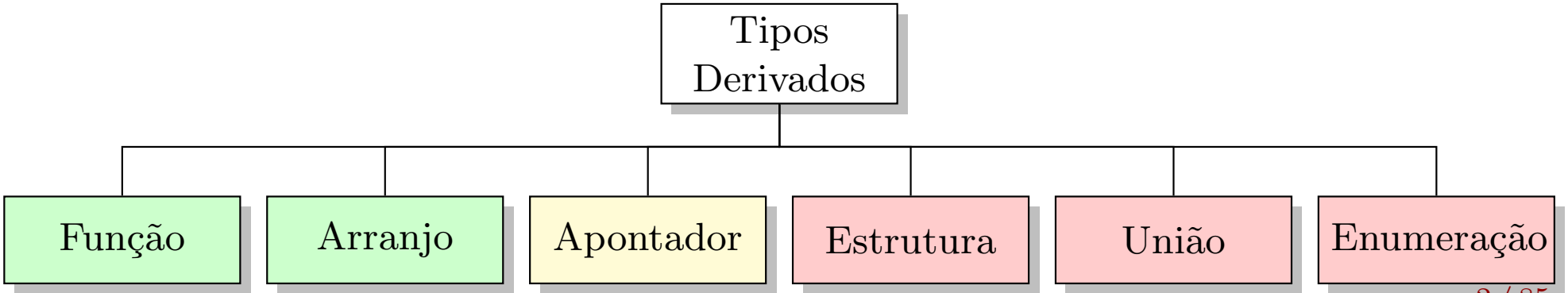
Apontadores

Alexsandro Santos Soares
prof.asoares@gmail.com

Universidade Federal de Uberlândia
Faculdade de Computação

Introdução

- Todo computador possui localizações de memória endereçáveis.
- Até agora todas as manipulações de dados que fizemos, sejam elas para inspeção ou alteração, usavam *nomes* de variáveis para localizar os dados.
  - Associávamos identificadores aos dados e depois manipulávamos seus conteúdos por meio desses identificadores.
- Apontadores tem muitos usos em C:
  - É um método muito eficiente de acessar dados.
  - Possibilitam técnicas eficientes de manipulação de dados em arranjos.
  - São usados em funções como parâmetros passados por endereço.
  - Formam a base para a alocação dinâmica de memória.
- O apontador é o terceiro dos tipos derivados.



Definição

- Um apontador é uma constante ou variável que contém um endereço que pode ser usado para acessar dados.
- Apontadores são construídos sobre o conceito básico de constantes apontadoras.
- Para compreender e usar apontadores devemos primeiro compreender esse conceito.

Constantes caracteres

- Primeiro vamos comparar constantes caracteres com constantes apontadoras.
- Sabemos que podemos ter uma constante caractere, tal como uma letra.
- Uma constante caractere é um valor e pode ser armazenada em uma variável.
- Embora a constante caractere não possua um nome, a variável possui um nome que é declarado no programa.
- No trecho a seguir

```
char umChar;
umChar = 'G';
```

  a variável caractere `umChar` contém a constante caractere `'G'`. A variável `umChar` possui um nome e um endereço de memória.
- O nome é criado pelo programador mas o endereço é uma posição dentro do espaço de memória do programa.

Constantes apontadoras

- Assim como as constantes caracteres, constantes apontadoras não podem ser alteradas.
- O endereço de memória de `umChar` foi retirado do conjunto de constantes apontadoras do computador.
- Embora o endereço dentro de um computador não mude, o endereço de uma variável, por exemplo `umChar`, pode e mudará de uma execução para outra.
  - Isso se deve ao fato de que os sistemas operacionais modernos podem posicionar um programa na memória onde for mais conveniente.
  - Assim, se `umChar` estiver armazenado no endereço de memória 145 600 agora, na próxima execução ele poderá estar localizado em 876 050.
- Portanto, mesmo os endereços sendo constantes, podemos não saber quais serão.
  - Entretanto, ainda devemos ter uma forma se referir a eles simbolicamente.

Valores apontadores

- Acabamos de definir constantes apontadoras como endereços de memória.
- Agora voltaremos nossa atenção para o armazenamento desses endereços.
- Se temos uma constante apontadora, devemos conseguir armazenar seu valor se tivermos alguma forma de identificá-lo.
  - Isso não é apenas possível, mas já estamos fazendo isso desde que escrevemos o primeiro comando `scanf` com um operador de endereço.
- O operador de endereço (&) extrai o endereço de uma variável.
- O formato do operador de endereço é

```
&nome_da_variável
```

Exemplo 1 (Imprimido os endereços de variáveis do tipo char)

Vamos escrever um programa que define duas variáveis caracteres e imprime os endereços delas como ponteiros, cujo código de conversão em `printf` é `%p`.

```
9  #include <stdio.h>
10
11 int main(void){
12   char a;
13   char b;
14
15   printf("endereço de a: %p\n", &a);
16   printf("endereço de b: %p\n", &b);
17   return 0;
18 } // main
```

- Dependendo do sistema operacional, esse programa pode imprimir números diferentes cada vez que for executado.
- Note que, na maioria das vezes, o computador aloca dois endereços adjacentes de memória pois definimos as variáveis uma após a outra.

Saída do programa

Saída de três execuções diferentes do programa

```
> ./exemplo1.exe
endereço de a: 0x7ffc30970fe6
endereço de b: 0x7ffc30970fe7

> ./exemplo1.exe
endereço de a: 0x7ffe0a468746
endereço de b: 0x7ffe0a468747

> ./exemplo1.exe
endereço de a: 0x7ffcec667976
endereço de b: 0x7ffcec667977
```

Exemplo 2 (Imprimido os endereços de variáveis do tipo int)

Considere agora imprimir os endereços de variáveis inteiras. Em muitos computadores, cada variável ocupará 4 bytes. Quais desses quatro endereços de memória será usado para posicionar a variável?

```
 9 #include <stdio.h>
10
11 int main(void){
12   int a = -123;
13   int b = 145;
14
15   printf("a: endereço %p valor %4d\n", &a, a);
16   printf("b: endereço %p valor %4d\n", &b, b);
17   return 0;
18 } // main
```

- Em muitos computadores, a posição do primeiro byte é usado como endereço de memória.
- No caso da variável caractere, existe somente 1 byte, assim, a posição desse único byte já é o endereço.
- A mesma ideia é aplicada a outros tipos de dados:
  - O endereço de uma variável é aquele do primeiro byte ocupado pela variável.

