Apontadores

Em C há mecanismos especiais para lidar com os endereços de variáveis. Esses meanismos são ativados através do uso de variáveis especiais, chamdas de **apontadores** ("pointers"). O domínio deste conceito em C é muito importante, pois C é uma linguagem que permite ao programador manipular diretamente estes endereços. Manipulando estes endereços o programador pode construir estruturas de dados sofisticadas, além de requisitar e liberar memória dinamicamente durante a execução do programa. Ademais, as noções de apontadores e vetores, incluindo cadeias, estão intimamente ligadas em C.

Declaração de apontadores

Um apontador em C nada mais é do que uma variável comum, em cujo conteúdo armazenamos **o endereço de memória** de um outro objeto. O objeto cujo endereço é armazenado pode ser de qualquer tipo, como, por exemplo, uma outra variável, um vetor, ou mesmo uma função. É como se a variável "apontasse" para o objeto cujo endereço está ali armazenado (daí o nome).

Um apontador, sendo uma variável, deve também ser declarado. Também devemos declarar o tipo do objeto cujo endereço o apontador armazena. Uma vez declarado, o apontador só deve conter endereços de objetos desse tipo. Conhecendo o tipo do objeto cujo endereço o apontador armazena vai nos permitir escrever certos tipos de operações, ditas de aritmética com apontdores, e cujos resultados dependem do tipo do objeto. Assim, é preciso conhecer o tipo do objeto para o qual o apontador aponta. Examinaremos esses operadores mais adiante.
Para declarar uma variável de tipo apontador, iniciamos a declaração, como sempre, indicando um tipo. Este será o tipo dos objetos para os quais a variável poderá legalmente apontar. Logo em seguida, declaramos o nome da variável que será o apontador. Porém, para indicar que essa variável *é um apontador* (e não uma variável comum do tipo declarado) precedemos seu nome pelo símbolo `*` na declaração.

Por exemplo, as declarações

```
int *p;
float * f;
```

introduzem duas variáveis: `p` e `f`. Ambas são apontadores, como vemos pelo símbolo `*` que está aposto aos nomes das variáveis. Não é obrigatório encostar o operador `*` no nome da variável que está sendo declarada. Mas, para ênfase, esta prática é muito adotada em C.
Note que `p` é um apontador para objetos de tipo `int`, e `f` é um apontador para objetos de tipo `float`. Siginifca que a primaira variável só deve armazenar o endereço de variáveis de tipo `int`, enquanto que a segunda deve armazenar apenas o endereço de variáveis de tipo `float`. Se isto não for obervado, podemos obter resultados inesperados. Usualmente, o compilador emite um aviso se os tipos forem trocados (mas pode ir em frente, confiando no programador, seguindo a máxima de C: *sempre confie no programador*.)

Atribuindo para variáveis de tipo apontador

Apontadores armazenam endereços de outros objetos. Portanto, deve haver uma maneira de se obter endereços de objetos em C. Para isso, usamos o operador `&`. Quando aplicado sobre uma variável, o operador **`&`** devolve o endereço na memória dessa variável.
Por exemplo, considere

```
float  x=-1.0f;
float  *p;
p = &x;
```

Nas duas primeiras linhas, a variável `x` é declarada do tipo `float` (e tems eu conteúdo inicializado como `-1.0`) e a variável `p` é declarada um *apontador* para tipos `float`. Portanto, `p` pode armazenar o *endereço de memória* de uma variável que *contém* um valor tipo `float`.
Assuma que, numa certa execução desse trecho de código, o endereço em memória da variável `x` seja `500`. Na linha 3 este endereço será o valor *retornado* pelo operador `&` e que será armazenado em `p` com a execução do comando de atribuição. Veja que à variável `p` está sendo atribuído um endereço válido, uma vez que a variável `x` é do tipo `float`, foi declarada, e, portanto, deve ter um endereço de memória válido atribuído a si em tempo de execução.

Poderíamos (mas provavelmente não deveríamos) também escrever algo como

```
p = 0xaaa;
```

Neste caso, estamos atribuindo a `p` o valor `2730` (o mesmo que `aaa` em hexadecimal). O compilador deveria emitir um aviso neste caso. Note que o endereço armazenado em `p` provavelmente é inválido, no sentido de que neste endereço não encontraremos nenhuma das variáveis declaradas no programa. Os endereços de memória de variáveis só são conhecidos em tempo de execução. Mais ainda, estes endereços não são absolutos, no sentido de que, a cada execução, seriam sempre os mesmos. Num ambiente onde há múltiplas tarefas (processos) executando ao mesmo tempo, memória é alocada e liberada quando estas tarefas iniciam e terminam. Como o programador não tem controle sobre quais outras tarefas estarão executando em um dado instante, não pode saber quais endereços de memória estarão livres naquele instante para que posssam ser associados às suas variáveis. Esta é uma tarefa para o sistema operacional, que controla a tabela de alocação de memória.
O compilador, quando gera o código objeto, associa a este uma lista de variáveis para as quais se deve alocar memória *antes* da execução do programa começar e, junto a cada variável, gera uma lista de locais onde esta variável é referenciada no código objeto (compilado). Logo antes do início da excução um outro programa do sistema (o carregador, ou *loader*) se encarrega de, usando a lista de variáveis criada pelo compilador, interagir com o sistema operacional e bloquear endereços e espaços de memória para as variáveis que foram declaradas no programa. Quando o programa é carregado na memória para executar, estes endereços (físicos) de memória são também carregados pelo *loader* no locais indicados no código objeto para cada variável. Para tal, o *loader* usa a lista, criada pelo compilador, de locais que estão associados a cada variável no código compilado. Assim, quando o programa executa (obviamente) estará usando endereços físicos de memória válidos associados a cada

uma de suas variáveis. Não confunda este mecanismo com a *memória virtual*, que é apenas uma maneira do sistema operacional, com a ajuda de espaço em disco, articialmente estender o espaço de endereços de memória disponíveis além da capacidade física da memória principal do computador.
Podemos (mas provavelmente não devemos), também atribuir valores negativos à variável `p`.
Considere o seguinte trecho de código:

```
int i=1;               /* inteiros */
double x=5.0;          /* fracionario */
int *p, *p2;           /* apontador para inteiro */
double *f, *f1, *f2;   /* apontador para fracionario */
```

Uma variável inteira, uma fracionária, dois apontadores para `int` e três apontadores para `double`.

```
printf("Tamanho de um int: %1d bytes, e de um double = %1d bytes\n",
        sizeof(int),sizeof(double));
printf("Tamanho de um apontador para ints: %1d bytes, e para doubles:
%1d bytes\n\n",sizeof(int *),sizeof(double *));
```

Só para relembrar, `sizeof` informa o tamanho, em bytes, do objeto passado como parâmetro. No primeiro `printf` temos um `int` e um `double`. No segundo `printf` temos um *apontador* para um `int` e um *apontador* para um `double`. A execução revela

```
Tamanho de um int: 4 bytes, e de um double = 8 bytes
Tamanho de um apontador para ints: 4 bytes, e para doubles: 4 bytes
```

Portanto, o tamanho de variáveis tipo apontador é sempre 4 bytes, independente do tipo para o qual apontam. Isso porque apontadores carregam apenas endereços e, numa máquina de 32 bits, endereços têm sempre 32 bits, ou 4 bytes. Continuando,

```
p=&i; p2=p+1;
f=&x; f1=f+1; f2=f-2;
```

