Em qualquer linguagem de programação (ou na sua grande maioria) uma variável só é acessada dentro do escopo em que foi declarada ou se for declarada em um escopo dito "global".

Para que uma variável seja acessível em outro escopo, e passível de ser alterada, deverá haver, no escopo interessado, um ponteiro para esta variável, ou seja, uma variável que contém o endereço da variável "remota".

Algumas linguagens resolvem de forma automática a criação dos ponteiros, mas outras, como a C e C++, não resolvem, ficando a cargo do programador criar esses ponteiros.

Vamos supor um número do tipo int e um método para o qual será enviado:

**int x = 10;**

**calculo(x);**

Por ser do tipo primitivo, será criada uma cópia no método calculo, ou seja, será uma variável local lá que não tem nada a ver com a de cá (do método chamador).

Se o método calculo alterar o valor de x, nele denominado x\_remoto, em nada altera o valor de x.

**void calculo(int x\_remoto){**

**x\_remoto = x\_remoto--;**

**}**

Para que a alteração em x no método calculo também se reflita na variável x do método chamador é necessário um ponteiro para o x original.

Há mais de uma forma de se fazer isso:

* **1)**

**int x = 10;**

**int \*px;**

**px = &x;**

**calculo(px);**

E o método calculo deve estar na seguinte forma:

**void calculo(int \* x\_remoto){**

**\* x\_remoto = \* x\_remoto--;**

**}**

* **2)** O método chamador não se preocupa com o ponteiro, quem faz tudo é o método chamado.

**int x = 10;**

**calculo(x);**

**calculo(int &x){**

**x\_remoto = x\_remoto--;**

**}**

Obs: esta forma oferece alguma insegurança, pois um programador mal intencionado poderia alterar o método chamado para alterar indevidamente a variável remota, uma vez que o método chamador não tem nenhum controle.

Para evitar que a sua variável local seja alterada inadvertidamente, é necessário enviar uma constante. Se a própria já é uma constante, envia a própria, caso contrário, ou seja, ainda será alterada ao longo do algoritmo, declara-se uma constante de igual valor.

**int x = 10;**

**int const y = x;**

**calculo(y);**

Caso o programador insira o operador de endereços (&) no parâmetro x, ocorrerá erro na próxima compilação.

Resumindo:

1. **int x = 10;**
2. **int \* px = & x;**
3. **cout << px << endl;**
4. **cout << \* px << endl;**
5. **cout << & px << endl;**

A linha 3 imprime o conteúdo do ponteiro, ou seja, o endereço da variável que ele está apontando (x).

A linha 4 imprime o conteúdo de x (10).

A linha 5 imprime o endereço do ponteiro.

Se fizermos:

**px += 1;**

O conteúdo de px será incrementado em tantos bytes quantos os do *data type* que ele aponta (tipo int = 4 bytes).

================================================

**ALOCAÇÃO DINÂMICA DE MEMÓRIA**

A todo programa que é inicializado, é criado um processo. Esses processos são identificados por números inteiros, os PID (process identifier).

Por uma interface de administrador de sistema é possível encerrar (matar) um processo informando o seu número. No Unix, por exemplo, com a senha do root, aplica-se o comando KILL #, onde # = número do processo.

O processo recebe uma área de memória em que pode trabalhar onde se tem um segmento para código e um segmento para dados. Uma área estática, com as variáveis de tamanho já pré-definidos e também variáveis de tamanho dinâmico, inicializadas pelo operador **new**. Se essas variáveis, como os arrays, excederem esse espaço, ocorrerá um erro de *runtime*.

Uma forma de contornar isso é solicitar memória adicional através da instrução **malloc**.

O operador **new** funciona com objetos já definidos, ou seja, aloca a memória que o objeto necessita, não cabendo ao programador especificar a quantidade de bytes necessária.

Já a instrução **malloc** é de mais baixo nível, alocando a quantidade de bytes que se deseja.

1. long long int v1[100000];
2. long long int x;
3. cin >> x;
4. long long int \* v2 = new long long int[x];
5. long long int \* v3 = (long long int\*)malloc(x \* sizeof(long long int));

Na linha 1 foi criado um vetor estático (de tamanho já definido no código fonte), v1, para 100.000 elementos. Esse valor poderá não ser aceito em tempo de execução em função do conjunto de componentes do algoritmo.

