Técnicas de Programação II

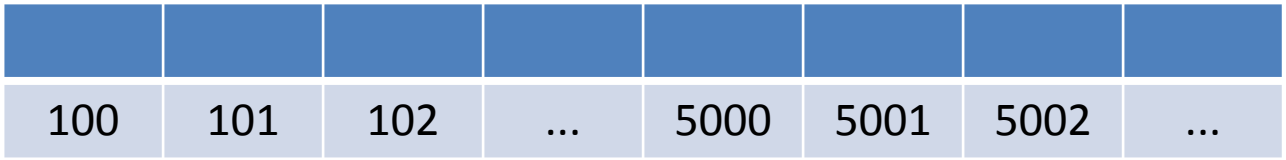
Ponteiros

Conceitos básicos

- Como se sabe, tudo o que acontece em um computador (ou quase tudo) se passa em memória RAM.
- É na memória RAM que são carregados os nossos programas, jogos, editor de texto, etc.
- É também na RAM que são armazenadas as variáveis que fazem parte dos nossos programas.

Conceitos básicos

- De fato, a memória RAM pode ser vista como um enorme vetor de *bytes* consecutivos.



- Cada um desses *bytes* ocupa uma posição bem determinada em memória que é identificada por um número único que varia entre 0 e a totalidade de *bytes* que ocupam a memória RAM do computador.

Conceitos básicos

- Sempre que declaramos uma variável temos que indicar qual o seu tipo e qual o seu nome:

ch

100	101	102	...	5000	5001	5002	...

- A variável *ch* ocupa, no exemplo acima, o *byte* de memória 5000.
- Pra nós, programadores, é muito simples referenciar uma variável pelo seu nome do que referenciá-la pela posição que essa variável ocupa em memória

Conceitos básicos

- Repare que quando se faz:

  ch = 'A';

ch

100	101	102	...	5000	5001	5002	...

Está se dizendo ao computador para ir à posição 5000 de memória e colocar lá o caracter 'A'. É o compilador que realiza esta conversão por nós. Apenas temos que nos preocupar em escrever o programa corretamente.

Conceitos básicos

- Suponhamos que tínhamos ao nosso dispor um ponteiro denominado **ptr**. Como qualquer variável, **ptr** ocupa uma posição de memória.

	ptr			**ch**			
				'A'			
100	101	102	…	5000	5001	5002	…

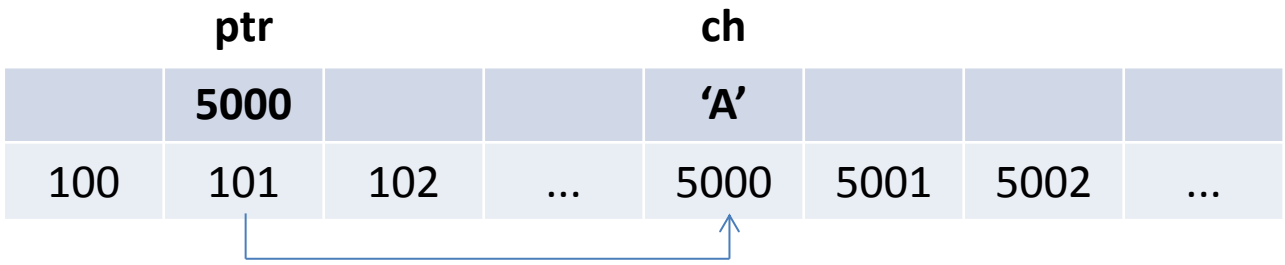
- Ora, *ptr* é um ponteiro, por isso deverá conter o endereço de memória de outra variável. Note que o endereço de uma variável é o número da casa que ocupa em memória.

Conceitos básicos

Por isso, para colocarmos *ptr* apontando para variável *ch* bastará colocar o endereço de *ch* em *ptr*. Isso se faz utilizando o operador **& (endereço de)**.

ptr = & ch;

A variável *ptr* fica, assim, com o valor 5000.



Isto é, *ptr* aponta para a variável *ch.*

Declaração de ponteiros

- Um conceito que não pode se pode perder de vista é que um ponteiro é uma variável como outra qualquer. O seu objetivo é armazenar o endereço de outra variável, o qual é, por sua vez um número.