Saída do programa

Saída de três execuções diferentes do programa

```
> ./exemplo2.exe
a: endereço 0x7fff8e813400 valor -123
b: endereço 0x7fff8e813404 valor  145

> ./exemplo2.exe
a: endereço 0x7ffc20adb730 valor -123
b: endereço 0x7ffc20adb734 valor  145

> ./exemplo2.exe
a: endereço 0x7ffdc83eee70 valor -123
b: endereço 0x7ffdc83eee74 valor  145
```

Variáveis apontadoras

- Se temos uma constante apontadora e um valor apontador, então podemos ter uma variável apontadora.
- Dessa forma podemos armazenar o endereço de uma variável em uma outra variável chamada de *variável apontadora*.
- Devemos diferenciar uma variável de seu valor, como veremos no próximo slide.

Variáveis apontadoras

- Como exemplo, suponha que a variável `a` esteja armazenada no endereço 234 560:

```
int a;
int *p;

a = -123;
p = &a;
```

- O valor de `a` é `-123`.
- Numa mesma execução o nome e o endereço de uma variável é constante, mas seu valor pode mudar à medida que o programa for executando.
- Nesse código há uma variável apontadora `p` que possui um nome e um endereço, ambos contantes numa mesma execução.
- O valor de `p` é o endereço de memória 234 560, o endereço de `a`.
- Diremos que `p` está apontando para `a`.

Variáveis apontadoras

- Vamos agora armazenar o endereço de uma variável em duas ou mais variáveis apontadoras diferentes:

```
int a;
int *p;
int *q;

a = -123;
p = &a;
q = &a;
```

- As apontadoras tem, cada uma, um nome e um endereço, ambos contantes.
- O valor das apontadoras nesse ponto é o endereço de memória 234 560, o endereço de `a`.
  - Ou seja, ambas estão apontando para `a`.
- Não há limite para o número de variáveis apontadoras que podem apontar para uma variável.

Variáveis apontadoras

- Se tivermos uma variável apontadora, mas não queremos que ela aponte para lugar algum, qual é o seu valor?
- Em C, temos a constante apontadora nula `NULL` que pode ser encontrada em `stdio.h`.
- NULL é definida como uma macro com um valor inteiro 0 ou 0 coagido para apontador void (`void *`).

Acessando variáveis por meio de apontadores

- Como podemos usar uma apontadora?
- Em C, podemos encontrar o operador de indireção (`*`) que permite o desreferenciamento.
- Quando desreferenciamos uma apontadora, estamos usando seu valor (um endereço) para referênciar uma outra variável.
- O operador de indireção é um operador unário cujo operando deve ser um *valor apontador*.
- O resultado é uma expressão que pode ser usada para acessar a variável apontada tanto para inspeção quanto para alteração.
- Para acessar a variável `a` do último exemplo via a apontadora `p`, usamos simplesmente

```
*p
```

Exemplo 3 (Alterando o valor de uma variável usando apontador)

Vamos supor que precisemos adicionar 1 à variável `a`. O programa a seguir mostra quatro formas equivalentes de alcançar esse resultado.

```
 9 #include <stdio.h>
10
11 int main(void){
12   int a = 0;
13   int *p = NULL;
14
15   printf("antes: a = %d\n", a);
16   a++;
17   printf("depois: a = %d\n\n", a);
18
19   a = 0;  // retorna ao valor original
20
21   printf("antes: a = %d\n", a);
22   a = a + 1;
23   printf("depois: a = %d\n\n", a);
```

Alterando o valor de uma variável usando apontador

```
25    a = 0;  // retorna ao valor original
26    p = &a; // coloca o endereço de a em p
27
28    printf("endereço de a: %p\n", &a);
29    printf("valor de p:  %p\n\n", p);
30
31    printf("antes: a = %d\n", a);
32    *p = *p + 1;
33    printf("depois: a = %d\n\n", a);
34
35    a = 0;  // retorna ao valor original
36
37    printf("antes: a = %d\n", a);
38    (*p)++;
39    printf("depois: a = %d\n\n", a);
40
41    return 0;
42 } // main
```

Na última forma, `(*p)++`, precisamos dos parênteses, já que o operador de incremento pósfixo possui precedência de 16, enquanto o de indireção, que é unário, possui precedência de 15. Logo, os parentêses fazem com que o desreferenciamento ocorra antes da adição, incrementando assim a variável de dados e não a apontadora.

