Smart Pointers

1. Introdução

Segundo (Tucker & Noonam, 2010) na linguagem C++, o gerenciamento da memória alocada dinamicamente fica a cargo do programador. Atividade então, que segundo (Horstmann, 2005) está propensa a erros. O problema está em “esquecer” a desalocação da memória dos recursos que não estão sendo mais utilizados pelo programa, o que em longos períodos de execução, podem levar ao esgotamento da memória disponível do sistema. Logo, de acordo com (Tucker & Noonam, 2010), “O gerenciamento em tempo de execução da memória dinâmica é uma atividade necessária para linguagens modernas de programação. Durante muito tempo, a automação dessa atividade tem sido um elemento- chave na implementação de linguagens de programação”.

Na linguagem C++, existem os Smart Pointers (Ponteiros Inteligentes, em tradução), recurso proposto para realizar um gerenciamento mais adequado da memória alocada dinamicamente, a qual garante a desalocação de recursos que não estão mais sendo utilizados pelo programa. (Rao, 2012).

2. O que é Ponteiro

Para explicar a variável Ponteiro, eis uma comparação: Uma variável do tipo int, armazena como valor, um número inteiro. Já a variável Ponteiro, armazena como valor, o endereço de memória de outra variável. Portanto, o valor de um ponteiro indica onde uma variável está armazenada, e não o que ela está armazenando. (Deitel & Deitel, 2006).

Ao armazenar o endereço de memória de outra variável, entende-se que o ponteiro “aponta” para esta variável, tal qual, contém um tipo de dado especifico, como exemplo, um valor do tipo int. Nesse caso, diz-se que o ponteiro aponta para um valor int. (Mizrahi, 1994).

Ainda, de acordo com (Deitel & Deitel, 2006), ao apontar para uma variável, o ponteiro referencia o valor da variável apontada indiretamente. Através de um operador especifico para ponteiro, podemos acessar/modificar o valor da variável apontada. Portanto, com ponteiros, podemos acessar diferentes regiões de memórias, que por sua vez, armazenam diferentes valores, os quais também através dos ponteiros podemos acessar/modificar.

3.1. Utilidades:

Nas duas seções seguintes serão descritas algumas das utilidades do uso de ponteiro.

3.1.1. Alocação dinâmica

Com ponteiros, torna-se possível utilizar o recurso de Alocação Dinâmica, no qual o programa aloca ou desaloca memória em tempo de execução, conforme necessidade. Isso torna o consumo de memória do programa mais eficiente, já que o espaço de memória podeser ocupado conforme necessidade, sem reserva antecipada da mesma. Isso resolve o problema de dimensionamento de espaço da alocação estática, onde reservava-se espaços superestimados para não haver problemas com falta de memória. (Medina & Fertig, 2006).

3.1.2. Estrutura de Dados

Permite a criação e manipulação de estruturas de dados complexas, como listas encadeadas e árvores binárias, onde um item deve conter referência a outro. Estas estruturas também podem crescer ou encolher, ocupando espaço sob demanda. (Mizrahi, 1994) (Medina & Fertig, 2006).

3.2. Problemas

Segundo (Deitel & Deitel, 2006), alocação e acesso dinâmico a memória são vulneráveis a falhas de programas. Portanto, a má manipulação de Ponteiros, podem ocasionar sérios problemas para o funcionamento do programa e do sistema. São exemplos:

- Acesso (ilegal) de áreas desconhecidas de memória, que podem estar sendo utilizadas por outros programas ou pelo sistema, devido a erros de programação.
- Vazamento de memória (memory leak): memória alocada e não liberada.

Vazamentos de memória são perigosos, pois não causam falhas imediatas e podem esgotar a memória do sistema após longos períodos de execução. Portanto, muitas linguagens de programação modernas empregam desalocação automática de memória, para prevenir ponteiros perdidos. (Pezzè & Young, 2008). Na linguagem C++ existem os Smart Pointers (Ponteiros Inteligentes, em tradução), que se propõe a realizar um gerenciamento mais adequado da memória, garantindo a liberação adequada dos dados alocados dinamicamente. (Rao, 2012).

4. Smart Pointers (Ponteiros Inteligentes)

De forma simples, um ponteiro inteligente se comporta como um ponteiro convencional. No entanto, ele fornece recursos adicionais que garantem a desalocação dos dados alocados dinamicamente, deixando o “ciclo de vida” de um objeto bem definido. (Rao, 2012).

