Universidade Federal do Espírito Santo

Departamento de Computação e Eletrônica - CEUNES
Estrutura de Dados I
Prof. Oberlan Romão

Ponteiros

Em um programa, cada variável possui seu endereço de memória – que representa o local, ou posição, na memória onde o dado está de fato armazenado – e um conteúdo, que é a informação propriamente dita.

Um ponteiro (ou apontador) é um tipo especial de variável, cujo conteúdo não é um simples valor, mas um endereço de memória. Na Figura 1, pode ser observado que o conteúdo de uma variável ponteiro corresponde a um endereço de uma variável não ponteiro. Note que o conteúdo das variáveis à esquerda (ponteiros) indicam os endereços das variáveis à direita (não ponteiros).

Variável ponteiro	
1200	4567
5496	1539
8954	0009

Variável não ponteiro	
4567	A
1539	34
0009	17

Figura 1

Ponteiros são ferramentas poderosas na linguagem de programação C. Dentre diversas funções, os ponteiros permitem a passagem de parâmetros sem cópia de muitos dados (passagem por referência), retorno de múltiplos valores em funções, alocação dinâmica de memória e construção de listas encadeadas.

Existe um valor especial para apontadores conhecido como valor nulo ou `NULL`. O valor `NULL` indica que um apontador, intencionalmente, não aponta para nenhuma variável. Vale lembrar que um apontador, quando não inicializado, não aponta necessariamente para `NULL`. A aplicabilidade do valor `NULL` ficará mais claro no decorrer da disciplina.

Sintaxe dos ponteiros

Uma declaração de ponteiro consiste no tipo do ponteiro, um * (asterisco) e o nome da variável:

```
<tipo do ponteiro> *<nome da variável>
```

No lugar do `<tipo do ponteiro>`, deve-se utilizar alguns dos tipos-padrão da linguagem C, como `char`, `int`, `double`, etc., ou até mesmo novos tipos definidos pelo programador. No código abaixo, são declarados ponteiros dos tipos `char`, `int` e `float`.

```c
int main(){
    char *c;
    int *p;
    float *f;

    return 0;
}
```

Assim, no código, `c` pode apontar para uma área de memória que guarda uma variável do tipo `char`, enquanto `p` e `f` pode apontar para variáveis do tipo inteiro e ponto flutuante, respectivamente

Os operadores de ponteiro

Existem dois operadores especiais para ponteiros: `*` e `&`. O `&` é um operador unário[1] que devolve o endereço da memória do seu operando. Por exemplo,

```c
int *m;
int count = 0;
m = &count;
```

atribui a `m` o endereço da memória que contém a variável `count`. Esse endereço é a posição interna ao computador da variável. O endereço não tem relação alguma com o valor de `count`. O operador `&` pode ser imaginado como retornando "o endereço de" ou "aponta para". Assim, o comando de atribuição `m = &count` significa "`m` recebe o endereço de `count`" ou "`m` aponta para `count`".

Para entender melhor a atribuição anterior, assuma que a variável `count` usa a posição de memória `0x123` (valor em hexadecimal) para armazenar seu valor. Assuma também que `count` tem o valor 100. Então, após a atribuição anterior, `m` terá o valor `0x123`.

O exemplo abaixo (`ex1.c`) mostra o uso do operador `&` para inicializar uma variável apontador. A variável `m` recebe o endereço da variável `count`. Note que a variável ponteiro `m` é do mesmo tipo da variável `count`, pois é fundamental que o apontador aponte para uma variável de tipo compatível com a sua declaração.

```c
#include <stdio.h>

int main(){
    int *m;
    int count = 100;
    m = &count;
    printf("          Endereco de 'count': %p\n", &count);
    printf("              Endereco de 'm': %p\n", &m);
    printf("Endereco de apontado por 'm': %p\n", m);
    return 0;
}
```

[1] Um operador unário requer apenas um operando

Exercício 1:

a) Compile e execute o código anterior. O que o código imprime?

b) Você consegue descobrir o que o %p, nos `printf`, faz?

c) Qual a diferença entre fazer `printf("%p", &m)` e `printf("%p", m)`?

d) Se comentarmos a linha 6, faz sentido fazer o `printf` da linha 8? E o da linha 9? Por quê?

e) Com a linha 6 comentada, compile e execute o código novamente. Algum erro foi gerado? Isso está certo?