Saída do programa

Saída da execução do programa:

```
> ./exemplo3.exe
antes:  a = 0
depois: a = 1

antes: a = 0
depois: a = 1

endereço de a: 0x7ffc549d3a4c
valor de p:    0x7ffc549d3a4c

antes:  a = 0
depois: a = 1

antes:  a = 0
depois: a = 1
```

Exemplo 4 (Acessando variáveis por meio de apontadores)

Suponha que uma variável `x` seja apontada por duas apontadoras, `p` e `q`. As expressões `x`, `*p` e `*q` permitem, todas elas, que a variável seja inspecionada ou alterada. Quano elas são usadas no lado direito de uma atribuição, elas podem somente inspecionar. Quando elas são usadas no lado esquerdo, elas podem alterar o valor de `x`.

```
 9 #include <stdio.h>
10
11 int main(void){
12   int x = 0;
13   int *p = NULL;
14   int *q = NULL;
15
16   p = &x;
17   q = &x; // ambas apontadores recebem o mesmo endereço
18
19   printf("endereço de x: %p\n", &x);
20   printf("valor de p:  %p\n", p);
21   printf("valor de q:  %p\n\n", q);
```

Acessando variáveis por meio de apontadores

```
23    x = 4;
24    printf("Primeira atribuição a x ocorreu.\n");
25    printf("depois: x = %d\n" , x);
26    printf("depois: *p = %d\n", *p);
27    printf("depois: *q = %d\n\n", *q);
28
29    x = x + 3;
30    printf("Segunda atribuição a x ocorreu.\n");
31    printf("depois: x = %d\n" , x);
32    printf("depois: *p = %d\n", *p);
33    printf("depois: *q = %d\n\n", *q);
34
35    *p = 8;
36    printf("Atribuição a *p ocorreu.\n");
37    printf("depois: x = %d\n" , x);
38    printf("depois: *p = %d\n", *p);
39    printf("depois: *q = %d\n\n", *q);
40
41    *&x = *q + *p;
42    printf("Atribuição a *&x ocorreu.\n");
43    printf("depois: x = %d\n" , x);
44    printf("depois: *p = %d\n", *p);
45    printf("depois: *q = %d\n\n", *q);
```

Acessando variáveis por meio de apontadores

```
47    x = *p * *q;
48    printf("Atribuição a x com multiplicação.\n");
49    printf("depois: x = %d\n" , x);
50    printf("depois: *p = %d\n", *p);
51    printf("depois: *q = %d\n\n", *q);
52
53    return 0;
54  } // main
```

- Os operadores de indireção e de endereço são o inverso um do outro.
  - Quando combinados em uma expressão, tal como `*&x`, eles se cancelam mutuamente.
  - Para ver isso, observe que ambas as expressões unárias são avaliadas a partir da direita.
    1. Logo, a primeira expressão a ser avaliada é `&x`, o endereço de `x`, e, como visto, é um valor apontador.
    2. A segunda expressão `*(&x)`, desreferencia a constante apontadora, levando à variável `x`.
  - Assim, os operadores efetivamente cancelam-se um ao outro.
- Nota: em um programa real nunca usamos `*&x`, ela foi usada aqui somente como ilustração.

Saída do programa

```
> ./exemplo4.exe

endereço de x: 0x7ffde7fe5a84
valor de p:    0x7ffde7fe5a84
valor de q:    0x7ffde7fe5a84

Primeira atribuição a x ocorreu.
depois:  x = 4
depois: *p = 4
depois: *q = 4

Segunda atribuição a x ocorreu.
depois:  x = 7
depois: *p = 7
depois: *q = 7

Atribuição a *p ocorreu.
depois:  x = 8
depois: *p = 8
depois: *q = 8

Atribuição a *&x ocorreu.
depois:  x = 16
depois: *p = 16
depois: *q = 16

Atribuição a x com multiplicação.
depois:  x = 256
depois: *p = 256
depois: *q = 256
```

Definição e declaração de apontadores

- Usamos o asterisco para declarar variáveis apontadoras.
- No trecho abaixo é mostrado como declarar diferentes variáveis apontadoras.

```
char a;          char *p;
int n;           int *q;
float x;         float *r;
```

- As variáveis de dados correspondentes são mostradas para comparação.

Exemplo 5 (Imprimindo com uma apontadora)

O programa a seguir armazena o endereço de uma variável em uma apontadora e depois imprime os dados usando o valor da variável e uma apontadora.

```
 9 #include <stdio.h>
10
11 int main(void){
12   int a = 0;
13   int *pa = NULL;
14
15   a = 14;
16   pa = &a;
17
18   printf("endereço armazenado em p: %p\n", pa);
19   printf("valor no endereço apontado por p: %d\n", *pa);
20   printf("valor armazenado em a: %d\n\n", a);
21
22   return 0;
23 } // main
```

A execução do programa anterior produz

```
> ./exemplo5.exe

endereço armazenado em p:  0x7ffd7eb7d59c
valor no endereço apontado por p: 14
valor armazenado em a: 14
```

Declaração versus Redireção

- Usamos o operador asterisco em dois contextos diferentes: para declaração e para redirecionamento.
- Quando um asterisco é usado para declaração ele é associado a um tipo.

```
int *pa;
int *pb;
```

- Quando usado para redirecionamento, o asterisco é um operador que redireciona a operação da variável apontadora para uma variável de dados.

```
pa = &a;
pa = &b;
soma = *pa + *pb;
```

Aqui, `*pa` é o valor da variável apontada por `pa`, ou seja, é o valor de `a`. Analogamente, `*pb` é o valor de `b`.

Inicialização de variáveis apontadoras

- Se declararmos, mas não incializarmos, uma váriavel apontadora, quando o programa entrar em execução, o valor da variável será algum endereço desconhecido de memória.
  - Mais precisamente, ele terá um valor desconhecido que será interpretado como um endereço de memória.
- Muito provavelmente, o valor não será um endereço válido no computador que estiver usando ou, se ele for, o endereço não será válido para a memória que foi alocada para o programa.
- Se o endereço não existir, obteremos um erro em tempo de execução.
- Se ele for um endereço válido, com frequência, mas nem sempre, obteremos um erro em tempo de execução.
- Um dos erros mais comum em programação com C é deixar uma apontadora sem inicialização.
  - Esses erros podem ser muito difíceis de depurar, pois o efeito do erro pode aparecer bem depois do programa iniciar.

Inicialização de variáveis apontadoras

- Como já visto, podemos inicializar uma variável apontadora na definição usando `NULL`

```
int *p = NULL;
```

- Em muitos sistemas operacionais, o endereço 0, que é o valor de `NULL` é inválido.
- Assim, qualquer tentativa de desreferenciar uma apontadora quando seu valor é `NULL` produzirá um erro em tempo de execução.

