementos e retorne um ponteiro;

b) Um subprograma que receba um ponteiro para um vetor e um valor N e leia os elementos desse vetor;

c) Um subprograma que receba um ponteiro para um vetor e um valor N e imprima os N elementos desse vetor;

d) Um subprograma que receba um ponteiro para um vetor e libere a memória alocada para o mesmo.

Seu programa deve ler um valor N e chamar os subprogramas necessários para criar um vetor de N elementos. O programa deve ler e imprimir os elementos do vetor e, ao final, liberar a memória alocada para o mesmo.

Aritmética de Ponteiros

- O que faz o programa a seguir?

```
 5  int main()
 6  {
 7      int v[10];
 8
 9      for (int* p = v; p < v + 10; p++)
10      {
11          cin >> *p;
12      }
13
14      for (int i = 0; i < 10; i++)
15      {
16          cout << v[i] << " ";
17      }
18      cout << endl;
19  }
```

Aritmética de Ponteiros


```
 7     bool b;
 8     int i;
 9
10     bool *pB = &b;
11     int *pI = &i;
12
13     cout << pB   << endl;
14     cout << pB+1 << endl;
15     cout << pB+2 << endl;
16     cout << endl;
17
18     cout << pI   << endl;
19     cout << pI+1 << endl;
20     cout << pI+2 << endl;
```

```
0x61ff17
0x61ff18
0x61ff19

0x61ff10
0x61ff14
0x61ff18
```

Veja que no caso do ponteiro **pB**, na linha 13 é exibido o endereço da variável **b**, e ao aumentar o endereço em **1** resulta em um endereço 1 byte à diante.

Mas no caso do ponteiro **pA**, a cada vez que o endereço é aumentado em **1** o resultado é um incremento de **4** bytes no endereço.

Adição de Ponteiros

- Os ponteiros admitem as operações de adição e subtração. Mas ao usar essas operações elas funcionam de acordo com o tipo do ponteiro.
- O tipo bool ocupa 1 byte de memória, portanto, ao adicionar 1 ao ponteiro, o endereço é incrementado em 1 byte.
- Já o tipo int ocupa 4 bytes* e, por isso, ao incrementar 1 ao ponteiro, o endereço é incrementado em 4 bytes.
- Já pensou a confusão que seria se o incremento de 1 em um ponteiro para inteiro incrementasse o endereço em 1 byte? Ele ia pegar ¾ de uma variável inteira.

Aritmética de Ponteiros e Vetores

Mas se nós conseguimos percorrer um vetor com um ponteiro, podemos usar derreferenciação para acessar os elementos do vetor, certo?

```
 4  int main() {
 5      int v[5];
 6
 7      for (int i = 0; i < 5; i++) {
 8          cin >> v[i];
 9      }
10
11      for (int *p2 = v; p2 < v+5; p2++) {
12          cout << *p2 << endl;
13      }
14  }
```

No primeiro laço percorremos o vetor normalmente, como sempre fizemos. Já no segundo percorremos o laço usando um ponteiro e exibimos os valores de cada elemento do vetor.

Repare que poderíamos também fazer os dois laços com ponteiros.

Dica

- Desenhe a representação da memória em papel ao fazer os exercícios de ponteiros.
-
- Invente endereços de memória que façam sentido e use esses valores nas atribuições de ponteiro.
-
- Fazendo isso você conseguirá entender melhor como o código funciona.

Uso de Ponteiros na Passagem de Parâmetros

- Desenhe a representação da memória em papel ao fazer os exercícios de ponteiros.
- Invente endereços de memória que façam sentido e use esses valores nas atribuições de ponteiro.
- Fazendo isso você conseguirá entender melhor como o código funciona.