A variável `p` recebe o endereço da variável `i`. Repare na compatibilidade de tipos. O segundo comando mostra uma forma comum de lidarmos com apontadores em C. Neste comando, estamos adicionando uma unidade ao apontador `p` e atribuindo esse valor ao apontador `p2`. Como o conteúdo de `p` é um endereço para tipo de `int`, a expressão `p+1` produz o endereço onde estaria localizado o *próximo* `int` na memória do computador. Na arquittura da minha máquina, cada `int` ocupa 4 bytes. Logo, a expressão `p+1` deve apontar para 4 bytes adiante de onde aponta `p`. Ou seja, a expressão resulta no conteúdo de `p` mais 4. Este resultado é armazenado em `p2`. Note que `p2` também é um apontador para `int`. O mesmo se passa com as expresões envolvendo os apontadores para `double`. Na minha máquina, cada double ocupa 8 bytes. Logo, o valor de `f1` aponta para um endereço de memória 8 posições adiante do endereço de `f`, e o valor de `f2` aponta para 16 posições de memória antes do endereço de `f`. Continuando

```
printf("Conteudo de p = %p, de p2 = %p\n",p,p2);
printf("Conteudo de f = %p, de f1 = %p e de f2 = %p\n\n",f,f1,f2);
```

Imprimimos os valores dos apontadores. Note o modificador `%p`. Este modificador se aplica a resultados de expressões que denotem endereços de memória. Os valores impresos na saída estarão codificados na *base hexadecimal*. O resultado impresso, numa rodada na minha máquina foi

```
Conteudo de p = 8f324, de p2 = 8f328
Conteudo de f = 8f318, de f1 = 8f320 e de f2 = 8f308
```

confirmando a expectativa de que o computador tenta alocar `p` e `p2`, como também `f`, `f1` e `f2`, em endereços contíguos (não esqueça de fazer a aritmética na base 16 !).

```
p=10;
p2=(int *)0xabcd;
printf("Depois de p=10, o conteudo de p = %1d\n",(int)p);
printf("Depois de p2=(int *)0xabcd, o conteudo de p2 = %1p\n", p2);
```

Atribuindo endereços arbitrários (uma prática ***perigosa***). Na primeira atribuição, a *constante inteira* `10` é armazenada em `p`, uma variável de tipo `int *`. Esta incompatibiliadde de tipos provoca um aviso do compilador que, mesmo assim, gera o código para a atribuição e a executa. Portanto, após a atribuição, a variável `p` aponta para o endereço de memória de número 10, que provavelmente não está alocado para este programa. Na terceira linha, note a conversão de tipos, ou "cast", `(int)p`, ou seja, `p` é um *apontador* e estamos convertendo `p` para um `int`.
Na segunda artibuição, usamos um "cast", convertendo a constande `0xabcd` em um apontador para inteiro, antes da atribuição.
Na próxima invocação da função `printf` usamos o "cast" de novo, uma vez que estamos pedindo para que o valor de um apontador, de tipo `int *`, seja impresso como um valor decimal (como pede o modificador `%d`). Neste caso, estamos convertendo um endereço de memória para um valor inteiro, e o "cast", então, deve ser da forma `(int)`. A segunda invocação da função imprime o valor de `p2` diretamente (como pede o modificador `%p`). A impressão do resultado, numa rodada revela

```
Depois de p=10, o conteudo de p = 10
Depois de p2=(int *)0xabcd, o conteudo de p2 = abcd
```

confirmando a discussão.

Obtendo o conteúdo do endereço apontado

Quando queremos obter o *conteúdo* de uma variável apontada por uma outra variável, usamos o mesmo operador `*`. Veja o trecho de código abaixo:

```
float  x=-1.0, y=10.0; float *p = &x;
y = (* p) + 5.0f;
```

Na linha de declaração, o tipo básico é `float`. Como as variáveis `x` e `y` não apresentam modificadores, são entendidas como variáveis que conterão objetos deste tipo. A variável `p`, por outro lado, apresenta o modificador `*`, o que faz com que seja entendida como um *apontador* para objetos *que contêm* objetos tipo `float`. Note, que, na declaração, podemos usar a expressão `*p=&f` pois neste ponto o nome `x` já é

conhecido como designando uma variável de tipo `float`, e, portanto, já tem um endereço designado que pode ser obtido pelo emprego do operador `&`.
Para entender a atribuição e o uso do operador `*`, vamos assumir que o endereço atribuído a `x` seja `500`. Note que `p` foi então inicializada com o valor `500`. Na atribuição, o valor retornado pela expressão `(* p)` não é o valor armazenado na variável `p`. Isto seria `500`, ou seja, seria o *endereço* da variável `x`. A expressão `(*p)` retorna, isto sim, o *conteúdo* da posição de memória *cujo endereço* está armazenado em `p`. Portanto, `(*p)` deve retornar o conteúdo armazenado na posição de memória cujo endereço é `500`, ou seja, retorna o valor armazenado em `x`, que é `-1.0`. E este é justamente o valor armazenado na variável para a qual `p` aponta. O espaço entre o símbolo `*` e o nome da variável `p` não é obrigatório, nem os parênteses envolvendo a expressão `(* p)` são obrigatórios. Eles estão lá por clareza. Os operadores `&` e `*` são unários e têm precedência sobre operadores binários.
O resto da expressão é calculado normalmente, e o valor final, `-4.0`, é armazenado na variável `y`.
Nesse mesmo exemplo, se omitirmos o símbolo `*`, a atribuição ficaria na forma

```
y = p + 5.0f;
```

e obteríamos um erro de compilação por usarmos tipos incompatíveis na adição, um *apontador para* `float` e um `float`.
Poderíamos (mas talvez não devêssemos) atribuir um valor para `p` e, em seguida, usar o valor que está naquele endereço para calcular uma expressão. Por exemplo,

```
p = 0xaaa;
y = (float)(* p)+5.0f;
```

Duas situações podem ocorrer. Por uma coincidência fortuita, o endereço `0xaaa` é válido, isto é, estava alocado para esse programa nesse momento; o computador recolhe o que estava neste endereço, transforma este valor para um `float` (por causa do "cast" para `float`), adiciona `5.0f` e armazena o resultado na variável `y`. Ou então o endereço `0xaaa` não está alocado para esse programa nesse momento, isto é, é inválido (como seria com certeza se fizéssemos `p = -0xaaa`), e aí obteríamos um erro de execução, pois o computador não teria como acessar o endereço indicado e seu conteúdo para calcular o valor da expressão. Em qualquer hipótese, um aviso de compilação seria provavelmente emitido, alertando que a atribuição `p = 0xaaa` pode causar problemas.