Exemplo 6 (Alterando variáveis com apontadoras)

Vamos escrever um programa para aprender mais sobre apontadores. Foque sua atenção nos valores das diferentes variáveis e apontadores na medida em que vão sendo alterados.

```
 9 #include <stdio.h>
10
11 int main(void){
12   int a = 0;
13   int b = 0;
14   int c = 0;
15
16   int *p = NULL;
17   int *q = NULL;
18   int *r = NULL;
19
20   a = 6;
21   b = 2;
22   p = &b;
23
24   q = p;
25   r = &c;
```

Alterando variáveis com apontadoras

```
27    p = &a;
28    *q = 8;
29
30    *r = *p;
31
32    *r = a + *q + *&c;
33
34    printf("a = %d, b = %d, c = %d\n", a, b, c);
35    printf("*p = %d, *q = %d, *r = %d\n", *p, *q, *r);
36
37    return 0;
38 } // main
```

Quais são os valores que serão impressos?

Uso

A execução do programa anterior produz como saída

```
> ./exemplo6.exe

a = 6, b = 8, c = 20
*p = 6, *q = 8, *r = 20
```

Exemplo 7 (Somando dois números)

Vamos usar ponteiros para somar dois números. Esse exemplo explora o conceito de usar apontadores para manipular dados.

```
 9 #include <stdio.h>
10
11 int main(void){
12   int a = 0;
13   int b = 0;
14   int r = 0;
15
16   int *pa = &a;
17   int *pb = &b;
18   int *pr = &r;
19
20   printf("Digite o primeiro número: ");
21   scanf("%d", pa);
22   printf("Digite o segundo número: ");
23   scanf("%d", pb);
```

Somando dois números

```
25    *pr = *pa + *pb;
26    printf("\n%d + %d = %d\n", *pa, *pb, *pr);
27
28    return 0;
29  } // main
```

Que valores serão impressos se digitarmos como entradas 15 e 51?

Uso

A execução do programa anterior produz como saída

```
> ./exemplo7.exe

Digite o primeiro número: 15
Digite o segundo número: 51

15 + 51 = 66
```

Exemplo 8 (Flexibilidade dos apontadores)

Este exemplo mostra como podemos usar o mesmo apontador para imprimir o valor de variáveis diferentes.

```
10 #include <stdio.h>
11
12 int main(void){
13   int a = 0;
14   int b = 0;
15   int c = 0;
16
17   int *p = NULL;
18
19   printf("Digite três números: ");
20   scanf("%d %d %d", &a, &b, &c);
21
22   p = &a;
23   printf("%3d\n", *p);
```

Flexibilidade dos apontadores

```
25    p = &b;
26    printf("%3d\n", *p);
27
28    p = &c;
29    printf("%3d\n", *p);
30
31    return 0;
32  } // main
```

Que valores serão impressos se digitarmos como entradas 10 20 30?

Uso

A execução do programa anterior produz como saída

```
> ./exemplo8.exe

Digite três números: 10 20 30
 10
 20
 30
```

Exemplo 9 (Múltiplos apontadores para uma variável)

Este exemplo mostra como podemos usar apontadores diferentes para imprimir o valor da mesma variável.

```
10 #include <stdio.h>
11
12 int main(void){
13   int a = 0;
14
15   int *p = NULL;
16   int *q = NULL;
17   int *r = NULL;
18
19   p = &a;
20   q = &a;
21   r = &a;
22
23   printf("Digite um número: ");
24   scanf("%d", &a);
```

Múltiplos apontadores para uma variável

```
26    printf("%d\n", *p);
27    printf("%d\n", *q);
28    printf("%d\n", *r);
29
30    return 0;
31  } // main
```

Que valores serão impressos se digitarmos como entrada 15?

Uso

A execução do programa anterior produz como saída

```
> ./exemplo9.exe
Digite um número: 15
15
15
15
```

Apontadores para comunicação entre funções

- Uma das aplicações mais úteis de apontadores é o seu uso em funções.
- Quando discutimos funções algumas aulas antes, vimos que C usa passagem por valor para comunicação descendente.
- Já para a comunicação ascendente, o único modo direto de enviar algo de volta de uma função é usar o comando `return`.
- Também vimos que podemos usar as comunicações ascendente e bidirecional passando um endereço, usando-o para devolver dados para a função chamadora.
- Quando passamos um endereço, estamos de fato passando um apontador para uma variável.
- Agora, vamos desenvolver melhor a ideia de comunicação bidirecional.

Passando Endereços

Exemplo 10 (Permutando valores)

Este exemplo mostra como permutar os valores de duas variáveis usando apontadores.

```
10 #include <stdio.h>
11
12 void permuta(int *px, int *py){
13   int temp;
14
15   temp = *px;
16   *px = *py;
17   *py = temp;
18   return;
19 } // permuta
```

Permutando valores

```
21 int main(void){
22   int a = 5;
23   int b = 7;
24
25   permuta(&a, &b);
26   printf("%d %d\n", a, b);
27
28   return 0;
29 } // main
```

A execução desse programa produz como saída

```
> ./exemplo10.exe

7 5
```

Resumo

- Quando precisamos enviar mais que um valor para a função chamadora, usamos apontadores.
- Cada vez que precisamos que a função chamada tenha acesso a uma variável na função chamadora, passamos o endereço daquela variável para a função chamada e usamos o operador de indireção para acessá-lo.

Funções que devolvem apontadores

- Mostramos alguns exemplos de funções usando apontadores, mas não mostramos nenhum delas devolvendo um apontador.
- Nada impede uma função de devolver um apontador para a função chamada.
  - De fato, é muito comum que funções devolvam apontadores.
- Veremos um exemplo disso no próximo slide.