Abra o link `https://repl.it/@oberlan/ED1-Aula-Pratica-0-Exercicio-1` e execute o código. Após compilar o código, o programa Valgrind é chamado para testar o programa. Veja que o código (aba "Code") contém o código anterior, mas com a linha 6 comentada. O Valgrind detectou alguma coisa errada?

Se você estiver usando o site `https://ide.cs50.io/` e quiser usar o Valgrind, basta compilar o código e executar o comando: `valgrind ./executavel`, onde `executavel` é o nome do programa compilado.

O segundo operador de ponteiro, `*`, é o complemento de `&`. É um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se `m` contém o endereço da variável `count`,

```
1 int q = *m;
```

coloca o valor de `count` em `q`. Portanto, `q` terá o valor 100 porque 100 estava armazenado na posição `0x123`, que é o endereço que estava armazenado em `m`. O operador `*` pode ser imaginado como "no endereço". Nesse caso, o comando anterior significa "q recebe o valor que está no endereço apontado por `m`".

Veja o exemplo abaixo (`ex2.c`),

```
1  #include <stdio.h>
2
3  int main(){
4      int *p;
5      int c = 15;
6      p = &c;
7      printf("Antes da alteracao de 'p'\n");
8      printf(" Valor de 'c': %d\n", c);
9      printf("Valor de '*p': %d\n", *p);
10     *p = *p + 1; // Altera o conteudo para onde o ponteiro 'p' aponta
11     printf("Depois da alteracao de 'p'\n");
12     printf(" Valor de 'c': %d\n", c);
13     printf("Valor de '*p': %d\n", *p);
14     return 0;
15 }
```

Exercício 2: Compile e execute o código anterior. O que acontece com o valor de `c`? Por quê?

Como qualquer outro tipo de variável, um ponteiro também pode ser usado no lado direito de um comando de atribuição para passar seu valor para um outro ponteiro. Por exemplo (`ex3.c`):

```c
#include <stdio.h>

int main(){
  int x = 10;
  int *p1, *p2;
  p1 = &x;
  p2 = p1;
  printf("End. x  = %p\n", &x);
  printf("End. p1 = %p\n", &p1);
  printf("End. p2 = %p\n", &p2);
  printf("End. apontado por p1 = %p\n", p1);
  printf("End. apontado por p2 = %p\n", p2);
  printf("Valor do end. apontado por p1 = %d\n", *p1);
  printf("Valor do end. apontado por p2 = %d\n", *p2);
  return 0;
}
```

Veja uma possível saída ao se executar o código anterior:

```
End.  x = 0x7ffebfe2dda4
End.  p1 = 0x7ffebfe2dda8
End.  p2 = 0x7ffebfe2ddb0
End.  apontado por p1 = 0x7ffebfe2dda4
End.  apontado por p2 = 0x7ffebfe2dda4
Valor do end.  apontado por p1 = 10
Valor do end.  apontado por p2 = 10
```

A Figura 2 ilustra o que acontece internamente na memória ao executar o código anterior. Note que ao declararmos os ponteiros `p1` e `p2` (linha 5), os mesmo apontam para qualquer endereço (ou seja, possuem um "lixo" como valor). **Tentar acessar o valor apontado para um apontador não inicializado irá causar (quando não puder, Lei de Murphy) *segmentation fault*, se atentem a isso**. A próxima instrução inicializa o ponteiro `p1` (`p1 = &x;`). A partir desse momento, `p1` passa a apontar para o endereço de `x` e qualquer modificação no valor apontado por `p1` significa uma mudança no valor de `x`. Em seguida, `p2` recebe o valor de `p1`, ou seja, `p2` também passa a apontar para a `x`. Dessa forma, se modificarmos o valor apontado por `p1` ou `p2` (ou seja, se fizermos, por exemplo, `*p1 = 15` ou `*p2 = 20`) estamos modificando o valor da variável `x`.

