



Unidade 6

Ponteiros

DEC0012 - Linguagem de Programação I
Curso de Engenharia de Computação
Profa. Andréa Sabedra Bordin

Introdução

Cada variável alocada em memória tem um endereço específico que pode ser acessado com operador (&).

Esse operador, aplicado ao nome de uma variável, indica que se trata de um endereço na memória daquela variável.

Introdução

O exemplo a seguir mostra como podemos acessar o endereço de uma variável usando o operador **&**.

Exemplo 1: operador &

```
1  #include <stdio.h>
2
3  int main ()
4  {
5      int var1;
6      char var2[10];
7
8      printf("Address of var1 variable: %p \n", &var1 );
9      printf("Address of var1 variable: %p \n", &var2 );
10
11     return 0;
12 }
```

Address of var1 variable: 0x7ffe4ee3a388

Address of var2 variable: 0x7ffe4ee3a38e

Os endereços da memória de computador são representados em formato **hexadecimal**. Portanto, para imprimir o endereço de uma variável usando função **printf()** precisamos de um especificador correspondente (**%p** indica *pointer address*).

Introdução

Um ponteiro (apontador == *pointer*) é uma **variável** capaz de armazenar um **endereço de memória de outra variável**, sendo declarado com a seguinte sintaxe:

Tipo_de_dado *Identificador;

Esta declaração pode ser assim entendida:

Identificador é capaz de armazenar o **endereço** de uma variável de tipo **Tipo_de_dado**.

Por exemplo, a declaração:

int *p;

Indica que **p** é uma variável capaz de armazenar o endereço de uma variável do tipo int.

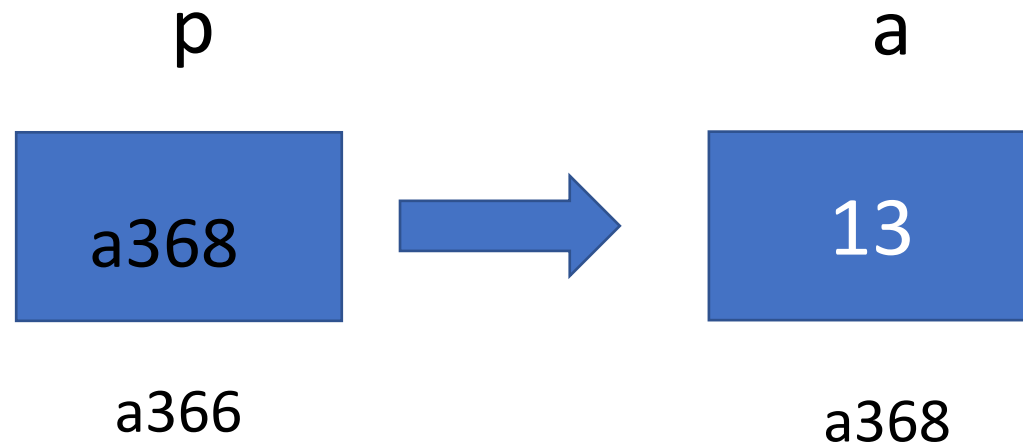
p é uma variável do tipo ponteiro.

p aponta para um inteiro.

Introdução

Ponteiro é uma variável que armazena o endereço da memória de uma outra variável.

```
int *p;  
int a = 13;  
*p = &a;
```



Representação na memória

Código de exemplo

// Declaração e manipulação de ponteiros

```
int main() {
```

```
    int a = 10;
```

```
    printf("Valor inicial da variável a: %d \n", a);
```

```
    int* b = &a; //declaração da variável ponteiro b e atribuição do endereço da variável a
```

```
    *b = 11; //atribuição do valor 11 à variável que o ponteiro b aponta
```

```
    printf("Valor da variável a: %d \n", a);
```

```
    printf("Endereço da variável a: %p \n", &a);
```

```
    printf("Valor apontado por b (conteúdo): %d \n", *b);
```

```
    printf("Endereço apontado por b: %p \n", b);
```

```
    printf("Endereço do ponteiro b: %p", &b);
```

```
    return 0;
```

```
}
```

```
Valor inicial da variável a: 10
Valor final da variável a: 11
Endereço da variável a: 0x7ffffb82d562c
Valor apontado por b (conteúdo): 11
Endereço apontado por b: 0x7ffffb82d562c
Endereço do ponteiro b: 0x7ffffb82d5630
```