Exemplo 11 (Menor entre dois números)

Neste exemplo vamos escrever uma função para determinar o menor entre dois números. Nesse caso, precisamod de um apontador para o menor valor entre as duas variáveis `a` e `b`.

```
10 #include <stdio.h>
11
12 int * menor(int *px, int *py){
13   return (*px < *py ? px : py);
14 } // menor
15
16 int main(void){
17   int a = 0;
18   int b = 0;
19   int *p = NULL;
20
21   printf("Digite dois números: ");
22   scanf("%d %d", &a, &b);
23
24   p = menor(&a, &b);
25
26   printf("O menor valor é%d\n", *p);
27   return 0;
28 } // main
```

Permutando valores

A execução desse programa produz como saída

```
> ./exemplo11.exe

Digite dois números: 30 2
O menor valor é 2
```

Erros na devolução de apontadores

- Sempre que devolvermos um apontador, ele deve apontar para dados na função chamadora ou em alguma outra função em um nível mais alto na chamada.
- É um erro sério devolver um apontador para uma variável local na função chamada.
  - Quando a função termina o espaço na memória ocupado por ela está livre para ser usado por outras partes do programa.
- Talvez esse erro não seja percebido em um programa pequeno, pois o espaço pode não ser reusado, mas, em um programa grande pode-se obter uma resposta errada ou uma falha quando a memória sendo referenciada pelo apontador for alterada.

Apontadores para apontadores

- Até agora os nossos apontadores estiveram sempre apontando diretamente para os dados.
- É possível – e com estruturas de dados avançadas, frequentemente necessário – usar apontadores que apontem para outros apontadores.
- Por exemplo, podemos ter uma apontador apontando para um apontador para um inteiro.
- Essa indireção em dois níveis é mostrada no trecho abaixo

```
int a;
int *p;
int **pp;

a = 58;
p = &a;
pp = &p;
printf(" %3d",  a);
printf(" %3d", *p);
printf(" %3d", **pp);
```

Apontadores para apontadores

- Não há limite para quantos níveis de indireção usar, mas, na prática, raramente usamos mais que dois.
- Cada nível de indireção de apontadores requer um operador de indireção separado quando ele for desreferenciado.
- No exemplo anterior para se referir a `a` usando o apontador `p`, tivemos que desreferenciá-lo uma vez

```
*p
```

- Para se referir à variável `a` usando o apontador `pp`, tivemos que desreferenciá-lo duas vezes para obter o inteiro `a` pois existem dois níveis de indireção envolvidos.
  - Se desreferenciarmos ele apenas uma vez, estaremos referenciado `p`, que é um apontador para um inteiro.
- Uma outra foram de dizer isso é que `pp` é um apontador para um apontador de inteiro.
- A desreferência dupla é mostrada a seguir

```
**pp
```

Exemplo 12 (Apontador para apontador)

Nesse exemplo vamos usar ponteiros diferentes com apontadores para apontadores e também apontadores para apontadores para apontadores para ler o valor da mesma variável.

```
10 #include <stdio.h>
11 int main(void){
12   int a = 0;
13
14   int *p = NULL;
15   int **pp = NULL;
16   int ***ppp = NULL;
17
18   p   = &a;
19   pp  = &p;
20   ppp = &pp;
21
22   printf("Digite um número: ");
23   scanf("%d", &a);                    // usando a
24   printf("Número digitado: %d\n\n", a);
25
26   printf("Digite um número: ");
27   scanf("%d", p);                     // usando p
28   printf("Número digitado: %d\n\n", a);
```

Apontador para apontador

```
30    printf("Digite um número: ");
31    scanf("%d", *pp);                    // usando pp
32    printf("Número digitado: %d\n\n", a);
33
34    printf("Digite um número: ");
35    scanf("%d", **ppp);                  // usando ppp
36    printf("Número digitado: %d\n\n", a);
37
38    return 0;
39  } // main
```

Apontador para apontador

A execução desse programa produz como saída

```
> ./exemplo12.exe

Digite um número: 1
Número digitado:  1

Digite um número: 2
Número digitado:  2

Digite um número: 3
Número digitado:  3

Digite um número: 4
Número digitado:  4
```

Compatibilidade

- É importante reconhecer que apontadores possuem um tipo associado a eles.
- Eles são apontadores para um tipo *específico*, tal como um caractere.
- Portanto, cada apontador possui entre seus atributos o tipo para o qual ele se refere, além de outros atributos.

Compatibilidade de tamanho de apontadores

- O tamanho de todos os apontadores é o mesmo.
- Cada variável apontadora guarda o endereço de uma localização na memória do computador.
- Por outro lado, o tamanho da variável a qual o apontador referencia pode ser diferente:
  - Ele recebe esse atributo do tipo sendo referenciado.
- No próximo slide veremos um exemplo ilustrando esse ponto.

Exemplo 13 (Apontador para apontador)

Neste exemplo vamos imprimir o tamanho de cada apontador e o tamanho do objeto por ele referenciado.

```
 9 #include <stdio.h>
10
11 int main(void){
12   char c;
13   char *pc;
14   int tamanho_de_c    = sizeof(c);
15   int tamanho_de_pc   = sizeof(pc);
16   int tamanho_de_ast_pc = sizeof(*pc);
17
18   int a;
19   int *pa;
20   int tamanho_de_a    = sizeof(a);
21   int tamanho_de_pa   = sizeof(pa);
22   int tamanho_de_ast_pa = sizeof(*pa);
```

