1.Declarações e inicializações de variáveis do tipo ponteiro I

Os ponteiros são variáveis que contêm endereços de memória como valores. Normalmente, uma variável faz uma referência direta a um valor específico. Um ponteiro, por outro lado, contém um endereço de uma variável que contém um valor específico. Um nome de variável faz uma referência direta a um valor, e um ponteiro faz referência indireta a um valor. Diz-se que fazer referência a um valor por meio de um ponteiro é fazer uma referência indireta. Os ponteiros devem ser declarados antes de serem usados

```
int *nomePonteiro;
```

A declarada variável ***nomePonteiro** do tipo ***int*** (um ponteiro para um valor inteiro) e é lida como **nomePonteiro** é um ponteiro para ***int*** ou aponta para um objeto do tipo inteiro.

O asterisco se aplica somente a **nomePonteiro** na declaração. Quando o asterisco é usado dessa forma em uma declaração, ele indica que a variável que está sendo declarada é um ponteiro. Os ponteiros podem ser declarados para apontar objetos de qualquer tipo de dados.

Os ponteiros devem ser inicializados ao serem declarados ou em uma instrução de atribuição. Um ponteiro pode ser inicializado com 0, NULL ou um endereço. Um ponteiro com o valor NULL não aponta para lugar algum, onde NULL é uma constante simbólica definida no arquivo de cabeçalho <stdio.h>. Inicializar um ponteiro com 0 é equivalente a inicializar um ponteiro com NULL, mas NULL é mais recomendado. Quando o valor 0 é atribuído, inicialmente ele é convertido em um ponteiro do tipo apropriado. O valor 0 é o único valor inteiro que pode ser atribuído a uma variável de ponteiro.

2.Declarações e inicializações de variáveis do tipo ponteiro II

O **&** ou operador de ponteiro é um operador unário que retorna o endereço de seu operando.

```
int y = 5;
int *yPtr;
int yPtrt = &y;
```

A instrução **yPtr = &y;** atribui a variável ***y*** na variável de ponteiro **yPtr**. Diz-se que a variável ***yPtr*** aponta para ***y***. O operando do operador de endereço deve ser uma variável e o operador de endereço não pode ser aplicado a constantes, expressões ou a variáveis declaradas com a classe de armazenamento ***register***. O operador asterisco, chamado frequentemente operador de frequência indireta ou operador de desreferenciamento, retorna o valor do objeto ao qual seu operando aponta.

3.Utilizando o qualificador *const* com ponteiros

O qualificador ***const*** permite ao programador informar ao compilador que o valor de uma determinada variável não deve ser modificado. Existem 6 possibilidades de usar ***const*** com parâmetros de funções: duas com a passagem de parâmetros por meio de uma chamada por

valor e 4 por meio de uma chamada por referência. Sempre ceda a uma função acesso suficiente aos dados em seus parâmetros para realizar sua tarefa específica, não mais.

Todas as chamadas em C são por valor, onde é feita uma cópia do argumento da função e esta cópia é passada à função. Se a cópia for modificada na função, o valor original é mantido sem modificação no local que executa a chamada. Em muitos casos, um valor passado a uma função é modificado para que a função possa realizar sua tarefa. Entretanto, algumas vezes, o valor não deve ser alterado na função chamada, muito embora a função manipule uma cópia do valor original.

Agora, considere uma função que utiliza um ***array*** unidimensional e seu tamanho como argumentos e imprima o ***array***. Tal função deve fazer um loop através do ***array*** e enviar para impressão cada elemento do ***array*** individualmente. O tamanho do ***array*** é usado no corpo da função para determinar o maior subscrito do ***array*** de forma que o loop possa terminar quando a impressão for concluída. O tamanho do ***array*** não se modifica no corpo da função.

Se for feita uma tentativa de modificar um valor declarado ***const***, o compilador reconhece e emite um aviso ou um erro, dependendo do compilador em particular.

