INF 110
Programação I

Prof. Fabio R. Cerqueira
frcerqueira@gmail.com
DPI / UFV

Aula 15:
Alocação dinâmica em C++

Alocação dinâmica

- A linguagem C++ provê o mecanismo de **alocação dinâmica** (**em tempo de execução**), adicionalmente à alocação estática (em tempo de carga do código) que vimos até o momento.

- No caso da alocação dinâmica, fica sob responsabilidade do programador a alocação e também a liberação de memória.

- A alocação dinâmica é feita com o comando `new`.

- Para liberar memória alocada dinamicamente, utiliza-se o comando `delete`.

Alocação dinâmica

- Exemplo :

```
int* pi; // Alocação estática já conhecida.
float* pf; // Alocação estática já conhecida.
pi = new int; // Aloca din. área para int e faz
              // pi apontar para esta área.
pf = new float; // Aloca din. área para float e
                // faz pf apontar para esta área.
*pi = 1024; // Escreve em área alocada din.
*pf = 3.25; // Escreve em área alocada din.
cout<<"Um inteiro: "<<*pi<<" e um float: "<<*pf;
delete pi; // desaloca área apontada por pi
delete pf; // desaloca área apontada por pf
// Obs.: pi e pf continuam vivas aqui.
```

Alocação dinâmica

- Observações importantes:
  - Quando se faz: `new <tipo>` aloca-se memória suficiente para o tipo em questão e o endereço (do início) da área alocada é retornado. Se não houver memória disponível, é retornado o ponteiro `NULL`.
  - Quando se faz: `delete <ponteiro>` libera-se a área apontada pelo ponteiro. Não se está desalocando o ponteiro. Cuidado com isto!
  - O tempo de vida da área alocada será o trecho entre o `new` e o `delete`. Veja que na alocação estática o tempo de vida da variável vai até o fecha-chave do bloco onde a mesma foi declarada.

Alocação dinâmica – Arranjos dinâmicos

- Para se alocar arranjos dinamicamente, deve-se utilizar colchetes à frente do tipo, indicando o tamanho. Note ainda que para desalocar o arranjo há a necessidade de se acrescentar `[]` à frente da palavra `delete`, caso contrário, somente a primeira posição do arranjo será desalocada.

```
int* v;
v = new int[3];
v[0]=3;
v[1]=5;
v[2]=8;
...
delete [] v;
```

- Veja que ao invés de `v = new int[3]`, poderíamos fazer `v = new int[n]`, onde `n` seria um valor, por exemplo, digitado pelo usário.
- Outra observação importante é que `v` **não** é um ponteiro constante como no caso da alocação de arranjos estáticos.

Alocação dinâmica – Arranjos dinâmicos

- Exercício: seja a `struct Ponto` abaixo. Mostre como seria a alocação dinâmica de um arranjo do tipo abaixo para representar uma curva composta por 500 pontos?