Apontador para apontador

```
24   double x;
25   double *px;
26   int tamanho_de_x    = sizeof(x);
27   int tamanho_de_px   = sizeof(px);
28   int tamanho_de_ast_px = sizeof(*px);
29
30   printf("sizeof(c): %3d | ", tamanho_de_c);
31   printf("sizeof(pc): %3d | ", tamanho_de_pc);
32   printf("sizeof(*pc): %3d\n", tamanho_de_ast_pc);
33
34   printf("sizeof(a): %3d | ", tamanho_de_a);
35   printf("sizeof(pa): %3d | ", tamanho_de_pa);
36   printf("sizeof(*pa): %3d\n", tamanho_de_ast_pa);
37
38   printf("sizeof(x): %3d | ", tamanho_de_x);
39   printf("sizeof(px): %3d | ", tamanho_de_px);
40   printf("sizeof(*px): %3d\n", tamanho_de_ast_px);
41
42   return 0;
43 } // main
```

Apontador para apontador

A execução desse programa produz como saída

```
> ./exemplo13.exe
sizeof(c):   1 | sizeof(pc):   8 | sizeof(*pc):   1
sizeof(a):   4 | sizeof(pa):   8 | sizeof(*pa):   4
sizeof(x):   8 | sizeof(px):   8 | sizeof(*px):   8
```

O que esse código está no dizendo?

1. As variáveis `a`, `b` e `c` nunca recebem valores. Isso significa que o espaço ocupado por elas na memória é independente do valor que elas guardam. Isto é, os tamanhos são dependentes do tipo e não dos valores.
2. Observe o tamanho dos apontadores. Ele sempre é 8 em todos os casos, já que esse é o tamanho em bytes de um endereço de memória no computador onde o programa foi executado.
3. Quando imprimimos o tamanho do tipo que o apontador está referenciando, ele é sempre igual ao tamanho do dado. Isso significa que além do tamanho do apontador, o sistema também sabe o tamanho de qualquer coisa que o apontador esteja apontando.

Compatibilidade de tipo de desreferenciamento

- A segunda questão relativa à compatibilidade diz respeito ao tipo de desreferenciamento.
- O tipo de desreferenciamento é o tipo da variável que o apontador está referenciando.
- Com uma única exceção, é inválido atribuir um apontador de um tipo a um apontador de um outro tipo.
  - Mesmo que os valores em ambos os casos sejam endereços de memória e poderiam, em tese, ser totalmente compatíveis.
- Em C, não podemos usar atribuição entre apontadores com tipos diferentes.
  - Se tentarmos fazer isto, obtemos um erro de compilação.

Compatibilidade de tipo de desreferenciamento

- Um apontador para um `char` somente é compatível com um apontador para um `char`.
- Um apontador para um `int` somente é compatível com um apontador para um `int`.
- Não podemos atribuir um apontador para um `char` a um apontador para um `int` e vice-versa.
- Observe o trecho de código abaixo

```
char c;
char *pc;

int a;
int *pa;

pc = &c;        // Bom e válido
pa = &a;        // Bom e válido

pc = &a;        // Erro:  tipos diferentes
pa =  a;        // Erro:  níveis diferentes
```

Apontador para Void

- A exceção à regra da compatibilidade de tipo é o apontador para void.
- Um apontador para `void` é um tipo genérico que não está associado a um tipo de referência.
  - Ele não é o endereço de um caractere, nem de um inteiro, nem de um ponto flutuante e nem de qualquer outro tipo.
- Entretanto, *somente para fins de atribuição*, ele é compatível com todos os outros tipos.
- Assim, um apontador para o tipo `void` pode ser atribuído a um apontador de qualquer tipo e vice-versa.
- Há, no entanto, uma restrição: como um apontador para `void` não possui um tipo objeto, ele não pode ser desreferenciado a menos que ele seja coagido.
- Abaixo mostramos como declarar uma variável apontadora do tipo `void`:

```
void *pVoid;
```

Coerção de tipo nos apontadores

- O problema de incompatibilidade de tipos pode ser resolvido se usarmos a coerção.
- Podemos tornar explícita uma atribuição entre apontadores de tipos incompatíveis usando uma coerção, assim como podemos coagir um `int` em um `char`.
- Se no exemplo anterior, precisássemos fazer com que o apontador para `char`, pc, passasse a apontar para um `int`, `a`, poderíamos usar a coerção:

```
pc = (char*) &a;
```

- Nesse caso, esteja ciente que a menos que usemos a coerção em todas as operações envolvendo `pc`, temos uma grande chance de produzir erros graves.
- De fato, exceto no caso de apontador para `void`, nunca devemos usar coerção em apontadores.

Coerção de tipo nos apontadores

- As atribuições seguintes são todas válidas, mas são perigosas e devem ser usados com muito critério:

```
void *pVoid;
char *pChar;
int  *pInt;

pVoid = pChar;
pInt  = pVoid;
pInt  = (int *) pChar;
```

- Um outro uso para a coerção é fornecer um tipo para um apontador para `void`.
- Como vimos, um apontador para `void` não pode ser desreferenciado pois ele não possui um tipo objeto.
- Entretanto, se usarmos a coerção damos um tipo a ele.

Compatibilidade de nível de desreferenciamento

- A compatibilidade também inclui a compatibilidade de nível de desreferenciamento.
- Por exemplo, um apontador para `int` não é compatível com uma apontador para apontador para `int`.
- O apontador para `int` possui um tipo de referência `int`, enquanto um apontador para apontador para `int` possui como tipo de referência um apontador para `int`.

Compatibilidade de nível de desreferenciamento

- O exemplo abaixo mostra dois apontadores declarados em diferentes níveis. O apontador `pa` é um apontador para `int` e o apontador `ppa` é um apontador para apontador para `int`.

```
int a;
int b;
int *pa;
int **ppa;

pa  = &a;     // Válido:  mesmo nível
ppa = &pa;    // Válido:  mesmo nível
b   = **ppa; // Válido:  mesmo nível

pa  = a;     // Inválido:  nível diferente
ppa = pa;    // Inválido:  nível diferente
b   = *ppa; // Inválido:  nível diferente
```

Lvalue e Rvalue

- Em C, uma expressão ou é um lvalue ou um rvalue.
- Qualquer expressão produz um valor, mas esse valor pode ser usado de dois modos diferentes:
  1. Uma expressão lvalue, valor à esquerda, deve ser usada quando o objeto estiver recebendo um valor, ou seja, estiver sendo modificado.
  2. Uma expressão rvalue, valor à direita, pode ser usada para fornecer um valor para uso posterior, ou seja, para examinação ou cópia.
- Como saber quando uma expressão é um lvalue ou rvalue?
- Existem sete tipos de expressões que são lvalue. Elas serão mostradas no próximo slide.