Declaração de ponteiros

- Se uma variável do tipo ponteiro é capaz de armazenar um número (o endereço de outra variável), então terá que ocupar algum espaço de memória. Tem por por isso de ser declarada tal como qualquer outra variável.
- Sua sintaxe de declaração é:

tipo * ptr

onde:

ptr – é o nome da variável do tipo ponteiro.

tipo – é o tipo da variável para a qual apontará.

* – indica que é uma variável do tipo ponteiro.

Declaração de ponteiros

- A declaração de ponteiros pode ser realizada no meio de outras variáveis do mesmo tipo.
- Exemplo:

```
char a, b, *p, c, *q;
int idade, *p_idade;
```

Declaração de ponteiros

- **Pergunta:** Onde deve ser colocado (encostado) o asterisco na declaração de um ponteiro?

```
int * p; // Asterisco separado do tipo e da variável
int* p; // Asterisco junto ao tipo
int *p; // Asterisco junto à variável
```

- **Resposta Oficial:** Tanto faz. As três declarações são equivalentes.
- **Porém cuidado!** Quando numa declaração: `int* x, y, z;`
- Apenas a variável **x** é um ponteiro, as demais são inteiros normais.

Carga inicial de ponteiros

- A carga inicial de ponteiros se faz por meio do operador **Endereço de: &.**
- Tal operador pode também ser utilizado para iniciar uma variável do tipo ponteiro no momento de sua declaração.
- Exemplo:

```
int x = 5;
float pi = 3.14;
int *ptr_x = &x;
float *pointer_to_pi = &pi;
```

Carga inicial de ponteiros

Uma vez declarado um ponteiro, podem ser realizados sobre ele praticamente todos os tipos de operações que podemos realizar sobre inteiros. No entanto um ponteiro serve, sobretudo, para permitir acessar outros objetos através de seus endereços.

int a=5, b=7;

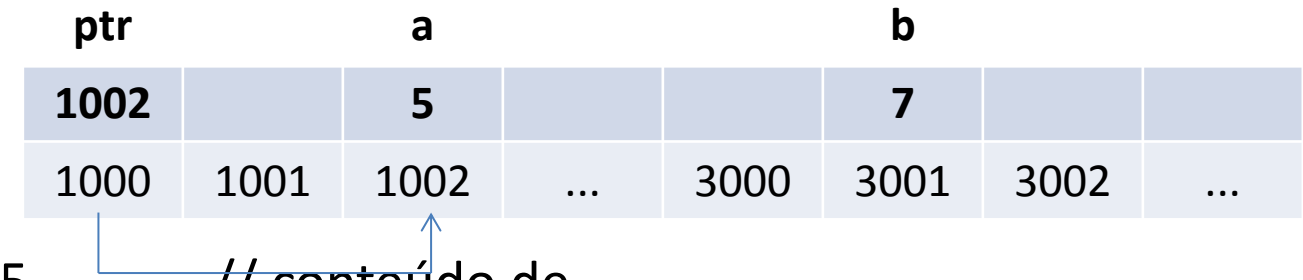
int *ptr; // *ptr aponta para nada

ptr		**a**			**b**		
'lixo'		**5**			**7**		
1000	1001	1002	...	3000	3001	3002	...

para colocar *ptr* apontando para *a* faz-se

ptr = &a;

Carga inicial de ponteiros



a=5 → 5 // conteúdo de

ptr → 1002 // endereço de a

*ptr → 5 // conteúdo que está no endereço 1002

Dessa forma, fazer

printf(“%d”, a);

é equivalente a

printf(“%d”, *ptr)

Pois *ptr* aponta para *a* e *ptr corresponde ao conteúdo de *a.*

Ponteiros e tipos de dados

- Um variável do tipo ponteiro sempre aponta para uma variável do mesmo tipo e ocupa o mesmo número de bytes que o tipo ocupa.
- Nunca aponte uma variável do tipo ponteiro de um tipo para outro tipo. Ocorrerão erros inevitáveis.

```
char *pch, ch;
pch = &ch;
float *pf, p;
pf = &pf;
int *ptr; p;
ptr = &p;
```

Ponteiros e Vetores

- Os ponteiros também são normalmente utilizados no tratamento e na manipulação de vetores e *strings.*
- O nome de um vetor corresponde ao endereço do seu primeiro elemento, isto é, se v for um vetor

  v == &v[0].
- Isto significa que o nome de um vetor não é mais do que o endereço do primeiro elemento desse vetor.
- Se o nome de um vetor é um endereço, então é um número, isto é, o nome de um vetor é um ponteiro para o primeiro elemento desse vetor. Entretanto, é um ponteiro constante.