Armazenando no endereço apontado

Como fazer para armazenar um valor num endereço apontado por uma variável de tipo apontador?
Usamos o mesmo operador **`*`**, porém agora junto da variável que é atribuída, ou seja, no *lado esquerdo* da atribuição. Usado junto ao nome de um *apontador* do *lado esquerdo* de uma atribuição, o operador `*` devolve o conteúdo *do apontador*, ou seja, devolve o endereço da variável apontada. É neste endereço que vamos armazenar o valor que foi calculado para a expressão que ocorre à direita da atribuição.
Assim,

```
(*p) = . . .
```

resultará em que usaremos o *conteúdo* do apontador `p` como o *endereço* onde armazenar o valor calculado. A linguagem C exige que a variável `p` seja um apontador, provocando erro de compilação se `p` for uma variável comum (embora esta pudesse conter um valor que fosse um endereço de memória válido). Considere o trecho de código:

```
float  x=-1.0; float *p=&x;
(*p) = 5.0f;
```

Vamos assumir que nessa execução o endereço atribuído a `x` seja `500`. Portanto, o apontador `p` conterá `500`. O valor da expressão à direita, na linha 2, é `5.0f`. Este valor deverá ser armazenado na variável cujo endereço que está contido em `p`, ou seja, na varfiável cujo e endereço é `500`. Mas `500` é o endereço da variável `x`, pois inicializamos `p` atribuindo-lhe o endereço de `x`, na forma `&x`. Portanto, o efeito da atribuição será armazenar o resultado `5.0f` na variável `x`.
Valem as mesmas observações colocadas acima, se atribuímos um valor arbitrário para o apontador `p` e, em seguida, tentamos usar `(*p)` à esquerda de uma atribuição. Se, por acaso, o endereço que atribuirmos resultar válido, o comando é executado, armazenando o valor calculado para a expressão à direita da atribuição `=`, no endereço que atribuímos a `p`. Se o endereço não for válido, obtermos um erro durante a execução do programa.
Vamos agora examinar outro trecho de código.

```
int i=10; float x=3.1415f;
int *p, *q;
float *f, *e;
```

Declaramos e inicializamos `i` e `x`; declaramos `p` e `q` como apontadores para inteiros; declaramos `f` e também `e` como apontadores para `float`.

```
 printf("Ender i = %p, ender x = %p\n",&i,&x);
 printf("Ender p = %p, ender q = %p, ender f = %p, ender e = %p\n", &p, &q,
&f, &e);
```

Informa os endereços de toda as variáveis envolvidas. Primeiro informa os endereços das variáveis numéricas. Na próxima linha, informa os endereços dos apontadores. Estes endereços são atribuídos logo antes do início da execução do programa, e não devem mudar durante a execução do código. Continuando,

```
printf("Val p = %p, val q = %p, val f = %p, val e = %p\n",p,q,f,e);
 printf("Val i = %d, x = %f\n",i,x);
```

Informa os valores contidos nas variáveis. Note que os apontadores ainda *não foram inicializados* e, portanto, agora ainda contêm valores arbitrários. Alguns compiladores inicializam variáveis declaradas com um valor padrão (por exemplo, zero). Continuando,

```
p=&i; f=&x;
*f = sqrt(*p)+1.0;
*p = (int)(*f) - 3;
```

```
printf("Val p = %p, val f = %p\n",p,f);
printf("Val i = %d, val x = %f\n",i,x);
```

Operações típicas com apontadores. Primeiro, os apontadores são carregados com o endereço de certas variáveis. No caso, `p` conterá o endereço de `i` e `f` conterá o endereço de `x`. A segunda linha é uma atribuição. No cálculo do valor à direita, a função `sqrt` recebe como parâmetro o conteúdo de memória *apontado* por `p`. Como `p` contém o *endereço* de `i`, e o valor armazenado em `i` é `10`, o valor `10` será o resultado da operação `(*p)`. Este valor é passado para a função `sqrt` que inicialmente o converte para um `double` usando um "cast" implícito de `int` para `double` (a função `sqrt` foi definida, em `math.h`, como uma função que requer um `double` e retorna um `double`). O retorno de `sqrt` é

```
3.1622. . . .
```

Este valor é somado com `1.0`, resultando em

```
4.1622 . . .
```

que deve ser armazenado na variável indicada no lado esquerdo da atribuição, no caso `(*f)`. Então, o valor contido em `f` é tomado como um *endereço* e, neste endereço, devemos armazenar o valor calculado. Como `f` contém o endereço de `x`, o resultado final, `4.1622. . .`, é armezenado em `x` (após ser silenciosamente convertido para um `float` por outro "cast" implícito).

A terceira linha ilustra uma situação semelhante. O valor *apontado por* `f` é convertido em um `int` (pelo "cast"), resultando em `4`. Subtraímos `3`, obtendo `1`. Este valor é armazenado no endereço *apontado por* `p`. Como `p` contém endereço de `i`, armazenamos `1` na variável `i`. As duas última linhas devem confirmar que o valor dos endereços armazenados em `p` e `f` não devem se alterar, enquanto que os valores contidos em `i` e `x`, que são as variáveis apontadas por `p` e `f`, devem sofrer as alterações previstas. Finalmente,

```
*q=i+1;              /* PERIGO !! */
x=(*e)-1.0;          /* PERIGO !! */
printf("Val i = %d, x = %f\n",i,x);
printf("Val (*q) = %d, val x = %f\n",*q,x);  /* PERIGO !! */
```

Na primeira linha, estamos calculando `i+1`, com resultado `2`. Em seguida estamos armazenando este valor no *endereço apontado por* `q`. Mas `q` ainda não foi inicializada com nenhum endereço. Neste ponto, `q` pode conter uma configuração qualquer de bits que, quando interpretada como um endereço, poderia apontar para uma posição de memória reservada por outro programa. A atrubuição seria inválida e obteríamos um erro de execução. O mesmo problema acontece com a expressão `(*e)`, uma vez que o apontador `e` também não foi inicializado, A última linha contém outra referência a um endereço (possivelmente) inválido quando vamos avaliar o resultado da expressão `(*q)`.

Lendo tipos complexos

A linguagem C suporta vetores, funções e apontadores. Com esses objetos podemos usar, respectivamente, os operadores `[ ]`, `( )` e `*`. Combinando esses operadores entre si, podemos construir outros tipos bem mais complexos em C.

Declarações mais complexas em C sempre causam problemas de interpretação e *são potenciais focos de erros*. Para dominar a construção e a interpretação de declarações mais complexas em C, há duas regras básicas que devem ser conhecidas: a regra de precedência entre os operadores de tipos e a regra para agrupamentos entre eles.

Quanto à precedência, os operadores para construir vetores e funções têm a mesma precedência. Além disso, ambos têm precedência sobre o operador que designa apontadores. Assim, a hierarquia de precedência fica na forma

```
Precedência alta:    [ ], ( )
Precedência baixa:   *
```

Note que sempre podemos usar os parênteses para impor outra ordem de precedência, como é feito em expressões aritméticas comuns.

Quanto à associatividade, os operadores `[ ]` e `( )` associam-se da esquerda para a direita. O operador `*` associa-se da direita para a esquerda. Assim,

```
int   x[ ]( )
```

seria agrupado da esquerda para a direita, resultando em

```
int ( x[ ] )( )
```

onde o primeiro par de parênteses serve de agrupador e o segundo par indica função. Então, a variável `x` está sendo declarada como um vetor de funções que retornam inteiros. No caso do operador `*`, a declaração

```
int **p
```

seria agrupada da direita para a esquerda, resultando em

```
int *(*p)
```

e estaríamos declarando `p` como uma variável que armazenará o endereço do endereço de um valor inteiro.

Repare que é natural o operador `*` agrupar pela direita, uma vez que ele é usado de maneira pré-fixada, ou seja, é aposto ao nome da variável. Não faria sentido escrever `(* *)p`. Da mesma forma, os operadores de construção de vetores e funções são usados de maneira pós-fixada, sendo natural agrupá-los pela esquerda. Não poderíamos escrever, pois, `x( []() )`.

Sempre devemos analisar uma declaração em C partindo do nome da variável que está sendo declarada e, obedecendo as regras de precedência e agrupamento, a cada passo envolver mais operadores, terminando por esgotar a declaração toda.