Expressões lvalue

	Tipo da expressão	Comentários
1	identificador	Identificador da variável
2	expressão[...]	indexação de arranjo
3	(expressão)	a expressão já deve ser um lvalue
4	*expressão	expressão desreferenciada
5	expressão.nome	seleção em estrutura
6	expressão -> nome	seleção indireta em estrutura
7	chamada de função	se a função usa `return` com endereço

- Abaixo estão exemplos de expressões lvalue:

```
a = ...     a[5] = ...     (a) = ...     *p = ...
```

- Todas as expresseõs que não lvalue são rvalue. Por exemplo:

```
5     a + 2     a * 6     a[2] + 3     a++
```

- Note que mesmo se uma expressão é um lvalue, se ela for usada como parte de expressão maior cujos operadores criem apenas expressões lvalue, então a expressão toda é um rvalue.

Lvalue e Rvalue

- Por que se preocupar com lvalues e rvalues?
- O motivo é que alguns operadores necessitam de lvalues como operandos.
- Se usarmos um desses operadores e colocarmos um rvalue no lugar do operando, teremos um erro de compilação.
- O operado da direita em uma atribuição sempre deve ser uma expressão rvalue.
- Apenas seis operadores necessitam de expressão lvalue como operando: operador de endereço, incremento e decremento pósfixos, incremento e decremento préfixos e atribuição.
- Um nome de variável pode assumir tanto o papel de um lvalue quanto de um rvalue dependendo de como ele é usado na expressão.
- No exemplo abaixo `a` é um lvalue pois está no lado esquerdo da atribuição enquanto `b` é um rvalue porque está no lado direito:

```
a = b
```

Exemplos com apontadores

- Mostraremos nos próximos slides dois exemplos do uso de apontadores na chamada de funções.
- No primeiro exemplo veremos como converter um tempo dado em segundos para horas, minutos e segundos.
- No segundo exemplo vamos criar uma função para calcular e devolver raízes reais de equações de segundo grau.

Exemplo 14 (Converter segundos para horas)

Vamos escrever uma função que converte o tempo em segundos para horas, minutos e segundos. Esta função requer três parâmetros de endereço para retornar valores.

```
25 int segParaHoras (long tempo, int *horas, int *minutos, int *segundos){
26   long tempoLocal = 0;
27
28   tempoLocal = tempo;
29   *segundos = tempoLocal % 60;
30   tempoLocal = tempoLocal / 60;
31
32   *minutos = tempoLocal % 60;
33
34   *horas = tempoLocal / 60;
35
36   if (*horas > 24)
37     return 0;
38   else
39     return 1;
40 } // segParaHoras
```

Converter segundos para horas

```
42 int main(void){
43   long tempo = 0;
44   int horas = 0;
45   int minutos = 0;
46   int segundos = 0;
47
48   printf("Digite o tempo em segundos: ");
49   scanf("%ld", &tempo);
50
51   if (segParaHoras(tempo, &horas, &minutos, &segundos) == 1){
52     printf("%dh %dmin %ds\n", horas, minutos, segundos);
53   } else
54     printf("A quantidade de horas ultrapassou 24\n");
55
56   return 0;
57 } // main
```

Uso

A execução desse programa produz como saída

```
> ./exemplo14.exe
Digite o tempo em segundos: 3678
1h 1min 18s

> ./exemplo14.exe
Digite o tempo em segundos: 678123
A quantidade de horas ultrapassou 24

> ./exemplo14.exe
Digite o tempo em segundos: 67812
18h 50min 12s
```

O comando switch

- Muitas vezes nos deparamos com programas que envolvem uma forma de seleção de uma entre várias alternativas:

```
int teste;
teste = expressão;
if (teste == alternativa1)
   bloco_de_comandos_1;
else if (teste == alternativa2)
   bloco_de_comandos_2;
...
else if (teste == alternativaN)
   bloco_de_comandos_N;
else  // se nenhum dos testes anteriores for verdadeiro
   bloco_de_comandos_padrão;
```

- Esse tipo de código é tão comum em C que foi criado um comando para abreviá-lo, o `switch`, como veremos no próximo slide.

O comando switch

- O trecho anterior poderia ser expresso assim com `switch`:

```
switch (expressao){
   case alternativa1: bloco_de_comandos_1; break;
   case alternativa2: bloco_de_comandos_2; break;
   ...
   case alternativaN: bloco_de_comandos_N; break;
   default:           bloco_de_comandos_padrão; break;
} // fim switch
```

- O comando `switch` somente pode ser usado com testes que produzam um valor inteiro, ou que possam ser reduzido a um.
- Assim também é o caso das alternativas usadas com `case` e mais ainda, ele deve ser um valor constante.
- o caso padrão, marcado no código por `default`, somente é usado se nenhuma das alternativas anteriores for verdadeira.

O comando switch

```
switch (expressao){
   case alternativa1: bloco_de_comandos_1; break;
   case alternativa2: bloco_de_comandos_2; break;
   ...
   case alternativaN: bloco_de_comandos_N; break;
   default:           bloco_de_comandos_padrão; break;
} // fim switch
```

- Note também o uso do comando `break` cujo efeito é fazer com o fluxo de execução do programa salte para fora do comando `switch` e continue com o código que segue o `switch`.
  - Se no final de bloco de comandos de um `case` você esquecer de colocar o `break`, o `case` seguinte será executado.