Uso de Ponteiros na Passagem de Parâmetros


```
 5  int* soma(int *p1, int *p2)
 6  {
 7      int *r = new int;
 8      *r = *p1 + *p2;
 9
10      return r;
11  }
12
13  int main()
14  {
15      int *p1 = new int;
16      int *p2 = new int;
17
18      cin >> *p1 >> *p2;
19
20      int *p3 = soma(p1, p2);
21
22      cout << *p3 << endl;
23
24      delete p1;
25      delete p2;
26      delete p3;
27  }
```

Função recebe dois ponteiros para inteiro por parâmetro.

É declarado um ponteiro de resultado (e é alocada memória dinamicamente).

Estão sendo usados ponteiros com memória alocada dinamicamente para serem usados na função.

Não podemos nos esquecer de desalocar toda a memória alocada.

Variáveis Escalares na Passagem de Parâmetros


```
5   int* soma(int *p1, int *p2)
6   {
7       int *r = new int;
8       *r = *p1 + *p2;
9
10      return r;
11  }
```

```
13  int main()
14  {
15      int a, b;
16
17      cin >> a >> b;
18
19      int *p3 = soma(&a, &b);
20
21      cout << *p3 << endl;
22
23      delete p3;
24  }
```

Ponteiros guardam endereços de memória. Portanto se passarmos os endereços de variáveis escalares a função funcionará normalmente.

Variáveis Escalares na Passagem de Parâmetros


```
 5  void subprograma(int *p1, int *p2)
 6  {
 7      (*p1)++;
 8      (*p2) += 2;
 9  }
10
11  int main()
12  {
13      int a, b;
14
15      cin >> a >> b;
16
17      subprograma(&a, &b);
18
19      cout << a << " " << b << endl;
20  }
```

Veja que o valor das variáveis a e b é alterado ao chamar o subprograma. Isso acontece porque como estamos passando o endereço da variável, o que é alterado no subprograma, na verdade mexe na variável.

E como aumentar o tamanho de um vetor?


```
27  int *criarVetor(int tam) {
28      int *p = new int[tam];
29      return p;
30  }
31
32  void preencherVetor(int v[], int n) {
33      for (int i = 0; i < n; i++) {
34          cin >> v[i];
35      }
36  }
37
38  int main() {
39      int n;
40      cin >> n;
41      int *v = criarVetor(n);
42      preencherVetor(v, n);
43
44      int novoTam;
45      cin >> novoTam;
46      // como aumentar v para o novo tamanho?
```

E como aumentar o tamanho de um vetor?


```
5  int *aumentarVetor(int *v, int tamAtual, int novoTam) {
6      int* novoV = criarVetor(novoTam);
7
8      for (int i = 0; i < tamAtual; i++) {
9          novoV[i] = v[i];
10     }
11
12     delete [] v;
13
14     return novoV;
15 }
```

Alocamos o tamanho que precisamos em uma nova região de memória.

Copiamos os dados atuais do vetor para essa nova posição.

Precisamos nos lembrar de desalocar a memória usada anteriormente. ATENÇÃO: estamos assumindo que quem chamar a função alocou o vetor dinamicamente.

Retornamos então um ponteiro para a nova região de memória.

```
 5  int *aumentarVetor(int *v, int tamAtual, int novoTam) {
 6      int* novoV = criarVetor(novoTam);
 7
 8      for (int i = 0; i < tamAtual; i++) {
 9          novoV[i] = v[i];
10      }
11
12      delete [] v;
13
14      return novoV;
15  }
```

```
38  int main() {
39      int n;
40      cin >> n;
41      int *v = criarVetor(n);
42      preencherVetor(v, n);
43
44      int novoTam;
45      cin >> novoTam;
46
47      v = aumentarVetor(v, n, novoTam);
48
49      exibirVetor(v, novoTam);
50
51      delete [] v;
52  }
```

Passamos o vetor por parâmetro e depois fazemos v apontar para a nova região de memória (ao receber o retorno da função).

Se exibirmos o vetor logo depois dele ter sido aumentado, o que será exibido na tela?