A melhor forma de se adquirir experiência na análise de declarações mais complexas em C, é estudar alguns exemplos típicos. Nos casos abaixo, as duas regras descritas acima são sempre seguidas à risca. Além disso, usamos o tipo primitivo `int` em

todas as declarações. Deve ficar claro, porém, que o tipo `int` poderia ser substituído por qualquer outro tipo já definido pelo programador.
Nos próximos exemplos, iremos progredindo sistematicamente na direção de tipos mais complexos. Repare que algumas combinações dos operadores produzem tipos inválidos, que não são suportados pelo compilador C.

- `int x;`
  Variável `x` declarada como de *tipo inteiro*.

- `int *x;`
  Variável `x` declarada como um *apontador* para um *inteiro*.

- `int x[ ];`
  Variável `x` declarada como um *vetor* que contém *inteiros*.

- `int x( );`
  Variável `x` declarada como uma *função* que retorna um *inteiro*.

- `int *x[ ];`
  Variável `x` declarada como um *vetor* que contém *apontadores* para *inteiros*.
  Equivalente a

  ```
  int *(x[ ]);
  ```

  Portanto, prosseguindo de "dentro para fora", vemos que `x` é um vetor (por causa da parte `x[]`), onde cada elemento é um apontador para inteiro (parte `int *y`, onde `y` representa `x[]`).

- `int *x( );`
  Variável `x` declarada como uma *função* que retorna um *apontador* para *inteiro*.
  Equivalente a

  ```
  int *(x( ));
  ```

- `int x[ ]( );`
  Variável `x` declarada como um *vetor* onde cada posição contém uma *função* que retorna um *inteiro*. Equivalente a

  ```
  int (x[ ])( );
  ```

  Mas, essa declaração resulta num erro de compilação, pois vetores de funções não são suportados em C. Isto é razoável, visto que o tamanho do código de cada função é variável, ou seja, é heterogêneo, indo contra a noção de vetores, que agrupam apenas objetos homogêneos.

- `int x( )[ ];`
  Variável `x` declarada como uma *função* que retorna um *vetor* de *inteiros*.
  Seria equivalente a

  ```
  int (x( ))[ ];
  ```

Mas essa declaração também resulta num erro de compilação, pois funções não podem retornar vetores em C. Isto é razoável, pois não poderíamos atribuir o resultado retornado. Lembre que não há atribuição direta de um vetor para outro em C.

- `int x[ ][ ];`
  Variável `x` declarada como um *vetor* onde cada posição contém um *segundo vetor* de *inteiros*. Equivalente a

```
int (x[ ])[ ];
```

  É o mesmo que uma matriz bidimensional.

- `int x( )( );`
  Variável `x` declarada como uma *função* que retorna uma *segunda função* que, por sua vez, retorna um *inteiro*. Equivalente a

```
int (x( ))( );
```

  Causa um erro de compilação, pois funções em C não podem retornar (o código de) outras funções. De novo, não poderíamos "atribuir" o código retornado a outra variável em C.

- `int *x[ ]( );`
  Equivalente a

```
int *( (x[ ])( ) );
```

  Variável `x` declarada como um *vetor* onde cada posição contém uma *função* que retorna um *apontador* para um *inteiro*. Causa erro de compilação, pois não temos vetores de funções em C.

- `int *x( )[ ];`
  Equivalente a

```
int *( (x( ))[ ] );
```

  Variável `x` declarada como uma *função* que retorna um *vetor* onde cada posição contém um *apontador* para um *inteiro*. Causa erro de compilação, pois não temos funções que retornam vetores em C.

- `int *x[ ][ ];`
  Equivalente a

```
int *( (x[ ])[ ] );
```

  Variável `x` declarada como um *vetor* onde cada posição contém um *segundo vetor* de *apontadores* para *inteiros*. É o mesmo que uma matriz bidimensional de apontadores para inteiros.

- `int *x( )( );`
  Equivalente a

  ```
  int  *( (x( ))( ) );
  ```

  Variável `x` declarada como uma *função* que retorna uma *segunda função* que, por sua vez, retorna um *apontador* para um *inteiro*. Causa um erro de compilação, pois funções em C não podem retornar (o código de) outras funções.

- `int (*x)[ ];`
  Parênteses usados para quebrar a regra de precedência natural. Variável `x` declarada como um *apontador* para um *vetor* de *inteiros*.

- `int (*x)( );`
  Parênteses usados para quebrar a regra de precedência natural. Variável `x` declarada como um *apontador* para uma *função* que retorna um *inteiro*.

- `int **x;`
  Equivalente a

  ```
  int *(*x);
  ```

  Variável `x` declarada como um *apontador* (parte `*x`) que aponta para outro *apontador* que, por sua vez, aponta para um *inteiro* (parte `int *( ... )`).

- `int (*x[ ])( );`
  Variável `x` declarada como um *vetor* de *apontadores* (parte `*x[ ]`) para um *funções* que retornam um *inteiro* (parte `int (...)( )`).

- `int *(*(*x)( ))[ ];`
  Vamos por partes:
  - `(*x)( )`: variável `x` declarada como um *apontador* para uma *função* que retorna ....
  - `(*(...))`: um *apontador* para ...
  - `int *(...)[ ]`: um *vetor* de *apontadores* para *inteiros*.

  Juntando: variável `x` declarada como um *apontador* para uma *função* que retorna um *apontador* para um *vetor* de *apontadores* para *inteiros.*
  Nesse ponto, as declarações estão ficando muito longas para serem expressas todas numa mesma linha. Estamos perdendo em clareza e aumentando muito as chances de introduzir erros. Mais adiante veremos uma maneira mais prática para escrever tais declarações.

Vetores: pontos importantes

Vetores são uma maneira de reservarmos espaço de memória para toda uma série de elementos *homogêneos*, *i.e.*, elementos que são todos do *mesmo tipo*. Assim uma declaração na forma

```
int  vet[4];
```

reserva espaço em memória para armazenar *até quatro* objetos de tipo `int`. Os objetos poderão ser acessados através de seu *índice* no vetor. Observe que:

- Todos os objetos armazenados em `vet` serão do tipo `int`.
- O nome do vetor, *i.e.*, do objeto sendo declarado, é `vet`.
- O *primeiro* índice é sempre *zero*. O fato de que, em C, vetores são sempre indexados a partir de 0 costuma ser uma grande fonte de erros, quando se começa a programar na linguagem. Portanto, fique alerta para isso.
- A declaração acima reserva espaço para *quatro* objetos de tipo `int`. Portanto o número que aparece na declaração do vetor é o número *total* de elementos que poderá ser armazenado. O índice do último elemento do vetor, entretanto, é uma unidade *a menos* que esse número (porque a indexação começa em zero).

Quando lê a declaração de um vetor, o compilador sabe quantos bytes de memória deve reservar. No exemplo, vai reservar 4x4 = 16 bytes. Isto porque cada inteiro ocupa 4 bytes (nesta arquitetura de máquina, digamos) e estamos reservando espaço para 4 inteiros. Estes 16 bytes serão reservados, não importa se usemos todos eles durante a execução do programa ou não.

Por outro lado, o compilador *não* garante que os objetos no vetor serão inicializados com qualquer valor.

O compilador garante que o espaço reservado será formado por um único bloco consecutivo de endereços de memória, e que os elementos do vetor serão armazenados seqüencialmente neste espaço reservado. Assim, se o endereço do primeiro byte no exemplo acima for 101, o primeiro elemento do vetor estará alocado nos endereços de 101 a 104 (ocupa 4 bytes); o segundo elemento deve ocupar os quatro próximos bytes de endereços, ou seja, o segundo elemento ocupará os bytes de endereços 105 a 108; e assim por diante.