Código de exemplo

```
#include <stdio.h>
main()
{
    float *a, *b;
    float Aux, x, y;
    x = 1;                /* o conteúdo de x agora é igual a 1 */
    y = 2;                /* o conteúdo de y agora é igual a 2 */
    a = &x;               /* a aponta para x */
    b = &y;               /* b aponta para y */
    Aux = *a;             /* o conteúdo de Aux agora é 1 (conteúdo de x) */
    *a = *b;              /* o conteúdo de x agora é 2 (conteúdo de y) */
    *b = Aux;             /* o conteúdo de y agora é 1 */
    printf("x = %f e y = %f\n", x, y);
}
```

Se p é um ponteiro, a indicação *p num programa acessa o **conteúdo da variável** para a qual p aponta

Aplicação

Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras, digamos i e j.

```
void troca (int i, int j) {  
    int temp;  
    temp = i;  
    i = j;  
    j = temp;  
}
```

É claro que a função acima não produz o efeito desejado, pois recebe apenas os valores das variáveis e não as variáveis propriamente ditas.

A função recebe **cópias** das variáveis e troca os valores dessas cópias, enquanto as variáveis originais permanecem inalteradas.

Passagem de parâmetros por referência

Para obter o efeito desejado, é preciso passar à função os *endereços* das variáveis:

...

```
troca(&x, &y);
```

...

```
void troca(float *a, float *b) {  
    float Aux;  
    Aux = *a;  
    *a = *b;  
    *b = Aux;  
}
```

A utilização de ponteiros como **parâmetros de funções** permite a **passagem de parâmetros por referência**.

Ponteiros do tipo **NULL**

Boa prática de programação:

Atribuir o valor **NULL** para as variáveis do tipo ponteiro ajuda evitar os resultados inesperáveis do programa.

A palavra-chave **NULL** indica que o ponteiro não aponta para nenhum lugar específico da memória.

```
int *ptr;  
ptr = NULL;  
printf("%p", ptr);
```



Endereço do ponteiro ptr

Operações aritméticas com ponteiros

O valor armazenado em um ponteiro é um **endereço de memória** que é um **número**. Por isso é possível realizar as operações aritméticas com ponteiros: **+**, **-**, **++** e **--**.

Essas operações serão executadas considerando o tipo de dados para qual um ponteiro específico pode apontar.

Operações aritméticas com ponteiros

Por exemplo, se temos um ponteiro **iPtr** que aponta para variável do tipo **int**:

```
int * iPtr;
```

Vamos considerar a seguinte situação:

- o endereço armazenado no ponteiro **iPtr** é **1000**
- o tamanho de um **int** no nosso sistema é de **4 bytes**

Depois de executar o comando:

```
iPtr++; //iPtr = iPtr+1
```

o endereço armazenado em **iPtr** será **1004**.

Operações aritméticas com ponteiros

Agora vamos considerar uma situação diferente:

- o ponteiro **cPtr** aponta para variáveis do tipo **char**
- o endereço armazenado no ponteiro **cPtr** é **1000**
- o tamanho de um **char** é de **1** byte

```
char *cPtr;
```

Nesse caso, depois de executar o comando:

```
cPtr++;
```

o endereço armazenado em **cPtr** será **1001**.

Ponteiros e vetores

Em C o **nome de um vetor** é, na verdade, é um ponteiro para **primeiro elemento do vetor**. Então a declaração:

```
int v[4];
```

Significa que **v** é um ponteiro para **&v[0]**.

A declaração a seguir atribui para o ponteiro **ptr** o endereço do **primeiro elemento** do vetor **v**.

```
int *ptr;  
int v[4];  
ptr = v;
```

Ponteiros e vetores

Atribuições de valores entre os **ponteiros** e **nomes dos vetores** são válidas e permitidas pela linguagem.

No exemplo anterior, uma vez que o endereço do primeiro elemento do vetor **v** é armazenado em ponteiro **ptr** todos os elementos do vetor **v** podem ser acessados usando:

***ptr // primeiro elemento do vetor**

***(ptr+1) // segundo elemento do vetor**

***(ptr + 2) //terceiro elemento do vetor**

... .