Há 4 maneiras de passar um ponteiro para uma função: um ponteiro não-constante para um dado não-constante; um ponteiro consiste para um dado não-constante; um ponteiro não-constante para um dado constante e um ponteiro constante para um dado constante. Cada uma das 4 combinações fornece um nível diferente de privilégios de acesso.

O maior nível de acesso aos dados é concedido por um ponteiro não-constante para um dado não-constante. Nesse caso, o dado pode ser modificado por meio de um ponteiro desreferenciado, e o ponteiro pode ser modificado de modo a apontar para outros itens de dados. Uma declaração de um ponteiro não-constante para um dado não-constante não inclui ***const***. Tal ponteiro pode ser usado para receber uma ***string*** como um argumento de uma função que use a aritmética dos ponteiros para processar cada caractere da ***string***.

Um ponteiro não-constante para um dado constante é um ponteiro que pode ser modificado para apontar para qualquer item de dado do tipo apropriado, mas o dado ao qual ele aponta não pode ser modificado. Tal ponteiro pode ser usado para receber um argumento ***array*** em uma função que processará cada elemento do ***array*** sem modificar os dados.

Quando uma função é chamada tendo um ***array*** como argumento, o ***array*** é passado automaticamente para a função por meio de uma chamada por referência. Entretanto, as ***structs*** são sempre passadas por meio de uma chamada por valor (uma cópia da “struct”). Isso exige o “overhead” em tempo de execução de fazer uma cópia de cada item dos dados na ***struct*** e armazená-lo na pilha de chamada da função no computador. Quando a ***struct*** de dados deve ser passada a uma função, podemos usar ponteiros para dados constantes para obter o desempenho de uma chamada por referência com a proteção de uma chamada por

valor. Quando é passado um ponteiro a uma estrutura, deve ser feita apenas uma cópia do endereço no qual a estrutura está armazenada. Em um equipamento com endereços de 4 bytes, é feito uma cópia de 4 bytes de memória em vez de uma cópia de possivelmente centenas ou milhares de bytes de ***struct***.

Usar ponteiros para dados constantes desta maneira é um exemplo de compensação de tempo/espaço. Se a memória estiver pequena e a eficiência da execução for a principal preocupação, devem ser usados ponteiros. Se houver memória em abundância e a eficiência não for a preocupação principal, os dados devem ser passados por meio de chamadas por valor para impor o princípio do privilégio mínimo.

Um ponteiro constante para um dado não-constante é um ponteiro que sempre aponta para o mesmo local da memória, e os dados naquele local podem ser modificados por meio do ponteiro. Esse é o default para um nome de ***array***. Um nome de ***array*** é um ponteiro constante para o início do ***array***. Pode-se ter acesso a todos os dados do ***array*** e modificá-los usando o nome do ***array*** e os subscritos. Um ponteiro constante para dados não-constantes pode ser usado para receber um ***array*** como argumento de uma função que tenha acesso aos elementos do ***array*** usando apenas a notação de seus subscritos. Os ponteiros declarados como ***const*** devem ser inicializados ao serem declarados.

O privilégio de acesso mínimo é garantido por um ponteiro constante para um dado constante. Tal ponteiro sempre aponta para o mesmo local da memória, e os dados nesse local da memória não podem ser modificados. E assim que um ***array*** deve ser passado a uma função que apenas verifica o ***array*** usando a notação de subscritos e não modifica o ***array***.

O operador ***sizeof*** pode ser aplicado a qualquer nome de variável, tipo ou constante. Ao ser aplicado a um nome de variável ou constante, é retornado o número de bytes usado para armazenar o tipo específico de variável ou constante.

4.Expressões de ponteiros e aritmética de ponteiros I

Os ponteiros são operandos válidos em expressões aritméticas, expressões de atribuições e expressões de comparação. Apesar disto, nem todos os operadores usados normalmente com essas expressões são válidos ao estarem junto de variáveis de ponteiros.

Um conjunto limitado de operações aritméticas pode ser realizado com ponteiros. Um ponteiro pode ser incrementado ou decrementado, pode ser adicionado um inteiro a um ponteiro, um inteiro pode ser subtraído de um ponteiro ou um ponteiro pode ser subtraído de outro.