Para acessar (ou atribuir) o valor armazenado em uma das posições do vetor, referenciamos o elemento que queremos acessar através de seu *índice*. O índice pode ser calculado através de qualquer expressão que retorne um tipo **`int`**. Por exemplo, `vet[2]`, `vet[i+1]`, `vet[(j++)*3]`, uma vez que as expressões `2`, `i+1` e `(j++)*3` retornam inteiros (assumindo que `i` e `j` sejam de tipo `int`). No primeiro caso, estamos acessando o elemento na posição `2` do vetor, i.e., o *terceiro* elemento do vetor; nos outros dois casos, o valor do índice depende dos valores armazenados nas variáveis `i` e `j` quando as expressões forem calculadas.

Mas note que o resultado da expressão *deve estar* entre os índices *válidos* para aquele vetor. No exemplo, são válidos os índices de `0` a `3`. Se tentarmos referenciar um elemento do vetor com um índice que esteja fora do intervalo de índices válidos estaremos tentando acessar um endereço de memória (que pode estar) fora do controle do programa. Ou seja, é um endereço que não está *reservado* para *esta* execução deste programa. Neste caso, provavelmente, vamos obter um erro de execução e o programa será sumariamente terminado. *Esta é uma grande fonte de dor de cabeça quando se lida com vetores de maneira descuidada.*

Dependendo de como o compilador reservou espaço de memória para as outras variáveis do programa, pode ser que, por uma coincidência, o endereçamento inválido ainda caia dentro de um local de memória reservado para o programa que está executando. Aí a situação será ainda *pior* pois o erro de execução não será flagrado e o programa continua executando, porém com os dados em memória possivelmente corrompidos (se *atribuirmos* um valor à posição com índice inválido, por exemplo).

Se o índice inválido for calculado dinâmicamente (por exemplo, através de uma expresão que contém variáveis), o erro poderá se tornar intermitente e difícil de detectar, o que o transforma em um erro *ainda mais difícil* de ser caçado e eliminado. Portanto, *cuidado com os índices e lembre que o primeiro é zero, não um.*
Quando declaramos um vetor, por exemplo, na forma

```
char v[4];
```

o que o compilador C faz é criar uma *constante* de nome `v`, com tipo `char*` (um *apontador* para caractere), e inicializa esta constante com o *endereço* do *primeiro* elemento do vetor. Suponha que os endereços de memória alocados para o vetor comecem na posição `101`. A declaração, na verdade, cria uma constante de nome `v` e inicializa seu conteúdo com `101`. Os demais elementos do vetor estão nos endereços `102`, `103` e `104` (lembre que cada caractere ocupa um byte na memória). Note que `v` se comporta como uma variável de tipo apontador para `char`, com a exceção de que não podemos atribuir para ela. Em particular, `v` também ocupa espaço na memória. Suponha que o endereço de `v` seja `500`.
A notação `v[4]` na declaração é apenas uma forma natural de se informar o nome do vetor e seu tamanho. A partir da declaração, o compilador sabe que deve criar a constante `v`, reservar 4 bytes de memória e atribuir a `v` o endereço do primeiro byte reservado.
Podemos também inicializar o vetor, já no momento da declaração. Para isso, basta listar os elementos do vetor, em ordem, separando cada dois componentes por uma vírgula e incluindo a lista entre `{` e `}`. Assim,

```
int v[5] = { 0, 1, 2 };
```

declara um vetor de 5 posições. As três primeiras são inicializadas com os números `0`, `1` e `2`. As duas últimas são automaticamente inicializadas em zero pelo compilador. Sempre que inicializamos apenas parte de um vetor, as demais posições recebem zeros.

Vetores e apontadores

Consire de novo a declaração

```
int  vet[4];
```

e lembre que `v` contém o endereço de `v[0]`. Então `*v` resulta no *conteúdo* de `v[0]`. Também, a expressão `v+1` deve resultar no endereço de `v[1]`. Lembre da aritmética envolvendo apontadores, onde o compilador automaticamente soma ao conteúdo de `v` o número certo de bytes para que `(v+1)` resulte no endereço de `v[1]`, mesmo que este endereço não seja a posição de memória seguinte ao endereço de `v[0]`. Na verdade, o compilador soma a `v` o número de bytes ocupados por cada elemento de `v`, de forma que `(v+1)` resulte no endereço desejado. E `v+2` resulta no endereço de `v[2]`, e assim por diante. Logo, `*(v+1)` resulta no *conteúdo* de `v[1]`, e `*(v+2)` resulta no conteúdo de `v[2]`.

Na verdade, quando encontra uma expressão na forma `v[i+3]`, por exemplo, o compilador a substitui por `*(v+(i+3))`. Este mecanismo, de percorrer o vetor usando apontadores, é muito útil em várias situações.
Note que o nome `v` é declarado como uma *constante*. Então, não será permitido atribuir outro valor para `v` (mesmo porque isso poderia levar à perda do endereço real de `v[0]`).
Vamos agora examinar mais alguns trechos de código, para ilustrar como percorrer vetrores através de apontadores.

```
char v[ ] = {'Z','M','T','A'};
char *c;
```

Declara um vetor `v` de caracteres, com 4 posições e com a inicialização indicada. Declara também uma variável `c` como um apontador para caracteres.

```
printf("Tamanho de v = %1d bytes\n",(int)sizeof(v));
```

Informa o tamanho do objeto `v`. O resultado é

```
Tamanho de v = 4 bytes
```

Como esperado, cada caractere ocupa 1 byte, e os quatro caracteres em `v` ocupam 4 bytes. Então,

```
printf("Conteudo de v = %p, endereco de v[0] = %x\n",v,&v[0]);
```

Informa o conteúdo da constante `v` e o endereço de `v[0]`. Ambos devem coincidir. O resultado é (na minha máquina)

```
Conteudo de v = 8f124, endereco de v[0] = 8f124
```

Como esperado. Mais ainda,

```
printf("Mais: *v = %c, *(v+1) = %c, *v+1 = %c\n",*v,*(v+1),*v+1);
printf("E mais: *(v+3) = %c; e agora *(v+4) = %c\n",*(v+3),*(v+4));
```

Imprime informações sobre o conteúdo de algumas posições em `v`. No primeiro caso, o resultado de `*v` e de `*(v+1)` devem coincidir com os valores na primeira e segunda posições de `v`. A primeira impressão resulta em

```
Mais: *v = Z, *(v+1) = M, *v+1 = [
```

Vemos o resultado esperado. Mas observe a terceira informação. O resultado de `*v+1` é `[`. Veja a diferença entre a expressão com parênteses, `*(v+1)`, e sem parênteses, `*v+1`. Como o operador `*` se liga mais fortemente que o operador `+`, no segundo caso obtemos o valor de `*v` (que é o mesmo que `v[0]`, ou seja, é o caractere `'Z'`) e as este valor adicionamos `1`. O resultado é o próximo caractere na tabela ASCII, logo após o `'Z'`. Uma consulta à tabela vai confirmar que este caractere é o `'['`. No segundo `printf`, pedimos os conteúdos de `v[3]` e `v[4]`. A saída é

```
E mais: *(v+3) = A; e agora *(v+4) = @
```

O conteúdo de `v[3]` confere com o último caractere no vetor. O conteúdo de `v[4]`, entretanto, está uma posição *além* do vetor. Não se pode prever o que virá como resposta aqui. Em seguida:

```
c=v;
printf("\n*(c+2) = %c\n",*(c+2));
```

Colocamos em `c` o mesmo conteúdo de `v`. Ou seja, `c` passa a apontar também para a primeira posição de `v`. Agora, esperamos que `*(c+2)` resulte no mesmo caractere que há em `v[2]`, portanto. A saída é

```
*(c+2) = T
```

confirmando o esperado. E ainda,

```
*(v+2)='X';
*c='Y';
printf("v[0] = %c, v[1]= %c, v[2]= %c, v[3] = %c\n", v[0], v[1], v[2],
v[3]);
```

Essas duas atribuições devem mudar os conteúdos de `v[2]` e de `v[0]`. A saída do `printf` é

```
v[0] = Y, v[1]= M, v[2]= X, v[3] = A
```

Vemos que `v[0]` mudou para `'Y'` devido à segunda atribuição, e `v[2]` mudou para `'X'`, por conta da primeira atribuição.