E essa versão funciona?


```
11  void aumentarVetor(int *v, int tamAtual, int novoTam)
12  {
13      int* novoV = criarVetor(novoTam);
14
15      for (int i = 0; i < tamAtual; i++) {
16          novoV[i] = v[i];
17      }
18
19      delete [] v;
20
21      v = novoV;
22  }
```

Repare que a diferença é que, ao invés de retornar o novo vetor ele está alterando o endereço do ponteiro para vetor passado por parâmetro.

E essa versão funciona?


```
11  void aumentarVetor(int *v, int tamAtual, int novoTam)
12  {
13      int* novoV = criarVetor(novoTam);
14
15      for (int i = 0; i < tamAtual; i++) {
16          novoV[i] = v[i];
17      }
18
19      delete [] v;
20
21      v = novoV;
22  }
```

```
53      aumentarVetor(vetor, n, novoTam);
```

Suponha que o subprograma seja chamado assim:

Quando a chamada do procedimento acontece, o parâmetro **v** recebe o mesmo valor do ponteiro **vetor**, ou seja, eles guardam o mesmo endereço (mas não são a mesma variável, portanto existem dois lugares na memória , **v** e **vetor**, que guardam o mesmo endereço).

Quando fazemos **v** apontar para **novoV**, estamos alterando o endereço que **v** guarda, mas **vetor** continua apontando para o mesmo endereço.

Mas **o pior** é que depois do procedimento ser chamado o ponteiro **vetor** estará apontando para um região que não está mais alocada (foi desalocada na linha 19)

Cuidados com a Alocação Dinâmica

- Uma série de cuidados devem ser tomados ao se manipular ponteiros e realizar alocações de memória.
- Estes cuidados visam evitar problemas decorrentes do uso indevido dos ponteiros.

- Problemas comuns:
  - Ponteiros soltos.
  - Variáveis dinâmicas perdidas.

Ponteiros Soltos

- Um ponteiro que contém o endereço de uma variável alocada dinamicamente, mas já liberada para outros usos.
- A posição de memória sendo apontada pode ter sido realocada para uma outra variável dinâmica;
- Se o ponteiro solto é usado para modificar o valor da posição de memória, o valor da nova variável será destruído;
- A liberação explícita de variáveis dinâmicas é a causa dos ponteiros soltos.

Ponteiros Soltos


```
int main(){
  int const TAM = 5;
  int *vetor1, *vetor2;
  int i;
  vetor1 = new int[TAM];
  vetor2 = vetor1;

  cout << "Digite os elementos do vetor1: ";
  for (i = 0; i < TAM; i++) {
      cin >> vetor1[i];
  }

  // (...) Computações em vetor1

  delete [] vetor1;

  // (...) Computações em vetor2

  return 0;
```

Note que agora vetor2 está solto, porque o armazenamento para o qual estava apontando foi liberado.

Variáveis Dinâmicas Perdidas

- Variável alocada dinamicamente que não está mais acessível para os programas do usuário.
- Variáveis dinâmicas perdidas são frequentemente chamadas de lixo, pois não são úteis para seus propósitos gerais e não podem ser realocadas para algum novo uso no programa;
- Usualmente são criadas pela seguinte sequência de ações:
  - Um ponteiro p1 é configurado para uma variável dinâmica recém-alocada;
  - p1 é posteriormente configurado para apontar para outra posição de memória.

Referências

- Slide baseado na aula de Ponteiros e Alocação Dinâmica do Professor Júlio César da UFLA.

Links para as aulas

- **Aula de AED2 - Realizada no dia 01/04/2020 - 19:00**

  https://drive.google.com/file/d/1jY1NL6TaigoFt9BGMkW1MqiOk_8Fnwi_/view?usp=sharing

- **Aula de LAB2 - Realizada no dia 02/04/2020 - 19:00**

  https://drive.google.com/file/d/1U3xywgzwyZgne8EgYTqFHNKT1d4P-8TD/view?usp=sharing

Algoritmos e Estrutura de Dados II

Aula 10

Alocação Dinâmica

Claudiane Maria Oliveira
claudianeo@gmail.com