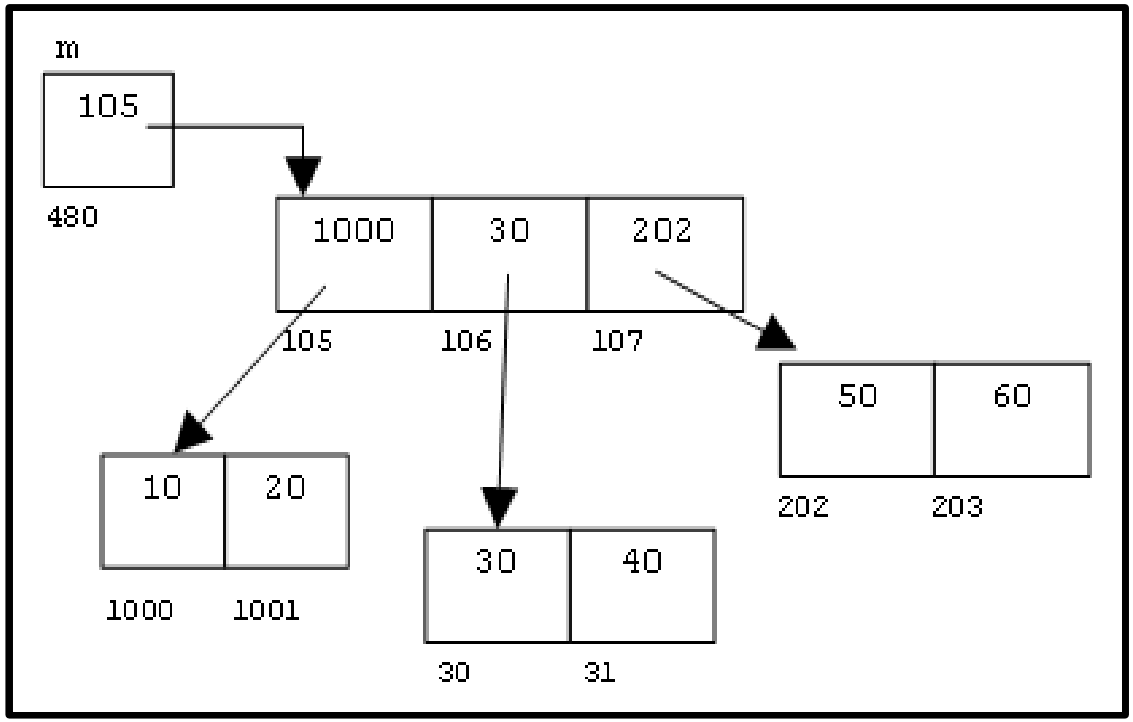
```
struct Ponto {
  float x, y;
};
```

Alocação dinâmica – Arranjos dinâmicos

- Para se alocar arranjos multidimensionais de forma dinâmica, aloca-se uma série de arranjos de ponteiros e também de arranjos do tipo com que se deseja trabalhar.

Alocação dinâmica – Arranjos dinâmicos

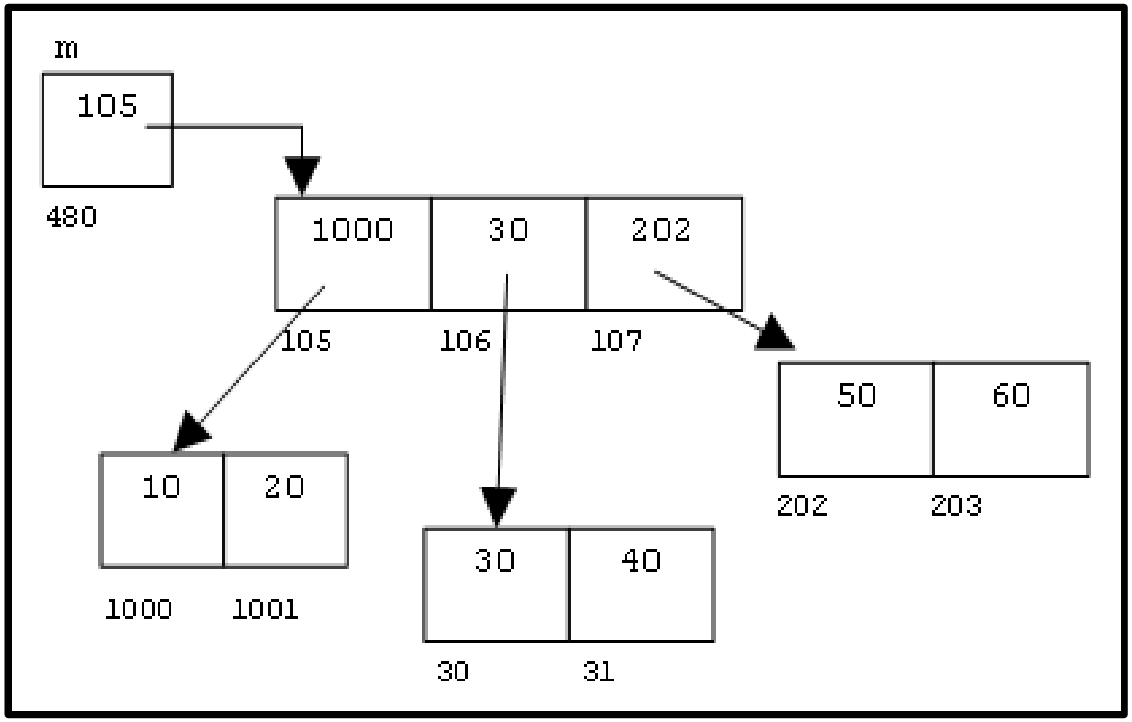
- Vamos exemplificar com matrizes (arranjos bidimensionais). A representação que normalmente se usa para alocação dinâmica de uma matriz é como ilustrado abaixo (imagine uma matriz `3x2` de `int` de nome `m`):