Exemplo

O próximo exemplo implementa o "método da peneira" para listar todos os números primos até um limite estabelecido (um número é primo se for pelo menos 2 e se for divisível de maneira exata apenas por 1 e por si mesmo).

O método funciona assim: construimos um vetor de inteiros onde armazenamos 1 em toda as posições. Em seguida, fazemos várias passadas pelo vetor, eliminando as posições que *não* correspondem a números primos. Eliminar uma posição significa colocar zero naquela posição do vetor. Ao final, as posições que não forem eliminadas corresponderão aos primos procurados (*i.e.*, para listar todos os primos, basta imprimir as posições do vetor, de 2 em diante, que contêm 1).

A cada passada, começamos com uma posição inicial, `inic`. Partindo de `inic+1`, percorrremos todo o resto do vetor, eliminando as posições que são múltiplos de `inic`. Isso feito, posicionamos `inic` na próxima posição ainda não eliminada e repetimos o ciclo.

Quando o cilclo termina? Podemos terminar quando `inic` ultrapassar a raíz quadrada do limite onde queremos chegar. Isso porque se um número tem um fator (que não é 1 nem é o próprio número), então também tem um fator que é, no máximo, igual a raíz quadrada do número (ele não pode ter dois fatores maiores que sua raíz quadrada pois o produto destes dois fatores já resultaria maior que o próprio número!). Iniciamos o código com

```
#include <math.h>
#define MAX 10000
#define MAX_LINHA 50
```

Incluimos o arquivo `math.h` pois vamos precisar da função `sqrt`. O identificador `MAX` representa o maior intervalo com o qual o programa pode lidar. Já `MAX_LINHA` é o máximo número de caracteres numa linha de saída, para quando formos imprimir a listagem dos primos encontrados.

```
void peneira(int inic, int dim); // elimina multiplos de inic no vetor p
                                 // de dimensão dim
void imprimevals(int dim);       // percorre vetor p e imprime os indices
                                 // de valores validos
int p[MAX+1];                    // vetor onde p[i]=1 se i ainda não foi
                                 // eliminado; variavel global
```

Pragmas de duas funções. O trabalho da primeira função é eliminar os múltiplos de `inic` no vetor `v`, cujo limite é `dim`. O trabalho da segunda função é percorrer o vetor `v` e imprimir todas as posições (a partir da posição `2`) que contenham um valor não nulo (estas posições são aquelas que *não foram* eliminadas). Ambas as funções não retornam valores úteis e o vetor `p` é declarado. Neste programa, optamos por ignorar o índice zero do vetor. Portanto, para acomodar até `MAX` elementos a partir da posição `1`, o vetor deverá ter `MAX+1` elementos.

```
   int inic, fim, limite;
   int  i;
```

As variáveis `inic`, `fim` e `limite` são locais na função `main` e sevirão para marcar posições no vetor. A variável `i` é auxiliar.

```
printf("Entre com o intervalo de procura (inteiro ate %1d): ",MAX);
do {
    scanf("%d",&limite);
    if ((limite>1) && (limite<=MAX)) break;
    printf("Limite de %1d invalido. Tente de novo: ",limite);
} while (1);
```

Pede um limite para o intervalo de procura (de onde vai extrair todos os números primos). O laço garante que teremos este limite (na variável de nome também `limite`) entre `2` e `MAX`. Se o usuário entrar com dado inválido, o programa pede outro dado, até que o limite esteja no intervalo indicado.

```
for(i=2;i<=MAX;i++) p[i]=1;
fim=(int)sqrt( (double) limite);
```

Este trecho inicializa o vetor (com todas as posições ativas) e armazena na variável `fim` o maior fator que teremos que considerar. Repare no "cast" para converter o tipo `double` retornado pela função **`sqrt`** para um `int`. O valor inicial de `inic` já foi colocado em `2`.

```
do {
 if (!p[inic]) continue;
 peneira(p,inic,limite);
} while (++inic<=fim);
```

Este é o laço principal. Repeditamente, invocamos a função `peneira` para eliminar posições múltiplas de `inic` (de cada vez indo da posição inicial até a posição armazenada em **`limite`**).

Repare no `if`. Se a posição `inic` já contiver um zero, então já foi eliminada e não precisamos mais eliminar seus múltiplos (eles também já foram eliminados em alguma passada anterior). O comando `continue` faz com que o programa desvie direto para a condição de teste do laço (no caso, a condição associada ao `while`), ignorando os outros comandos deste ponto até o fim do laço. No caso, isso faz com que o programa ignore a chamada da rotina `peneira` e passe direto para a condição de teste.
Na condição de teste, o valor de `i` é primeiramente incrementado. Em seguida, testamos se ainda não ultrapassamos o maior fator a considerar. Se ainda não, o laço executa de novo; caso afirmativo, o laço termina.
Esta estratégia implementa o "método da peneira".

```
    printf("Os primos entre 2 e %1d sao:\n",limite);
    imprimevals(p,limite);
```

Considere agora a função `peneira`.

```
void peneira(int *v, int inic, int dim){
    int i;
    for(i=inic+1;i<=dim;i++) {
    if (!*(v+i)) continue;
    if (!(i%inic)) *(v+i)=0;
    }
}
```

Esta função elimina posições múltiplas de `inic` no vetor `v`, até a posição `dim`.
Repare que declaramos o primeiro argumento da função na forma `int *v`, ou seja como um *apontador* para inteiros. Portanto, quando a função for invocada, podemos passar o *nome* (*i.e.* o endereço) de um vetor de inteiros como primeiro argumento, uma vez que o nome do vetor é também do tipo apontador para inteiro. Note também que esta é uma declaração *local* para a função `peneira`. Isto é, implicitamente, estamos agregando a declaração `int *v` às demais declarações locais de `peneira`. Localmente, a variável `v` é uma variável comum, à qual podemos atribuir e também referenciar. Se tivéssemos declarado o primeiro argumento de `peneira` na forma `int v[]`, o compilador também criaria uma variável local comum também do tipo apontador para inteiros. Note que se atribuirmos para `v` no corpo da função, apenas a *cópia local do endereço*. Quando a rotina terminasse, esta cópia local seria dinâmicamente desalocada e recuperaríamos o endereço original. No entanto, tome muito cuidado ao atribuir para `v` no corpo da função, uma vez que você estará destruindo o endereço do primeiro elemento do vetor, que está armazenado, para todo o resto da execução do código da função.
O laço `for` inicia em `inic+1`, pois não queremos eliminar a própria posição `inic`. A cada posição considerada, testamos se já foi eliminada. Caso afirmativo, vamos para a próxima posição. Este é o efeito do comando `continue` aqui. Portanto, o segundo `if` só é executado se a posição `i` ainda não foi eliminada. O `if` será positivo exatamente quando a posição `i` for um múltiplo de `inic` e, neste caso, eliminamos a posição `i` atribuindo `0` ao elemento `v[i]`.

```
void imprimevals(int *v, int limite) {
    int nc=1, col=1, ncols, i;
    for( i=limite; i>=10; i /= 10) nc++;
    nc++;
    ncols=MAX_LINHA/nc;
```