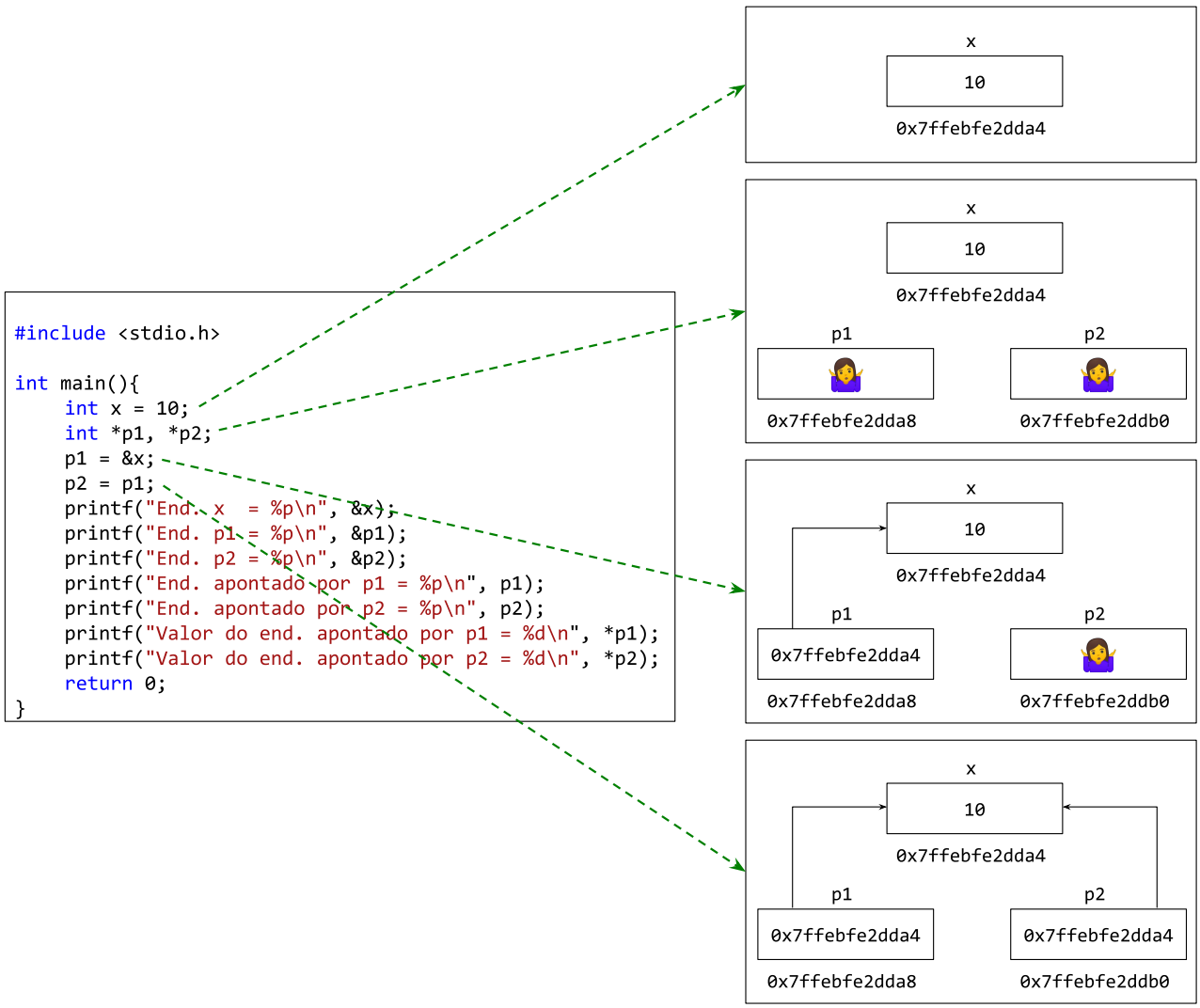

Figura 2: Exemplo do que acontece com os valores dos ponteiros e ao executar o código