- Neste caso, foram armazenados: 10, 20, 30, 40, 50 e 60.

Alocação dinâmica – Arranjos dinâmicos

- Veja portanto que `m`, na verdade, representa um arranjo de tamanho 3 (o número de linhas) de `ponteiros para int`.
- Cada posição do arranjo representado por `m`, representa um arranjo de tamanho 2 (o número de colunas) de `int`.
- Portanto, estes arranjos representados por `m[0]`, `m[1]` e `m[2]` são onde os inteiros que se quer armazenar ficarão.



- Assim, qual o tipo de dado de `m`? R: `int **`

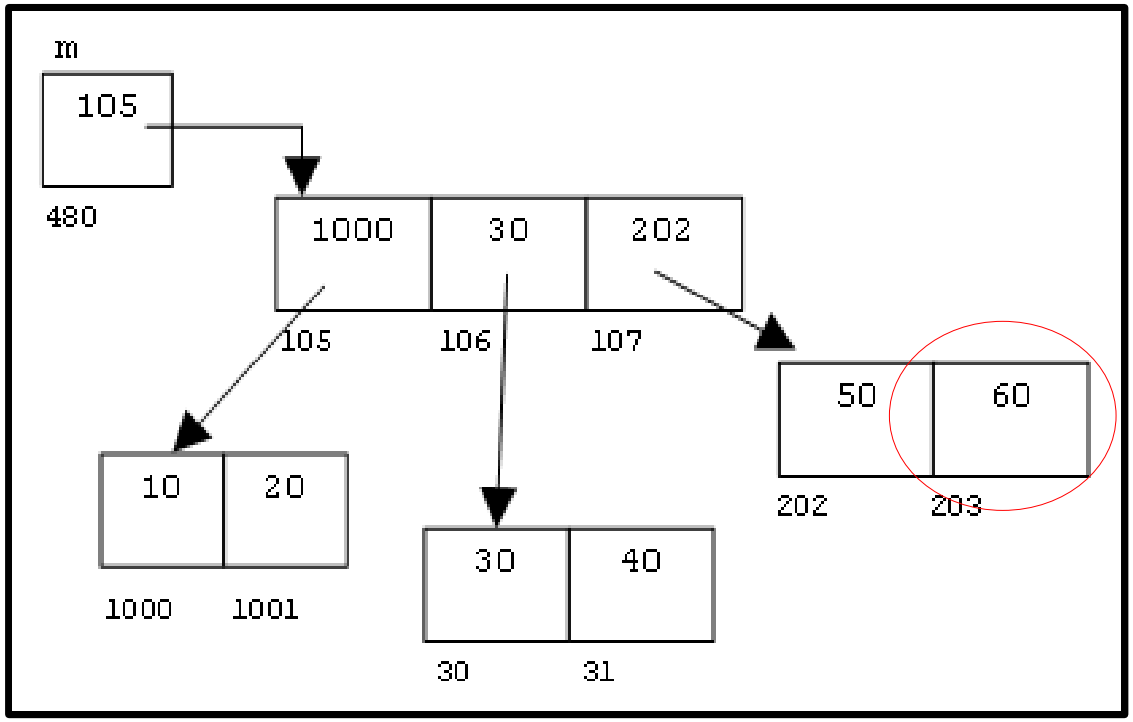
Alocação dinâmica – Arranjos dinâmicos

- Note que ao se fazer: `m[i][j]`, isto é equivalente a fazer:

```
*(*(m+i)+j)
```