A rotina `imprimevals` inicia calculando o número de dígitos que há no valor armezenado em `limite`. Esta informação será usada para auxiliar na impressão dos resultados, formando colunas alinhadas. Este número é armazenado na variável `nc` que, após o `for`, tem seu valor incrementado para permitir um espaço entre uma coluna e a seguinte. Na variável `ncol` calculamos o número de colunas numa linha, usando `MAX_LINHA` como um limite da extenção de uma linha.

```
for(i=2;i<=limite;i++) {
 if (*(v+i)) {
   printf("%*d",nc,i);
   if (col==ncols) {col=1; printf("\n");
 }
 else col++;
}
```

Este laço `for` percorre o vetor, imprimindo as posições onde há um valor não nulo.
Note o modificador `%*d` na função `printf`. Este modificador sinaliza que o próximo argumento do `printf` será usado no lugar do símbolo `*`, de modo que podemos controlar o número de casas para a impressão do argumento na lista de saída deste `printf`. No caso, usamos sempre `nc` como o número de casas a ocupar, garantindo então que as colunas estarão alinhadas.
Após imprimir cada valor, o segundo `if` testa se já imprimimos `ncol` colunas nesta linha. Se for este o caso, acrescentamos um caractere de "fim-de-linha" e reposicionamos o contador `col` em `1`. Caso contrário, basta incrementar `col`, indicando que estaríamos usando mais uma coluna desta linha.

Apontadores para funções

Como em outras linguagens de programação, em C podemos criar apontadores para variáveis de diferentes tipos. O apontador, nesses casos, armazena o endereço de memória onde está localizada a informação.
Mais C vai um passo além, permitindo com que criemos também *apontadores para funções*. Nesses casos, o apontador armazena o endereço de memória onde se encontra o (início do) código da função. Dessa forma, dereferenciando o apontador, podemos ter acesso ao código da função e podemos executá-lo.
O compilador lida com o nome de funções e nomes de vetores de forma similar. Quando passamos um vetor como argumento em uma função em C, o compilador trata o nome do vetor como um *apontador* para a primeira posição da área de memória ocupada pelo vetor. Assim, dada a declaração

```
int x[10];
```

seriam equivalentes os seguintes comandos:

```
printf("%d\n",x[0]);
```

e

```
printf("%d\n",*x);
```

ambos resultando na impressão do valor armazenado na primeira posição do vetor `x`. Quando o nome do vetor ocorre isoladamente, *i.e.*, sem estar associado ao operador `[ ]`, que indica uma posição indexada, o compilador interpreta esse nome como um *apontador* para o primeiro endereço de memória do vetor. Por exemplo, junto com a declaração

```
int *pi;
```

a seqüência de comandos

```
pi=x;
printf("%d, %d\n",*pi,pi[0]);
```

resulta também na dupla impressão do valor armazenado na primeira posição do vetor `x`. Note que, como `pi` é um apontador, podemos indexar a partir da posição de memória para onde `pi` aponta, sem problemas. Nesse caso, é claro, obteremos o primeiro valor armazenado em `x`, visto que atribuímos o valor de `x` a `pi`.
Se dereferenciarmos o nome `x`, tomando seu *endereço*, obteremos como valor o mesmo endereço armazenado em `x`, que é também o endereço da primeira posição do vetor `x`. Por exemplo, o comando

```
printf("%p, %p, %p\n",&x[0],x,&x);
```

imprimiria três valores idênticos. Mas o comando

```
printf("%p, %p\n",pi,&pi);
```

produz valores distintos. Ocorre que o nome `x` não é exatamente uma variável como é `pi`. Esta última é uma variável usual, com uma posição de memória reservada para si e, portanto, com um *endereço* de memória único onde são armazenados os valores atribuídos a `pi` . O nome `x`, que designa um vetor, é tratado de uma forma diferenciada pelo compilador. Por isso, o *endereço* de `x` resulta também no mesmo valor que o endereço do objeto apontado por `x` (que é o primeiro endereço na memória onde estão armazenados os valores do vetor), o que não ocorre com `pi`, por exemplo.
Devemos lembrar também que, embora o nome `x` possa ser encarado como um apontador para `int`, ele representa uma *constante*, i.e., não podemos atribuir novos valores para `x`. Assim, a atribuição

```
x=pi;
```

embora de tipos compatíveis, não será aceita.
Um *nome de função* é tratado de forma semelhante pelo compilador. Ou seja, não é uma variável no sentido usual, mas também designa um *apontador* (cujo valor não pode ser alterado) para o endereço de memória onde se encontra o código da função. Podemos atribuir esse apontador para outra variável de mesmo tipo, tomar o endereço desse apontador, ou dereferenciá-lo.
Por exemplo, assuma a declaração da função (externa ao programa principal)

```
int soma(int a, int b) { return (a+b);}
```

que introduz `soma` como uma função que recebe dois argumentos de tipo `int` e retorna outro valor de tipo `int`. Assuma também a declaração de variável

```
int (*f)( );
```

que introduz `f` como um apontador para uma função que retorna um `int`. Se executássemos o comando

```
printf("%p, %p, %p, %p\n",soma,&soma, f, &f);
```

os dois primeiros valores seriam idênticos, mostrando o endereço de memória onde está o código da função `soma`. Os dois últimos valores seriam distintos, o penúltimo indicando o valor armazenado na variável `f` (ainda sem atribuição) e o último indicaria o endereço de memória associado à essa variável.
Se os próximos comandos fossem

```
f=soma;
printf("%p, %p, %p, %p\n",soma,&soma, f, &f);
```

obteríamos os mesmos valores que anteriormente, exceto pelo terceiro deles. Agora, o valor armazenado em `f` resulta da avaliação da expressão `soma`. Como não usamos o operador `( )`, que forçaria a execução do código da função `f`, o compilador interpreta como uma referência ao nome soma, retornando um *apontador* (i.e. *o endereço*) da função `soma`. Portanto, os três primeiros valores impressos serão agora idênticos. O último ainda indica o endereço da variável `f`.
O comando

```
printf("%d, %d\n",soma(2,3),f(4,5));
```

resulta numa linha com os valores `5` e `9`. Note que não precisamos dereferenciar explicitamente o apontador `f`. O compilador, quando lê o nome `f` seguido do operador `( )`, entende que o programador está invocando o código da função apontada por `f` e gera código apropriado, nesse ponto, para realizar o desvio para o endereço de memória onde se encontra o código daquela função.
Podemos dereferenciar explicitamente o apontador designado por `f`. Na verdade, como o nome `soma` também designa um apontador desse mesmo tipo, podemos dereferenciá-lo também, obtendo o mesmo resultado. Assim, o comando

```
printf("%d, %d\n",(*soma)(2,3),(*f)(4,5));
```

resultaria na impressão dos mesmos valores: `5` e `9`. Quando dereferenciamos um apontador *para função*, o compilador retorna o mesmo valor associado ao nome antes de derefenciá-lo, ou seja, retorna o endereço do código da função. Isso porque entende que tudo que podemos fazer com o código é excutá-lo. Não podemos imprimí-lo, adicionara a ele, ou indexar a partir da sua posição inicial de memória, como podemos fazer com vetores.
Assim, os comandos

```
f=**soma;
```

```
printf("%d, %d\n",(*soma)(2,3),(****f)(4,5));
```

resultariam nos mesmos valores 5 e 9 impressos na saída.