Admita:

int v[10]

Admita que o ponteiro ***vPtr*** foi declarado e inicializado para apontar ***v[0]***, i.e., o valor de ***vPtr*** é 3000. Observe que ***vPtr*** pode ser inicializado para apontar para o ***array v*** com qualquer uma das seguintes instruções.

```
vPtr = v;
vPtr = &v[10]
```

Quando um inteiro é adicionado ou subtraído de um ponteiro, este último não é simplesmente incrementado ou decrementado por tal inteiro, mas sim por tal inteiro vezes o tamanho do objeto ao qual o ponteiro se refere.

```
vPtr += 2
```

Ao realizar este incremento, se produz 3008 (3000 + 2 * 4) admitindo que um inteiro está armazenado em 4 bytes de memória. No ***array v***, ***vPtr*** apontaria agora ***v[2]***. Se um inteiro for armazenado em 2 bytes de memória, o cálculo anterior resultaria na posição de memória 3004 (3000 + 2 * 2). Ao realizar a aritmética de ponteiros em um ***array*** regular de caracteres, os resultado serão consistentes com a aritmética regular porque cada caractere tem o comprimento de 1 byte.

Qualquer uma das instruções ***++vPtr*** ou ***vPtr++*** incrementa o ponteiro para apontar para a próxima posição no ***array***. Qualquer uma das instruções ***--vPtr*** ou ***vPtr--*** decrementa o ponteiro para apontar para a posição anterior do "array".

As variáveis de ponteiros podem ser subtraídas entre si. Se ***vPtr*** possui o local 3000 e ***v2Ptr*** possuir o endereço 3008, a instrução ***x = v2Ptr - vPtr*** atribuiria à x o número de elementos de ***array*** de ***vPtr*** a ***v2Ptr***, neste caso 2. Não podemos assumir que duas variáveis do mesmo tipo estejam armazenadas contiguamente na memória a menos que sejam elementos adjacentes de um ***array***.

Um ponteiro pode ser atribuído a outro ponteiro se ambos forem do mesmo tipo, caso contrário, deve ser utilizado um operador de conversão para transformar o ponteiro à direita da atribuição para o tipo do ponteiro à esquerda da mesma. A exceção a essa regra é o ponteiro para ***void***, que é um ponteiro genérico que pode representar qualquer tipo de ponteiro. Todos os ponteiros podem ser atribuídos a um ponteiro ***void***, e um ponteiro ***void*** pode ser atribuído a um ponteiro de qualquer tipo. Não é exigido um operador de conversão. Um ponteiro "void" não pode ser desreferenciado.

Os ponteiros podem ser comparados por meio de operadores de igualdade e relacionais, mas tais comparações não significam nada se os ponteiros não apontarem para membros do mesmo ***array***. Comparações de ponteiros comparam os endereços armazenados nos ponteiros. Um uso comum da comparação de ponteiros é para determinar se um ponteiro é "NULL".

5.Expressões e aritmética de ponteiros II

Os ***arrays*** e os ponteiros estão intimamente relacionados em C e um ou outro podem ser usados quase indiferentemente. Pode-se imaginar que o nome de um "array" é um ponteiro constante. Os ponteiros podem ser usados para fazer qualquer operação que envolva um subscrito de "array".

Admita:

int b[5] (I)
***bPtr (II)**
bPtr = b (III)

A instrução (III) é equivalente a tomar o endereço do primeiro elemento do ***array*** como se segue ***bPtr = &b[0]***

Três na expressão anterior é o deslocamento (offset) do ponteiro. Quando um ponteiro aponta para o início de um ***array***, o deslocamento indica que elemento do ***array*** deve ser referenciado, e o valor do offset é idêntico ao subscrito do ***array***. Os parênteses são necessários porque a precedência do asterisco é maior que a precedência do "+". Sem os parênteses, a expressão adicionaria 3 ao valor da expressão ****bPtr***.

