Alocação dinâmica de memória em C++

A alocação dinâmica permite solicitar espaço de memória em tempo de execução. Digamos que no início do programa você não sabe o tamanho do vetor que vai precisar. Se alocar um vetor muito grande estará desperdiçando memória RAM. O ideal é ir solicitando memória à medida que seu programa vai precisando.

A linguagem C++ suporta todas as funções de alocação de memória definidos na linguagem C. Porém, ela possui operadores específicos de alocação dinâmica e liberação de memória alocada. Em C, a alocação é feita com a função malloc (e, para liberar memória, usa-se a função free). Em C++, faz-se o mesmo tipo de alocação usando o operador `new`. Quando não quiser mais utilizar a memória, use o operador `delete` .

O operador `new` retorna o endereço onde começa o bloco de memória que foi reservado. Como retorna um endereço podemos colocá-lo num ponteiro. Assim, teremos um meio de manipular o conteúdo da memória alocada toda vez que mencionarmos o ponteiro.

Teste o código abaixo:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
   int *ptr_a;
   ptr_a = new int;
   // cria a área necessária para 01 inteiro e
   // coloca em 'ptr_a' o endereço desta área.

   if (ptr_a == nullptr){
      cout << "Memória insuficiente!" << endl;
      exit(1);
   }
   cout << "Endereço de ptr_a: " <<  ptr_a << endl;
   *ptr_a = 90;
   cout << "Conteúdo de ptr_a: " << *ptr_a << endl;

   delete ptr_a;
   return 0;
}
```

Na alocação dinâmica temos de nos certificar de que a alocação no heap foi feita com sucesso. Uma das maneiras de fazer isso é usando a opção **std::nothrow**. No caso de não se conseguir a memória, a opção nothrow retorna um ponteiro nulo, e o programa continua:

```
int *v = new (std::nothrow) int [5];
```

Alocação dinâmica de vetores

No exemplo anterior, foi alocado um espaço para armazenar apenas um dado do tipo int. Entretanto, é mais comum utilizar ponteiros para alocação de vetores.

Para tanto, basta especificar o tamanho desse vetor no momento da alocação. Nos exemplos abaixo, apresenta-se a alocação de vetores com o operador `new[]`.

Quando não quiser mais utilizar a memória alocada, use o operador `delete[]`.

Após a alocação de uma área com vários elementos, ela pode ser acessada exatamente como se fosse um vetor.

```
#include <iostream>
#include <cstdlib>

int main()
{
   // Aloca memória para vetor com 3 inteiros.
   int *v = new (std::nothrow) int[3];

   // Encerra programa se não conseguir alocar memória.
   if (v == nullptr) {
      std::cout << "Erro: não foi possível alocar memória.";
      return 1;
   }

   // Acessamos elementos como em um vetor.
   v[0] = 10;
   v[1] = 20;
   v[2] = 30;

   // Acessa os valores armazenados na memória alocada.
   cout << v[0] << " " << v[1] << " " << v[2] << "\n";

   // Libera memória alocada.
   delete[] v;

   return 0;
}
```

Alocação dinâmica de estrutura

```
#include <iostream>
#include <cstdlib>
#include <cstring>

struct pessoa {
   char nome[50];
   int idade;
};

int main()
{
   // Aloca memória para todos os campos da estrutura 'pessoa'.
   pessoa *p = new (std::nothrow) pessoa;

   // Encerra programa se não conseguir alocar memória.
   if (p == nullptr) {
      std::cout << "Erro: não foi possível alocar memória.";
      return 1;
   }

   // Acesso aos campos da estrutura.
   strcpy(p->nome, "Joao da Silva");
   p->idade = 20;
   std::cout << p->nome << " " << p->idade << std::endl;

   // Libera memória alocada.
   delete p;

   return 0;
}
```

Alocação dinâmica de vetor de estruturas

```
#include <iostream>
#include <cstdlib>
#include <cstring>

struct pessoa {
   char nome[50];
   int idade;
};

int main()
{
   // Aloca memória vetor com 2 elementos da estrutura 'pessoa'.
   pessoa *v = new (std::nothrow) pessoa[2];

   // Encerra programa se não conseguir alocar memória.
   if (v == nullptr) {
      std::cout << "Erro: não foi possível alocar memória.";
      return 1;
   }

   // Acesso aos campos da estrutura.
   strcpy(v[0].nome, "Joao da Silva");
   v[0].idade = 20;
   strcpy(v[1].nome, "Maria dos Santos");
   v[1].idade = 25;

   std::cout << v[0].nome << " " << v[0].idade << std::endl;
   std::cout << v[1].nome << " " << v[1].idade << std::endl;

   // Libera memória alocada.
   delete[] v;

   return 0;
}
```

Vazamento de memória

Quando temos um ponteiro que não tem em seu conteúdo um valor de memória válido, o qual pertence ao programa sendo executado, temos um problema grave, principalmente se isso ocorreu por engano. Isso pode ser um problema difícil de identificar, principalmente se o programa já está com um certo número de linhas considerável. Vejamos um exemplo:

```
#include <iostream>
using namespace std;

int main()
{
   int *myPointer;
   myPointer = new int;
   *myPointer = 10;
   cout << "O valor de myPointer eh " << *myPointer << endl;
   delete myPointer;
   *myPointer = 5; //Acesso indevido a uma área não identificada.(Lembre-se que o ponteiro perdeu o valor válido de memória que apontava antes do '''delete'''. Agora a área de memória não pertence mais ao programa.)
   cout << "O valor de myPointer e " << *myPointer << endl;

   return 0;
}
```

Neste exemplo liberamos a memória dinâmica, mas o ponteiro continua a existir. Isto é um "bug", e muito difícil de detectar. Acontece que se essa memória for acessada ou escrita será corrompida.

A melhor maneira de evitar isso é, depois do **delete**, fazer o ponteiro apontar para **nullptr**, fazê-lo um ponteiro nulo. Depois disso se tentarem usar o ponteiro iremos ter a "run time exception" e o "bug" poderá ser identificado.

Assim, corrigindo o código anterior ficaríamos com:

```
#include <iostream>
using namespace std;

int main()
{
   int *myPointer;
   myPointer = new int;
   *myPointer = 10;
   cout << "O valor de myPointer eh " << *myPointer << endl;
   delete myPointer;
   myPointer = nullptr;
   *myPointer = 5; //Essa instrução vai causar uma "run-time exception", agora.
   cout << "O valor de myPointer e " << *myPointer << endl;

   return 0;
}
```

Última atualização: sexta, 11 dez 2020, 20:46

◄ Ponteiros e Passagem por referência

Seguir para...

Exercícios - Ponteiros e alocação dinâmica ►