Devemos sempre inicializar um ponteiro com algum endereço válido (usando alocação dinâmica ou atribuindo o endereço de uma variável). O Sistema Operacional (SO) é responsável por definir quais endereços de memória um programa pode ou não acessar. Dessa forma, ele deve impedir que um programa `malware` tente/consiga acessar o espaço de memória que está alocado para o navegador ou outro programa. Entretanto, muitos endereços de memória não estão alocados a nenhum programa (são memória livre) e, muitas vezes, não são controlados pelo SO (por questões de eficiência). Assim, é possível "brincar" com ponteiros não inicializados. Execute o código abaixo (`ex4.c`) no seu computador e veja o que acontece.

```c
#include <stdio.h>

int main(){
  int *p1;
  *p1 = 10; // Em qual endereço o valor 10 será armazenado?
  printf("End. p1 = %p\n", &p1);
  printf("End. apontado por p1 = %p\n", p1);
  printf("Valor do end. apontado por p1 = %d\n", *p1);
  return 0;
}
```

Abaixo é mostrado uma possível saída ao se executar o código anterior. Isso está certo? Por quê?

```
End.  p1 = 0x7fffdf30fd90
End.  apontado por p1 = 0x7fffdf30fe80
Valor do end.  apontado por p1 = 10
```

Abra o link `https://repl.it/@oberlan/ED1-Aula-Pratica-0-Exercicio-4` e execute o código. O Valgrind detectou alguma coisa errada?

Para evitar esse tipo de problema, podemos compilar o código como mostrado abaixo. Dessa forma, qualquer *warning:* será tratado como erro. Nas próximas aulas entenderemos melhor o que significa cada um desses argumentos que estão sendo passados para o `gcc`. Compile seu código como mostrado abaixo para ver o que acontece.

```
gcc main.c -o main -Wall -Werror -Wextra
```

Outra forma de evitar esse tipo de problema, é sempre inicializar um ponteiro com `NULL`, ou seja, `int *p1 = NULL`. Altere o código anterior, inicializando o ponteiro com `NULL`, compile e execute novamente o código. O que aconteceu?

Resumo

Ao se trabalhar com ponteiros, duas tarefas básicas serão sempre executadas:

- Acessar o endereço de memória de uma variável;
- Acessar o conteúdo de um endereço de memória.

Para realizar essas tarefas, vamos sempre utilizar apenas dois operadores: o operador "`*`" e o operador "`&`".

Operador "`*`" versus operador "`&`"

Operador	Operação	Significado
`*`	`int *x;`	Declara um ponteiro
	`int y = *x;`	Acessa o conteúdo para onde o ponteiro aponta
	`*x = 15;`	Altera o conteúdo para onde o ponteiro aponta
`&`	`&y`	Endereço onde uma variável está guardada na memória

A Figura 3 e a Figura 4 apresentam uma ilustração de como um ponteiro funciona internamente na memória. Notem que o ponteiro `m` está no endereço `0x7ffebfe2dda8` e aponta (valor interno do ponteiro) para o endereço `0x7ffebfe2dda4` (endereço da variável `count`).

Figura 3: Representação de um ponteiro internamente na memória

Figura 4: Representação de um ponteiro de forma didática

A tabala abaixo mostra o resultado de algumas operações considerando a Figura 3.

Tabela 1: Exemplos de operações com ponteiros

Operação	Significado	Resultado
`&count`	Endereço da variável `count`	`0x7ffebfe2dda4`
`&m`	Endereço da variável `m`	`0x7ffebfe2dda8`
`m`	Endereço da variável apontada por `m`	`0x7ffebfe2dda4`
`*m`	Valor da variável apontada por `m`	100

Acesso a atributos da estrutura por meio de apontadores

Outro operador associado a apontadores é a seta (->). Quando um apontador indica uma variável do tipo estrutura (`struct`), para que se acesse os atributos dessa variável, em vez da combinação do operador * com o operador ponto (.), utiliza-se ->. Observe o exemplo abaixo (`ex5.c`):

```c
#include <stdio.h>

typedef struct {
  char nome[30];
  int matricula;
} Aluno;

int main() {
  Aluno *x;
  Aluno y;

  x = &y;
  x->matricula = 2031;
```

```
14    printf("%d\n", y.matricula);
15
16    return 0;
17  }
```

Note que, no código, x é um apontador para uma estrutura `Aluno` e y é uma variável `Aluno`. Após fazer x apontar para o endereço de y, para acessar a matrícula de y através de x, foi utilizado o operador `->`. Já para exibir o atual valor da matrícula de y, que passou a ser 2031, bastou utilizar o ponto.