- Portanto, ao se fazer: `m[2][1]=60`, isto é equivalente a fazer:

```
*(*(m+2)+1)=60
```



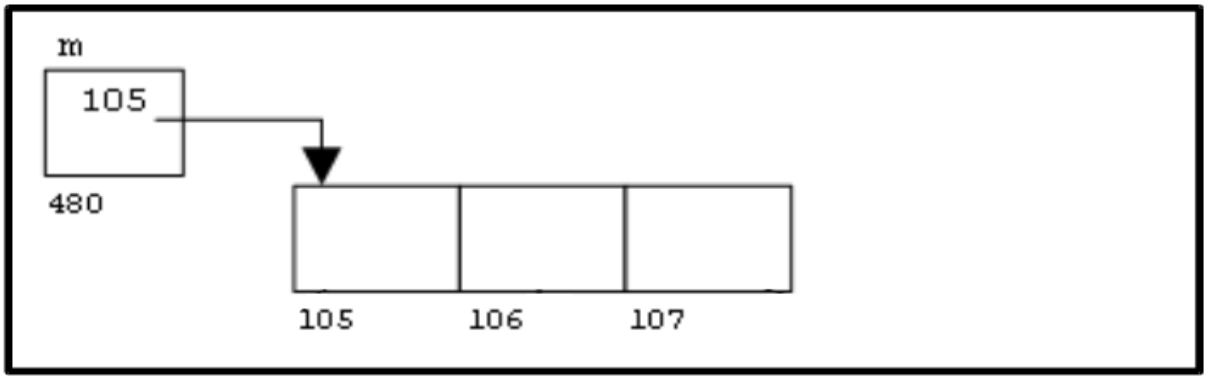
Alocação dinâmica – Arranjos dinâmicos

- Mas como fazer isto na prática?

- Primeiramente, temos que alocar um arranjo de ponteiros para o tipo em questão cujo tamanho será o número de linhas que se deseja para a matriz:

```
int ** m;
m = new int*[3];
```

- Obtendo:

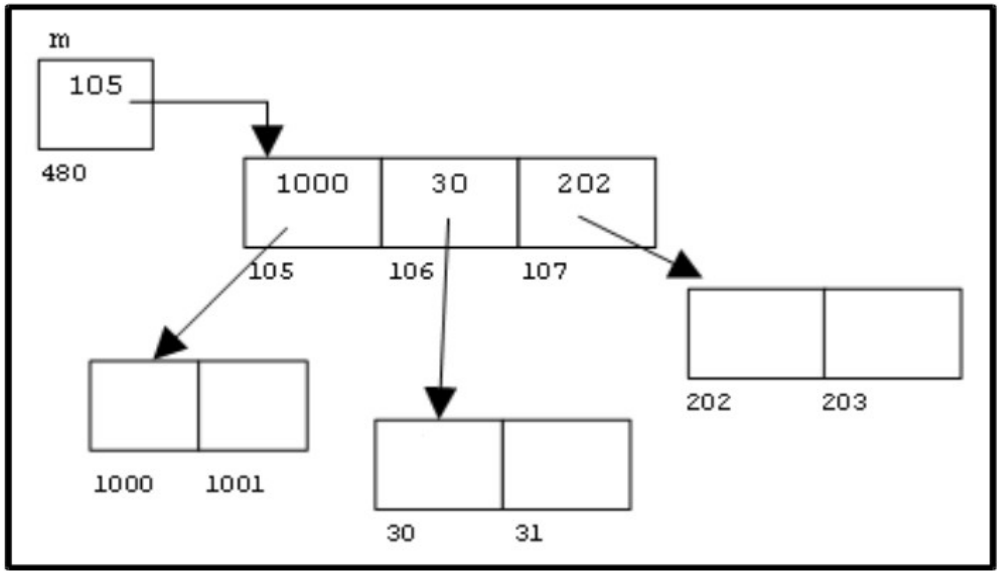


Alocação dinâmica – Arranjos dinâmicos

- Após isto, alocamos três arranjos do tipo `int` onde o tamanho de cada um será igual ao número de colunas da matriz. Estes arranjos devem ser representados pelos ponteiros `m[0]`, `m[1]` e `m[2]`:

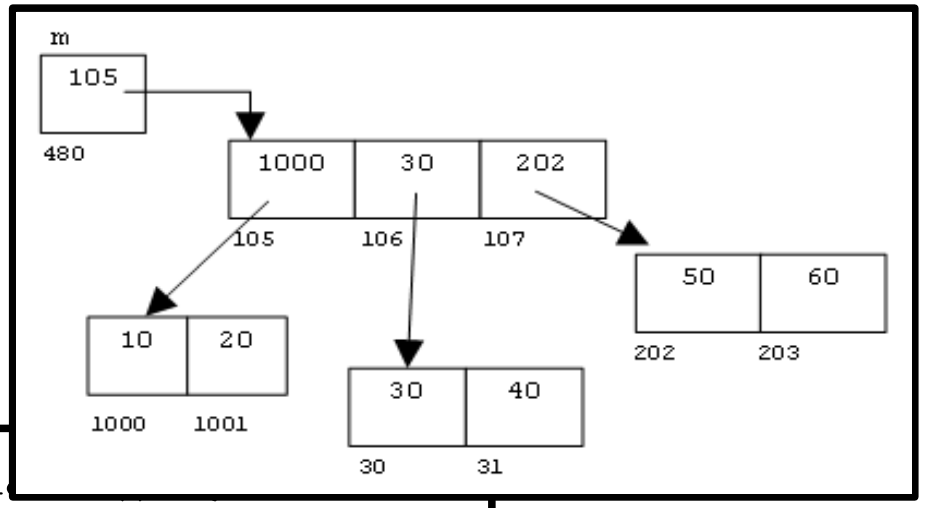
```
int ** m;
m = new int*[3];
//continuando
m[0] = new int[2];
m[1] = new int[2];
m[2] = new int[2];
```

- Obtendo:



Alocação dinâmica – Arranjos dinâmicos

- Para preencher a matriz, poderíamos utilizar os colchetes tradicionais ou substituí-los por operações sobre os ponteiros, como mostrado anteriormente:




```
int main() {
   ...
   m[0][0] = 10;
   m[0][1] = 20;
   m[1][0] = 30;
   m[1][1] = 40;
   m[2][0] = 50;
   m[2][1] = 60;
   ...
}
```

```
int m
   ...
   *(*(m+0)+0) = 10;
   *(*(m+0)+1) = 20;
   *(*(m+1)+0) = 30;
   *(*(m+1)+1) = 40;
   *(*(m+2)+0) = 50;
   *(*(m+2)+1) = 60;
   ...
}
```

Alocação dinâmica – Arranjos dinâmicos

- Note que todos os arranjos foram alocados dinamicamente. Isto significa que temos que desalocá-los com o `delete`.

- Cuidado, porém, com a ordem em que esta ação é realizada. Se o arranjo de ponteiros para `int` de tamanho 3 for desalocado primeiro, perder-se-á a referência para os outros arranjos e não será mais possível desalocá-los.

- Portanto, deve-se desalocar o arranjo de ponteiros por último.

Alocação dinâmica – Arranjos dinâmicos

- Assim, basta realizar a seguinte sequência de comandos para realizar a completa liberação da área ocupada pela matriz:

```
... // Uso da matriz.
// Quando m não for mais necessária:
delete [] m[0];
delete [] m[1];
delete [] m[2];
delete [] m;
```

- Obviamente que quando as dimensões da matriz forem suficientemente grandes, utilizaremos estruturas de repetição tanto na alocação quanto na liberação de memória.

Alocação dinâmica – Arranjos dinâmicos

- Uma última observação, ainda sobre matrizes, é que aquelas operações sobre ponteiros, equivalentes ao uso dos colchetes, também são utilizadas para matrizes estáticas .