Assim sendo, o elemento da posição **v[3]** do vetor pode ser acessado como ***(v + 3)**

Ponteiros e vetores

```
int *ptr;  
int v[3];  
ptr = v;  
*ptr = 1;  
*(ptr+1) = 2;  
*(ptr+2) = 3;  
printf("v[0] = %d \n", *ptr);  
printf("v[1] = %d \n", *(ptr+1));  
printf("v[2] = %d \n", *(ptr+2));
```

Qual o resultado da execução deste programa ?

Ponteiros e vetores

```
int *ptr;  
int v[4]={10,20,30,40};  
ptr = v;  
for (int i=0;i<4;i++) {  
    printf("v[%d] = %d \n",i, *ptr);  
    ptr++;  
}
```

Qual o resultado da execução deste programa ?

Ponteiro para *struct*

```
#include <stdio.h>
//declaração da struct pessoa
typedef struct {
    char nome[100];
    int idade;
} pessoa;

main() {
    pessoa p1; //variável p1 do tipo pessoa
    pessoa *p2 = &p1 //ponteiro *p2 recebe o endereço de p1
    strcpy(p1.nome, "joao da silva");
    p1.idade = 20;
    printf("%s, %d\n", p2->nome, p2->idade); //acesso via ponteiro
    p2->idade = 18; //acesso via ponteiro
    printf("%s, %d\n", p1.nome, p1.idade);
}
```

Alocação dinâmica de memória

- Em muitas aplicações, a **quantidade de memória a alocar** só se torna conhecida ***durante a execução*** do programa.
- Para lidar com essa situação é preciso recorrer à **alocação *dinâmica* de memória**.
- **A alocação dinâmica** permite ao programador alocar memória para novas variáveis quando o programa está sendo executado, e não apenas quando se está escrevendo o programa.
- Quantidade de memória é alocada sob demanda, ou seja, quando o programa precisa.

Alocação dinâmica de memória

- Menos desperdício de memória.
- Espaço é reservado até liberação explícita.
- Depois de liberado, estará disponibilizado para outros usos e não pode mais ser acessado.
- Espaço alocado e não liberado explicitamente é automaticamente liberado ao final da execução.

Alocação dinâmica de memória

A alocação dinâmica é realizada pelas funções **malloc**, **realloc** e **free**, que estão na biblioteca **stdlib**.

Para usar essa biblioteca, inclua a correspondente interface no seu programa:

```
#include <stdlib.h>
```

Alocação dinâmica de memória

A função **malloc** (*memory allocation*) aloca espaço para um bloco de bytes consecutivos na memória RAM do computador e devolve o endereço desse bloco.

O número de bytes é especificado no argumento da função. No seguinte fragmento de código, `malloc` aloca 1 byte:

```
char *ptr;  
ptr = malloc (1);  
scanf ("%c", ptr); //leitura do caracter
```

O endereço devolvido por `malloc` é do tipo genérico **void ***. O programador armazena esse endereço em um ponteiro de tipo apropriado.

Alocação dinâmica de memória

No exemplo anterior, o endereço é armazenado no ponteiro **ptr**, que é do tipo ponteiro-para-**char**.

A transformação do ponteiro genérico em ponteiro-para-**char** é **automática**; não é necessário escrever:

```
ptr = (char *) malloc (1);
```

Alocação dinâmica de memória

A fim de alocar espaço para um objeto que precisa de mais que 1 byte, convém usar o **operador** `sizeof`, que diz quantos bytes o objeto em questão tem.

Veja exemplo no código a seguir:

Alocação dinâmica de memória

```
int main() {  
  
    char *c = malloc(sizeof(char));  
    int *i = (int *)malloc(sizeof(int));  
    float *f = (float *)malloc(sizeof(float));  
    long *l = (long *) malloc(sizeof(long));  
    double *d = (double *) malloc(sizeof(double));  
  
    int iA = 3;  
    i=&iA;  
    printf("Valor da variavel inteira: %d\n", *i);  
  
    float fA = 3.5;  
    f=&fA;  
    printf("Valor da variavel float: %f\n", *f);  
  
    long lA = 360000000;  
    l=&lA;  
    printf("Valor da variavel long: %ld\n", *l);  
  
    double dA = 3.500000;  
    d=&dA;  
    printf("Valor da variavel double: %lf\n", *d);  
  
    return 0;  
}
```

alocando memória para
tipos de dados

Alocação dinâmica de memória - Vetor

Para armazenar um vetor (*array*) o compilador C calcula o tamanho, em bytes, necessário e reserva posições sequenciais na memória.