Exemplo
Considere o código C:

```
int x[10]={100,2,3,4,5,6,7,8,9,0};
int *pi;
printf("%d, %d\n",x[0],(*x));
pi=(int *)(&x);
printf("%p, %d, %d\n",pi,*pi,pi[0]);
printf("%p, %p, %p\n",&x[0],x,&x);
printf("%p, %p\n",pi,&pi);
```

Uma posssível saída seria:

```
100, 100
8f300, 100, 100
8f300, 8f300, 8f300
8f300, 8f2fc
```

```
int soma(int a, int b) {
      return (a+b);
}
int (*f)(int x, int y );
printf("%p, %p, %p, %p\n",soma,&soma, f, &f);
printf("%p, %p\n",*soma,*f);
printf("%p, %p\n",**soma,**f);
f=**soma;
printf("%p, %p, %p, %p\n",soma,&soma, f, &f);
printf("%d, %d\n",soma(2,3),f (4,5));
printf("%d, %d\n",(*soma)(2,3),(*f)(4,5));
printf("%d, %d\n",(***soma)(2,3),(****f)(4,5));
```

Uma possível saída seria:

```
1570, 1570, 0, 8f2f8
1570, 0
1570, 0
1570, 1570, 1570, 8f2f8
5, 9
5, 9
5, 9
```

Funções como parâmetros

Podemos passar um *nome de função* como argumento para outras funções em C. Isso possibilita um mecanismo flexível, onde a função que entra como argumento pode ser trocada a cada invocação da função mestre.
Uma declaração de função na forma

```
int f( double g( int x), int a);
```

declara `f` como uma função que recebe dois argumentos e retorna um `int.` O segundo argumento é uma variável de tipo `int`. O primeiro argumento é uma função que recebe um `int` e retorna um `double`.

Lembre que o compilador trata nomes de funções como apontadores para endereços de memória. Assim, essa declaração seria equivalente a

```
int f( double (*g)( int x), int a);
```

onde está claro que `g` é um *apontador* para uma função que recebe um `int` e retorna um `double`.

O método de Newton é um algoritmo eficiente para se achar raízes de funções (contínuas). Se **`f`** é uma função, uma raíz de **`f`** é todo valor **`x`** onde **`f`** se anula, isto é, **`f(x)=0`**.

Note que, se num ponto temos **`f(a) > 0`**, e se num outro ponto **`b`** mais adiante temos **`f(b) < 0`**, ou seja **`f(a)`** e **`f(b)`** têm sinais trocados, então `f` deve se anular em algum ponto intermediário entre `a` e **`b`.** O método de Newton determina um ponto intermediário **`c`**, entre **`a`** e **`b`**, tal que `f(c) =0` (pode haver mais de um tal ponto **`c`**; o método acha um deles).

O método funciona assim. Assuma que **`f(a) > 0`** e que **`f(b) <0`**. Calculamos o ponto intermediário **`m=(a+b)/2`**. Se **`f(m) = 0`**, o ponto `m` é a raíz desejada. Se **`f(m) > 0`** então a raíz deve estar entre os pontos `m` e **`b`**, visto que **`f(m)>0`** **e** **`f(b)<0`** indicam a presença de uma raíz entre `m` e `b`. Caso contrário, a raíz está entre **`a`** e **`m`.** Em qualquer hipótese, o intervalo onde se encontra a raíz foi dividido à metade. O processo continua até que a raíz seja encontrada. É o mecanismo de dividir o intervalo útil de busca à metade, a cada passo, que torna este método muito eficiente.

Exemplo

Considere o código:

```
#include<math.h>
#define EPSILON 1.0E-10
```

A biblioteca `math.h` é incluída para que tenhamos acesso à função cosseno. A constante `EPSILON` é usada para contornar o problema do interstício mínimo na representação binária de números fracionários. Assim, dois números fracionários que diferem de menos que `EPSILION` serão considerados iguais. É o mesmo que dizer que aceitamos um erro de até `EPSILON` nos valores calculados.

```
double poli(double a) {
 /* funcao = 3.3a^5 - 25.0a^2 + 204.5a - 18.6 */
 return (3.3*a*a*a*a*a - 25.0*a*a + 204.5*a - 18.6);
}

double raiz(double f(double), double a, double b) {
 double m,x;
 do {
  m=(a+b)/2.0;
  x=f(m);
  if ( (fabs(x<EPSILON)) || (fabs(b-a<EPSILON)) ) break;
```

```
  if (f(a)*x>0.0) a=m; else b=m;
 }while (1);
return m;
}
```

A função `poli` é um simples polinômio. A função recebe o ponto onde queremos calcular o valor do polinômio e devolve o valor calculado.
A função `raiz` implementa o método de Newton. Seu primeiro argumento é uma função `f` que recebe um `double` e devolve outro `double`. O código da função `raiz` é formado por um laço `do-while` que, a cada iteração, realiza um passo do método de Newton. Observe que a função retorna assim que acha a raíz desejada, quando `x<EPSILON`, ou quando a distância entre os pontos extremos do intervalo útil atual é menor que `EPSILON`. Nos dois testes, é claro, devemos tomar o valor absoluto das grandezas (qualquer valor negativo é trivialmente menor que `EPSILON`, mesmo grandezas negativas com um valor absoluto alto). Em qualquer hipótese, estamos informando o valor correto da raíz, a menos de um erro menor que `EPSILON`.
A rotina deve calcular corretamente uma das raízes de qualquer função `f` que seja passada como parâmetro, desde que a função tenha valores de sinais trocados nos extremos do intervalo `[a,b]`. É responsabilidade do programador garantir essa última condição, uma vez que a rotina não verifica se ela é verdadeira quando começa a executar.

```
printf("Funcao cosseno entre 0 e pi:\n");
r = raiz(cos,0.0,3.141592);
printf("\t Raiz no ponto %g\n",r);
printf("\t Valor de cos(%g) = %g\n",r,cos(r));
```

Nesse trecho do programa principal testamos a rotina `raiz` sobre a função cosseno. Essa função tem o valor `1` no ponto zero e o valor `-1` no ponto `3.14` (em radianos). Portanto, ela deve ter uma raíz em algum ponto intermediário. A saída é

```
Funcao cosseno entre 0 e pi:
 Raiz no ponto 1.5708
 Valor de cos(1.5708) = 1.56387e-11
```

A raíz está no ponto `1.5708` (em radianos). A linha seguinte confirma que a função calculada nesse ponto tem um valor próximo de zero (abaixo de `EPSILON`). Observe os parâmetros de chamada da função `raiz`. O primeiro deles é o nome da função cosseno, `cos`. Na execução do código de `raiz`, o nome do parâmetro formal `f` é substituído pelo nome da função cosseno, fazendo com que, durante a execução, a rotina raiz use de fato a função cosseno nos cálculos, como desejado.

```
printf("Funcao poli entre 0 e 10.0:\n");
r = raiz(poli,0.0,10.0);
printf("\t Raiz no ponto %g\n",r);
printf("\t Valor de poli(%g) = %g\n",r,poli(r));
```

Novo teste, agora usando a função `poli`. A saída é:

```
Funcao poli entre 0 e 10.0:
 Raiz no ponto 0.0919879
```

```
Valor de poli(0.0919879) =  -8.02357e-10
```

De novo, fica comprovado que no ponto calculado para a raíz o valor da função é próximo de zero.