O ***array*** em si pode ser tratado como um ponteiro e usado com a aritmética de ponteiros. Por exemplo, a expressão ****(b+3)*** também se refere ao elemento do ***array b[3]***. Em geral, todas as expressões de ***arrays*** com subscritos podem ser escritas com um ponteiro e um offset.

Os ponteiros podem possuir subscritos exatamente da mesma forma que os ***arrays***, por exemplo, na expressão ***bPtr[1]*** se refere ao elemento ***b[1]***. Isso é chamado notação subscrito.

6.Arrays de ponteiros

Os ***arrays*** podem conter ponteiros. Um uso comum de tais estruturas de dados é para formar um ***arrays de strings***. Cada elemento do ***array*** é uma ***string***, mas em C, uma ***string*** é essencialmente um ponteiro para o seu primeiro caractere. Assim, cada elemento de um ***array*** de ***strings*** é na verdade um ponteiro para o primeiro caractere de uma ***string***.

Admita:

char *naipe[4] = {"Copas", "Ouros", "Paus", "Espadas"};

A parte ***naipe[4]*** da declaração indica um ***array*** de quatro elementos. A parte ***char **** da declaração indica que cada elemento do ***array naipe*** é do tipo ***ponteiro a char***. Os quatro valores a serem colocados no ***array*** são "Copas", "Ouros", "Paus" e "Espadas". Cada um deles é armazenado na memória como uma ***string*** de caracteres terminada em NULL que tem o comprimento de um caractere a mais do que o número de caracteres entre aspas. As

quatro ***strings*** possuem comprimento 6, 6, 5, e 8 caracteres respectivamente. Embora pareça que essas ***strings*** estão sendo colocadas no ***array naipe***, apenas os ponteiros são de fato colocados no ***array***. Cada ponteiro aponta para o primeiro caractere de sua ***string*** correspondente. Dessa forma, mesmo que o ***array naipe*** tenha seu tamanho definido, o ponteiro fornece acesso a ***strings*** com qualquer quantidade de caracteres. Essa flexibilidade é um exemplo dos poderosos recursos de estruturação de dados.

Os naipes podem ser colocados em um ***array*** bidimensional no qual cada linha iria representar um naipe e cada coluna iria representar uma letra do nome de um naipe. Tal estrutura de dados precisaria ter um número fixo de colunas por linha, e esse número precisaria ter comprimento igual ao da maior ***string***, logo, uma quantidade considerável de memória poderia ser desperdiçada quando um grande número de strings fosse armazenado com muitas ***strings*** menores do que a maior delas.

7.Ponteiros para funções

Um ponteiro para uma função contém o endereço da função na memória. Sabendo que o nome de um ***array*** é realmente o endereço do primeiro elementos do ***array*** na memória, o nome de uma função é na realidade o endereço inicial, na memória, do código que realiza a tarefa da função. Os ponteiros para funções podem ser passados a funções, retornados de funções, armazenados em "arrays" e atribuímos a outros ponteiros de funções.

8.Chamando funções por referência

Há duas maneiras de passar argumentos a uma função: chamada por valor e chamada por referência. Todas as chamadas de funções em C são chamadas por valor. O comando ***return*** pode ser usado para retornar um valor de uma função chamada para o local que a chamou. Muitas funções exigem a capacidade de modificar uma ou mais variáveis no local chamador, ou passar um ponteiro para um grande objeto de dados para evitar o "overhead" de passar o objeto chamado por valor. Para essa finalidade, a linguagem C fornece a capacidade de simular chamadas por referência.

Em C, os programadores usam ponteiros e o operador de referência indireta para simular chamadas por referência. Ao chamar uma função com argumentos que devem ser modificados, são passados os endereços dos argumentos. Isso é realizado normalmente aplicando o operador de endereço **&** à variável cujo valor será modificado.

Os ***arrays*** não são passados usando operador **&** porque a linguagem C passa automaticamente o local inicial do ***array*** na memória. Quando o endereço de uma variável é passado a uma função, o operador de referência indireta asterisco pode ser usado na função para modificar o valor no local de memória do chamador.