Note que isso é muito parecido com alocação dinâmica.

Existe uma ligação muito forte entre ponteiros e *arrays*. O nome do *array* é apenas um ponteiro que aponta para o primeiro elemento do *array*.

Alocação dinâmica de memória - Vetor

```
main() {  
    // alocando um vetor com 3 inteiros  
    int *v = (int*) malloc( 3 * sizeof(int) );  
    // se v não for 0 (nulo)  
    if (v) {  
        v[0] = 10;  
        v[1] = 20;  
        v[2] = 30;  
        printf("%d %d %d\n", v[0], v[1], v[2]);  
    }  
}
```

Alocação dinâmica de memória - Vetor

```
main() {  
    // alocando um vetor com 3 inteiros  
    int *v = (int*) malloc( 3 * sizeof(int) );  
  
    for (int i=0;i<3;i++){  
        printf("Entre com valor de %d: ", i);  
        scanf("%d", &v[i]);  
    }  
  
    printf("%d %d %d\n", v[0], v[1], v[2]);  
}
```

Alocação dinâmica de memória - *struct*

```
typedef struct {  
    int dia, mes, ano;  
} data;
```



1 struct

```
data *d;  
d = malloc (sizeof (data));  
d->dia = 31;    // (*d).dia = 31  
d->mes = 12;  
d->ano = 2016;  
printf("%d/%d/%d", d->dia, d->mes, d->ano);
```

Alocação dinâmica de memória - *struct*

```
typedef struct {
    int dia, mes, ano;
} data;

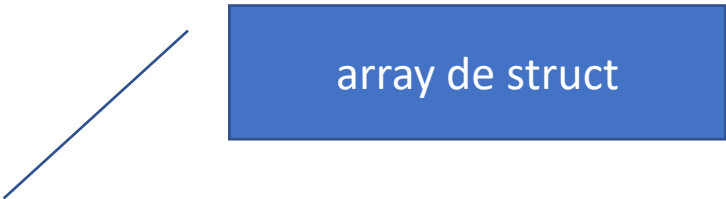
main() {
    data *d;
    d = malloc (3 * sizeof (data)); //aloca espaço para 3 struct

    d[0].dia = 31;
    d[0].mes = 12;
    d[0].ano = 2016;

    d[1].dia = 31;
    d[1].mes = 12;
    d[1].ano = 2017;

    d[2].dia = 31;
    d[2].mes = 12;
    d[2].ano = 2021;

    for (int i = 0; i < 3 ; i++){
        printf("%d/%d/%d \n", d[i].dia, d[i].mes, d[i].ano);
    }
}
```



array de struct

Alocação dinâmica de memória - *struct*

```
typedef struct {  
    int dia, mes, ano;  
} data;
```

//função que aloca espaço para um struct data e RETORNA o endereço para outro ponteiro

```
data* novoRegistro() {  
    data *d;  
    d = malloc (sizeof (data));  
    printf("%p", d);  
    return d;  
}
```

```
main() {  
    data *d1 = novoRegistro();  
    d1->dia = 31;  
    d1->mes = 12;  
    d1->ano = 2016;  
  
    data *d2 = novoRegistro();  
    d2->dia = 31;  
    d2->mes = 12;  
    d2->ano = 2017;  
}
```

Alocação dinâmica de memória

As **variáveis alocadas dinamicamente** continuam a existir mesmo depois que a execução da função termina.

Se for necessário liberar a memória ocupada por essas variáveis, é preciso recorrer à função `free`. A função `free` desaloca a porção de memória alocada por `malloc`.

A instrução `free(ptr)` avisa ao sistema que o bloco de bytes apontado por `ptr` está disponível para reciclagem.

Nunca aplique a função `free` a uma **parte de um bloco** de bytes alocado por `malloc` (ou `realloc`).

Aplique `free` apenas ao bloco todo.

Alocação dinâmica de memória