Segundo (Gregoire, 2014), ponteiros inteligentes é um recurso recomendado para evitar vazamentos de memória (memory leak). É uma noção que surgiu a partir de como as variáveis são tratadas na Stack, onde são automaticamente “destruídas” quando saem do escopo. Portanto, ponteiros inteligentes, combinam o gerenciamento de variáveis da Stack, com a flexibilidade de variáveis da Heap, onde espaços de memória são alocados em tempo de execução (alocação dinâmica). Seguem abaixo, características e exemplo de uso de dois ponteiros inteligentes da linguagem C++, o ponteiro unique_ptr e o ponteiro shared_ptr.

4.1. unique_ptr.

Segundo (Ricarte, 2014) o ponteiro inteligente unique_ptr tem as seguintes características:

- Ponteiro não pode ser copiado
- Posse do ponteiro pode ser transferida
    - Conteúdo é movido e área anterior passa a ser inválida

Segue abaixo, exemplo de código com uso do ponteiro unique_ptr:

```
/*
* ATIVIDADE 3 - SI300/A
* DEMONSTRAÇÃO DE USO DO PONTEIRO INTELIGENTE UNIQUE_PTR
* UNICAMP - FACULDADE DE TECNOLOGIA
* 2016
*/
#include <cstdlib>
#include <iostream>
#include <vector>
#include <memory>
```

```
using namespace std;
class Inteiros{
public:
   void setNumero();
   void getNumero();
   ~Inteiros();
private:
   vector <unique_ptr<int>> inteiros;
};

Inteiros::~Inteiros(){
   cout << endl;
   cout << "Limpando lista" << endl;
   /*
   * COM USO DO PONTEIRO INTELIGENTE UNIQUE_PTR, TORNA-SE
   * DESNECESSÁRIO AS LINHAS DE CÓDIGO COMENTADAS ABAIXO
   * QUE LIBERAM AS OS ESPAÇOS DE MEMÓRIAS ALOCADAS DINAMICAMENTE.
   * COM O USO DO PONTEIRO UNIQUE_PTR, OS ESPAÇOS SERÃO LIBERADOS
   * AUTOMATICAMENTE QUANDO NÃO FOREM MAIS USADOS (SAIREM FORA DO ESCOPO).
   *
   while (scan != inteiros.end()){
   delete (*scan);
   (*scan) = NULL;
   scan++;
   }
   */
   inteiros.clear();
   cout << "Finalizando" << endl;
}

void Inteiros::getNumero(){
   vector<unique_ptr<int>>::iterator scan = inteiros.begin();
   cout << endl << "Imprimindo vetor:" << endl;
   while (scan != inteiros.end()){
      cout << **scan << endl;
      scan++;
   }
}

void Inteiros::setNumero(){
   for (int i = 1; i < 5; i++){
      unique_ptr <int> x(new int);
      cout << "Digite o " << i << "o número" << endl;
      cin >> *x;
      if (*x != 0){
         inteiros.insert(inteiros.end(), (move(x)));
         /*
         * OBSERVE QUE FOI USADO O COMANDO "move" PARA A OPERAÇÃO DE INSERÇÃO
         * ACIMA. ISTO PORQUE, O PONTEIRO UNIQUE_PTR NÃO PERMITE CÓPIA.
         *
         * OBSERVE TAMBÉM, O PROBLEMA QUANDO É INSERIDO UM NÚMERO IGUAL A ZERO.
         * UM ESPAÇO DE MEMÓRIA FOI ALOCADO, PORÉM, NÃO FOI INSERIDO NO VETOR
         * FICANDO TAMBÉM, SEM REFERÊNCIA DE ACESSO. ESTE ESPAÇO DE MEMÓRIA,
         * FICARIA ALOCADO E SEM USO DURANTE TODA A EXECUÇÃO DO PROGRAMA.
         * ESTE É UM CASO DE VAZAMENTO DE MEMÓRIA (memory leak).
         * DEVIDO AO USO DO UNIQUE_PTR, ESSES ESPAÇOS SERÃO LIBERADOS ASSIM QUE
         * ESTIVEREM FORA DO ESCOPO.
         */
      }
      else{
         cout << "Número zero não permitido. Tente outro número." << endl;
         i--;
      }
```

```
    }
}

int main(int argc, char** argv){
    Inteiros numeros;
    cout << "Digite 4 números" << endl;
    numeros.setNumero();
    numeros.getNumero();
    return 0;
}
```

4.2.shared_ptr

Segundo (Ricarte, 2014), o ponteiro inteligente shared_ptr tem as seguintes características:

- Ponteiro para um recurso que pode ser compartilhado
- Com controle do número de referencias
- Quando última referência deixa de existir, recurso é liberado.