O operador `->` é uma sintaxe alternativa muito utilizada por sua simplicidade. Seria possível também utilizar o acesso ao conteúdo seguido do ponto. Na linha 13 do código anterior, a sintaxe equivalente poderia ser como mostrado abaixo:

```
1  (*x).matricula = 2031;
```

Uso de ponteiros nas passagens de parâmetros

O conceito de função em programação é fundamental. Uma das principais vantagens dos ponteiros, no que diz respeito às funções em C, é a passagem de valores sem cópia (passagem por referência).

Na passagem por cópia, as variáveis passadas como argumentos da função têm seus valores copiados para uma região de memória criada especificamente para a execução da função. São essas cópias que a função utiliza no decorrer da sua execução. Assim, qualquer alteração dos valores das cópias não implicará em mudança nas variáveis originais.

Por meio de apontadores, é possível alterar os valores das variáveis passadas como argumentos para uma função. Ao declarar um parâmetro de uma função como ponteiro, deve-se passar o endereço de uma variável como argumento correspondente. Nesse tipo de passagem, não é copiado o valor da variável argumento, mas, com o endereço dela, pode-se acessar o conteúdo e modifica-lo no decorrer da execução da função. O exemplo abaixo (`ex6.c`) mostra a passagem por cópia e por ponteiro.

```
1  #include <stdio.h>
2
3  void adicionaX1(int x, int b) {
4      b = b + x;
5  }
6  void adicionaX2(int x, int *b) {
7      *b = *b + x;
8  }
9  int main() {
10     int a = 0;
11     adicionaX1(10, a);
12     printf("a = %d\n", a);
13     adicionaX2(10, &a);
14     printf("a = %d\n", a);
15     return 0;
16 }
```

Em `adicionaX1`, no código anterior, uma cópia do valor de `a` é passada com argumento da função, ou seja, `b` recebe uma cópia do valor `0` de `a`. Dentro da função, o valor de `b` muda para `10`, mas o de `a` continua o mesmo. Assim, o primeiro valor impresso é `0`. Já em `adicionaX2`, o endereço de `a` é passado para o apontador `b` e, na execução da função, é adicionado `x` ao conteúdo da variável apontada por `b`, o que faz a ter seu valor modificado para `10`. Portanto, o que é apresentado no segundo `printf` é `a = 10`. Execute o código presente em `https://repl.it/@oberlan/ED1-Aula-Pratica-0-Exercicio-6` para ficar mais claro.

Exercício 3: Um aluno fez o código abaixo (`ex7.c`) com a intenção de implementar uma função que troca o valor de duas variáveis. Implemente, compile e execute o código. Ele funciona? Por quê? Se o código não funcionar como esperado, faça as modificações necessárias para que ele funcione.

```c
#include <stdio.h>

void troca(int x, int y) {
    int aux;
    aux = x;
    x = y;
    y = aux;
}

int main() {
    int a, b;
    a = 10;
    b = 20;
    printf("Antes da troca\n");
    printf("a = %d\n", a);
    printf("b = %d\n", b);

    troca(a, b);

    printf("Depois da troca\n");
    printf("a = %d\n", a);
    printf("b = %d\n", b);

    return 0;
}
```

Exercício 4: Abra o arquivo `ex8.c` e implemente as função `le_array`, `imprime_array` e `troca` (feita no exercício anterior). A função `le_array` deve preencher o array `a` com `n` números inteiros (lembre-se que array é sempre passado como referência). Já função `imprime_array` deve imprimir os valores do array passado como parâmetro. O que a função `funcao_misteriosa` faz?

Exercício 5: Caso não tenha lido, leia os links presentes em “Material complementar” da “Semana 0”.