As linhas 2 e 3 pedem um valor para dimensionar dinamicamente dois novos vetores.

A linha 4 instancia o vetor v2 com o tamanho especificado em x. Apesar de a alocação ser dinâmica, ainda se limita ao seu espaço de memória para dados.

Já a linha 5 utiliza a instrução **malloc** que amplia o espaço original teoricamente sem limites.

**Operações com matrizes:**

Vetores e matrizes são ponteiros por natureza, onde a variável aponta para o primeiro elemento e para acessar os demais é questão de se deslocar pela memória.

Um elemento de um vetor pode ser acessado por v[i] ou por \*(v+i).

Alocação estática: int m[10][5];

Alocação dinâmica:

**int\*\* alocarMatriz(int Linhas,int Colunas){** //Recebe a quantidade de Linhas e Colunas como Parâmetro

**int i,j;** //Variáveis Auxiliares

//Aloca um Vetor de Ponteiros

**int \*\*m = (int\*\*)malloc(Linhas \* sizeof(int\*));**

//Percorre as linhas do Vetor de Ponteiros

**for (i = 0; i < Linhas; i++){**

**//**Aloca um Vetor de Inteiros para cada posição do Vetor de Ponteiros.

**m[i] = (int\*) malloc(Colunas \* sizeof(int));**

**//**Percorre o Vetor de Inteiros atual e inicializa com 0.

//Este for é opcional.

**for (j = 0; j < Colunas; j++){**

**m[i][j] = 0; //Inicializa com 0.**

**}**

**}**

**return m; //**Retorna o Ponteiro para a Matriz Alocada

**}**

<https://programacaodescomplicada.wordpress.com/2012/11/09/aula-64-alocacao-dinamica-pt-6-alocacao-de-matrizes/>

**Computador de 4 bits**

**SET DE INSTRUÇÕES**

|  |  |
| --- | --- |
| 0000 | STOP (pára a execução) |
| 0100 | Lê o conteúdo da memória que está no endereço contido na posição seguinte e carrega no Registrador 1. |
| 0110 | Lê o conteúdo da memória que está no endereço contido na posição seguinte e carrega no Registrador 2. |
| 0111 | Soma o registrador 1 com o registrador 2 e armazena o resultado no endereço contido na posição seguinte |

**ÁREA DO PROGRAMA**

|  |  |  |
| --- | --- | --- |
| **Endereço** | **Conteúdo** | **Explicação** |
| 0000 | 0100 | Lê o conteúdo da memória que está no endereço contido na posição seguinte (endereço 0001 contém 1100) e carrega no Registrador 1. |
| 0001 | 1100 | O endereço 1100 (área de dados) contém 0101 => Registrador 1 passa a conter 0101. |
| 0010 | 0110 | Lê o conteúdo da memória que está no endereço contido na posição seguinte (endereço 0011 contém 1101) e carrega no Registrador 2. |
| 0011 | 1101 | O endereço 1101 (área de dados) contém 0111 => Registrador 2 passa a conter 0111. |
| 0100 | 0111 | Soma R1 com R2 e armazena o resultado no próximo endereço de memória (endereço 0101 contém 1110) |
| 0101 | 1110 | R1 + R2 = 0101 + 0111 = 1100 => o endereço 1110 passará a conter 1100 |
| 0110 | 0000 | Pára a execução (fim de programa) |

**ÁREA DE DADOS**

|  |  |  |
| --- | --- | --- |
| **Endereço** | **Conteúdo** | **Explicação** |
| 1100 | 0101 | O endereço 1100 (decimal 12) contém 0101 (decimal 5) |
| 1101 | 0111 | O endereço 1101 (decimal 13) contém 0111 (decimal 7) |
| 1110 | 0000 | O endereço 1110 (decimal 14) contém 0000 (zero)  Após a execução do programa anterior o conteúdo passará a 1100 |
| 1111 | 0000 | O endereço 1111 (decimal 15) contém 0000 (zero) |

0000 - programa

0001 - programa

0010 - programa

0011 - programa

0100 - programa

0101 - programa

0110 - programa

0111 **- livre**

1000 **- livre**

1001 **- livre**

1010 **- livre**

1011 **- livre**

1100 - dados

1101 - dados

1110 - dados

1111 - dados