Exemplo de uso do switch

- Algumas semanas atrás criamos um menu de escolhas assim:

```
void menu(unsigned long num){
    int opcao = 0;
    unsigned long resultado = 0;
    while (opcao != SAIR){
        exibirOpcoes();
        opcao = lerOpcao();

        if (opcao == 1){
            resultado = fatorial(num);
        } else if (opcao == 2){
            resultado = fibonacci(num);
        } else if (opcao == 3){
            printf("Digite um inteiro: ");
            scanf("%lu", &num);
        } else {
        } // else
        imprimirResultado(opcao, num, resultado);
    } // while
    return;
} // menu
```

Exemplo de uso do switch

- Se usarmos o comando `switch`, a mesma função agora será:

```
void menu(unsigned long num){
    unsigned long resultado = 0;
    int opcao = 0;
    while (opcao != SAIR){
        exibirOpcoes();
        opcao = lerOpcao();

        switch( opcao ){
          case 1: resultado = fatorial(num);
                  break;
          case 2: resultado = fibonacci(num);
                  break;
          case 3: printf("Digite um inteiro: ");
                  scanf("%lu", &num);
                  break;
          default: break; // não faz nada
        } // switch
        imprimirResultado(opcao, num, resultado);
    } // while
    return;
```

Exemplo 15 (Raízes reais de equações de segundo grau)

Vamos escrever um programa que leia, processe e imprima dados. Um ciclo comum em muitos programas. Para demonstrar essa ideia calcularemos as raízes reais de uma equação de segundo grau na forma

$$ax^2 + bx + c = 0$$

```
9  #include <stdio.h>
10 #include <math.h>
11
12 /**
13  * @brief Lê os coeficientes da equação de segundo grau.
14  *
15  * @param[out] pa coeficiente a
16  * @param[out] pb coeficiente b
17  * @param[out] pc coeficiente c
18  *
19  * @post os coeficientes serão modificados
20  */
21 void leiaDados(int *pa, int *pb, int *pc){
22   printf("\nDigite os coeficientes a, b e c: ");
23   scanf("%d %d %d", pa, pb, pc);
24   return;
25 } // leiaDados
```

```
44 int quadratico(int a, int b, int c,
45                double *pRaiz1, double *pRaiz2){
46   int resultado = 0;
47
48   double discriminante = 0.0;
49   double raiz = 0.0;
50
51   if (a == 0 && b == 0)
52     resultado = -1;
53   else if (a == 0) {
54       *pRaiz1 = -c / (double) b;
55       resultado = 1;
56   } // a == 0
57   else {
58       discriminante = b * b - (4 * a * c);
59       if (discriminante >= 0){
60          raiz = sqrt(discriminante);
61          *pRaiz1 = (-b + raiz) / (2 * a);
62          *pRaiz2 = (-b - raiz) / (2 * a);
63          resultado = 2;
64       } else
65          resultado = 0;
66   } // else
67
68   return resultado;
69 } // quadratico
```

```
82 void imprimeResultados(int numRaizes,
83                       int a, int b, int c,
84                       double raiz1, double raiz2){
85   printf("Sua equação: %dx^2 + %dx + %d\n", a, b, c);
86
87   switch(numRaizes){
88   case 2: printf("As raízes são: %6.3f e %6.3f\n", raiz1, raiz2);
89           break;
90   case 1: printf("A única raiz é: %6.3f\n", raiz1);
91           break;
92   case 0: printf("As raízes são complexas.\n");
93           break;
94   default: printf("Os coeficientes são inválidos.\n");
95            break;
96   } // switch
97
98   return;
99 } // imprimeResultados
```

```
101 int main(void){
102   int a = 0;
103   int b = 0;
104   int c = 0;
105   int numRaizes = 0;
106   double raiz1 = 0.0;
107   double raiz2 = 0.0;
108   char novamente = 'S';
109
110   printf("Resolve equações de segundo grau.\n\n");
111   while (novamente == 'S' || novamente == 's'){
112     leiaDados(&a, &b, &c);
113     numRaizes = quadratico(a, b, c, &raiz1, &raiz2);
114     imprimeResultados(numRaizes, a, b, c, raiz1, raiz2);
115
116     printf("\nVocê quer resolver uma outra equação? (S/N) ");
117     scanf(" %c", &novamente);
118   } // while
119   printf("\nAté mais\n\n");
120
121   return 0;
122 } // main
```

Uso

Lembre-se que ao incluir `math.h` no código deve-se incluir também a biblioteca matemática no processo de compilação:

```
gcc -std=c11 exemplo15.c -o exemplo15.exe -lm
```

Depois de compilado, a execução produz como saída:

```
> ./exemplo15.exe

Resolve equações de segundo grau.

Digite os coeficientes a, b e c: 2 4 2
Sua equação: 2x^2 + 4x + 2
As raízes são: -1.000 e -1.000

Você quer resolver uma outra equação? (S/N) s

Digite os coeficientes a, b e c: 0 4 2
Sua equação: 0x^2 + 4x + 2
A única raiz é: -0.500
```

Uso

```
Você quer resolver uma outra equação? (S/N) s

Digite os coeficientes a, b e c: 2 2 2
Sua equação: 2x^2 + 2x + 2
As raízes são complexas.

Você quer resolver uma outra equação? (S/N) s

Digite os coeficientes a, b e c: 0 0 2
Sua equação: 0x^2 + 0x + 2
Os coeficientes são inválidos.

Você quer resolver uma outra equação? (S/N) s

Digite os coeficientes a, b e c: 1 -5 6
Sua equação: 1x^2 + -5x + 6
As raízes são:  3.000 e  2.000

Você quer resolver uma outra equação? (S/N) n
Até mais.
```

Para saber mais

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.

Fontes

- Forouzan, B. A and Gilbert, R. F. *Computer Science: a structured programming approach using C*. 3rd edition. Cengage Learning, 2007.