Segue abaixo, exemplo de código com uso do ponteiro shared_ptr:

```
/*
* ATIVIDADE 3 - SI300/A
* DEMONSTRAÇÃO DE USO DO SHARED_PTR
* UNICAMP - FACULDADE DE TECNOLOGIA* 2016
*/

#include <cstdlib>
#include <iostream>
#include <vector>
#include <memory>

using namespace std;

class Inteiros{
public:
    vector <shared_ptr<int>> setNumero();
    /*
    * PONTEIRO SHARED_PTR PERMITE CÓPIA E PODE SER USADO COMO PARÂMETRO
    * PARA FUNÇÕES; SEGUE ABAIXO:
    */
    void getNumero(vector <shared_ptr<int>>);
    void Clear(vector <shared_ptr<int>>);
};
void Inteiros::Clear(vector <shared_ptr<int>> inteiros){
    cout << endl;
    cout << "Limpando lista" << endl;
    //PARA AS LINHAS DE CÓDIGO ABAXIO, VALE A MESMA OBSERVAÇÃO FEITA NOS
    //COMENTARIOS DO CÓDIGO DE EXEMPLO DE USO DO unique_ptr
    /*while (scan != inteiros.end())
    {
    delete (*scan);
    (*scan) = NULL;
    scan++;
    }*/
    inteiros.clear();
    cout << "Finalizando" << endl;
}
```

```
void Inteiros::getNumero(vector <shared_ptr<int>> inteiros){
   vector<shared_ptr<int>>::iterator scan = inteiros.begin();
   cout << endl << "Imprimindo vetor:" << endl;
   while (scan != inteiros.end()){
      cout << **scan << endl;
      scan++;
   }
}
vector <shared_ptr<int>> Inteiros::setNumero(){
   vector <shared_ptr<int>> inteiros;
   for (int i = 1; i < 5; i++){
      shared_ptr <int> x(new int);
      cout << "Digite o " << i << "o número" << endl;
      cin >> *x;
      if (*x != 0){
         inteiros.insert(inteiros.end(), x); //PONTEIRO SHARED_PTR PERMITE CÓPIA. NÃO NECESSITA move
      }
      else{
         cout << "Número zero não permitido. Tente outro número." << endl;
         i--;
      }
   }
   return inteiros;
}
int main(int argc, char** argv){
   Inteiros numeros;
   vector <shared_ptr<int>> inteiros;
   cout << "Digite 4 números" << endl;
   inteiros = numeros.setNumero();
   numeros.getNumero(inteiros); //RECURSO COMPARTILHADO
   numeros.Clear(inteiros); //RECURSO COMPARTILHADO
   return 0;
}
```

5. Considerações Finais

Vimos que, o ponteiro comum, chamado de ponteiro bruto, é um recurso essencial para o desenvolvimento de estruturas de dados e para o gerenciamento de memória em tempo de execução por exemplo, recursos essenciais para praticamente qualquer programa. Mas, em contrapartida, expõe o programador a erros que podem ser fatais. Portanto, toda automação por parte das linguagens de programação, que ofereça maior robustez e segurança para o uso de ponteiros, é bem-vinda. Esse é o caso, dos ponteiros inteligentes (smart pointers) em C++, que fornecem recursos automáticos principalmente para a alocação dinâmica, tornando o programa eficiente com o gerenciamento de memória.

6. Referências Bibliográficas

Deitel, H., & Deitel, P. (2006). C++ : como programar. São Paulo: Pearson Prentice Hall.

Gregoire, M. (2014). Professional C++. Indianapolis: Wrox.

Horstmann, C. (2005). Conceitos Essencias de Computação com o Essencial de C++. Porto Alegre: Bookman.

Medina, M., & Fertig, C. (2006). Algoritmos e Programação : Teoria e Prática. São Paulo: Novatec.

Mizrahi, V. V. (1994). Treinamento em Linguagem C++. São Paulo: Makron Books.

Pezzè, M., & Young, M. (2008). Teste e Análise de Software. Porto Alegre: Bookman.

Rao, S. (2012). Sams Teach Yourself C++ In One Hour a Day. Sams.

Ricarte, I. (18 de 05 de 2014). Slides de Aula. Um novo paradigma para o ensino de ponteiros frente à evolução de C++. Acesso em 02 de 06 de 2016, disponível em SlideShare: http://pt.slideshare.net/ivanricarte/ricarte-ft2014

Tucker, A., & Noonam, R. (2010). Linguagens de Programação - Princípios e Paradigmas. Porto Alegre: AMGH.