- A diferença, mais significativa, é que matrizes estáticas têm os seus arranjos alocados de maneira contínua na memória, como se fossem um único arranjo, enquanto que as matrizes dinâmicas têm seus arranjos alocados em qualquer área da memória que tenha espaço disponível. Ou seja, um arranjo constituinte da matriz dinâmica não tem nada a ver, em termos de localização, com qualquer outro que também esteja compondo esta matriz.

- Por isto que numa função o parâmetro que representa uma matriz estática precisa da segunda dimensão na declaração, para saber onde começa e termina cada bloco representando uma linha.

- Já um parâmetro que represente uma matriz dinâmica pode ser declarado como ponteiro. Se matriz de `int`, por exemplo: `int**m`.

Alocação dinâmica – Arranjos dinâmicos

- Exercício: seja a `struct PixelRGB` abaixo. Mostre como seria a alocação dinâmica de uma matriz `500x500` do tipo abaixo para representar uma figura de 500x500 pixels com cores expressadas pelo padrão RGB?

```
struct PixelRGB {
  int r, g, b;
};
```

Alocação dinâmica – Erros comuns

- Quando se programa com alocação dinâmica, dois erros são comumente cometidos por programadores desatentos:
  - *Geração de Lixo*: quando, após a saída de um trecho do programa onde a memória alocada não é mais necessária, esta não é liberada com `delete`.
  - *Referência Danosa*: quando um ponteiro referencia uma região da memória que não está alocada para o programa (ou foi alocada mas já foi liberada com `delete`).

Alocação dinâmica – Erros comuns

- Exemplo 1 - Geração de Lixo: programador esquece de desalocar a memória alocada.

```
void f() {
  int* x;
  x = new int;
  *x = 100;
      . . .
  // Suponha que não houve: delete x.
}
```

Alocação dinâmica – Erros comuns

- Exemplo 2 - Geração de Lixo: outra forma de deixar resíduos na memória.

```
void f() {
  int* x;
  x = new int;
  *x = 100;
  . . .
  x = new int; // Perde ref. da 1a área
  . . .
// A primeira região da memória ficou
// sem referência (não é mais possível
// desalocá-la). O ponteiro x passou
// a referenciar a segunda região
// de memória alocada.
}
```

Alocação dinâmica – Erros comuns

- Exemplo 3: Referência danosa.

```
void f() {
  int* x;
  int* y;
  x = new int;
  *x = 100;
  y = x; // Os 2 ponteiros referenciam
         // a mesma região de memória.
  delete x;  // A área é desalocada.
  *y = 500; // ERRO! Pois y apontava
            // para a mesma região
            // (já desalocada) que x.
  . . .
}
```

Alocação dinâmica

- Para finalizar, a gerência de memória dos computadores atuais disponibiliza para os programas três tipos de área de memória para armazenamento de dados, memórias estas utilizadas de diferentes modos.

- As três áreas são denominadas de:
  - área **estática** (*static*);
  - área de **pilha** (*stack*) ou **automática** e
  - área de **alocação livre** (*heap*).

Alocação dinâmica

- A área estática é alocada em tempo de carga do código. Nela são armazenados os dados globais e variáveis declaradas com a palavra-chave `static`.

- A área de pilha também é alocada em tempo de carga. Nela são armazenados os parâmetros para as funções e variáveis locais (e outras informações relativas às funções). Esta área é também chamada de memória automática, porque a alocação e liberação são realizadas automaticamente no início e no fim, respectivamente, do bloco de execução em questão. Seu uso é do tipo *last-in-first-out*, permitindo maior desempenho se comparada com a memória *heap*.

Alocação dinâmica

- A área de *heap* fica disponível para alocação dinâmica, ou seja, em tempo de execução. Nela o programador pode alocar e liberar áreas de memória para armazenar valores primitivos, de arranjos e de tipos compostos por meio dos operadores `new` e `delete`.

- O programador deve ficar atento para liberar a memória quando não precisa mais, pois, caso contrário, o programa pode esgotar a memória disponível. Um programa que não libera a memória que não está mais utilizando e continua alocando mais memória possui um problema denominado de vazamento de memória (*memory leakage*).