Ponteiros e Vetores

```
int v[3] = {10, 20, 30};     // vetor com 3 inteiros
int *ptr;                    // ponteiro para inteiro
```

Existem duas formas de colocar o ponteiro **ptr** apontando para o primeiro elemento de **v**:

1. `ptr = &v[0];// ptr fica com endereço do 1º elemento`
2. `ptr = v;                 // pois v == &v[0]`

- Ao contrário de **v**, que é um vetor (ponteiro constante associado à sua própria memória), **ptr** é um ponteiro puro, sendo assim pode ser alterado com endereços diferentes ao longo da execução de um programa, não estando obrigado a apontar, eternamente, para o primeiro elemento do vetor, como acontece com **v**.

Ponteiros e Vetores

Pode, assim, apontar para cada um dos elementos do vetor de **v**.

```
int v[3] = {10, 20, 30};        // vetor com 3 inteiros
int *ptr;                       // ponteiro para inteiro

ptr = v;                        // passa a apontar para o 1º elemento
printf("%d %d\n", v[0], *ptr);          → 10 10

ptr = &v[2];
printf("%d %d\n", v[2], *ptr);          → 30 30
```

- Os elementos de um vetor ocupam posições consecutivas de memória, sendo o nome do vetor igual ao endereço do primeiro elemento, isto é, o menor endereço do vetor.

Aritmética de ponteiros

- Sendo os ponteiros números que representam posições de memória, podem ser realizadas algumas operações aritméticas (incremento, decremento, diferença e comparação) sobre eles.

- Essas operações podem, no entanto, ser um pouco diferentes a que estamos habituados, mas elas só servem para nos facilitar a vida.

Aritmética de ponteiros - incremento

- Um ponteiro pode ser incrementado como qualquer variável. No entanto, o incremento de uma unidade não significa que o endereço anteriormente armazenado no ponteiro seja incrementado em um *byte.*

- Na realidade, se **ptr** é um ponteiro para um determinado tipo, quanto **ptr** é incrementado, por exemplo, de uma unidade, o endereço que passa a conter é igual ao endereço anterior de **ptr + sizeof(tipo)** para que o ponteiro aponta, isto é, o ponteiro avança não um *byte*, mas sim a dimensão do tipo do objeto para o qual aponta.

Aritmética de ponteiros - incremento

```
#include <stdio.h>
Int main()
{
        int x=5, *px = &x;
        float y=5.0, *py=&y;
        printf("%d %ld\n", x, (long) px);
        printf("%d %ld\n", x+1, (long) (px+1));

        printf("%f %ld\n", y, (long) py);
        printf("%f %ld\n", y+1, (long) (py+1));
}
```

5 1211048978
6 1211048980
5.000000 1211048970
6.000000 1211048974

Aritmética de ponteiros - incremento

5 1211048978
6 1211048980
5.000000 1211048970
6.000000 1211048974

os endereços são transformados em um *long int* para melhor compreensão do resultado.

Na operação de incremento, podem-se utilizar os operadores normais:

```
ptr++;
ptr = ptr + 2;
ptr += 4; // se ptr apontar para um float avança 4*4=16 bytes
```

Aritmética de ponteiros - decremento

- O decremento de ponteiros funciona da mesma forma que o incremento.

- Um ponteiro para o tipo **xyz** sempre sizeof(xyz) *bytes* por unidade de decremento.

- Exemplo: escreva um programa que mostre uma *string* na tela pela ordem em que está escrita e pela ordem contrária.

Aritmética de ponteiros - decremento

```
#include <stdio.h>
Int main()
{
        char s[100];
        char *ptr = s;          // aponta para o primeiro caractere de s
        printf("Introduz uma String: ");
        gets(s);
        if (*ptr == '\0')
          return 0;             // string vazia
        // imprimir a string normalmente
        while (*ptr != "\0)
            putchar(*ptr++);
        // imprimir a string ao contrario
        ptr--;      // por causa do '\0'
        while (ptr >=s) // enquanto ptr for >= que &s[0]
            putchar(*ptr--);
}
```

Aritmética de ponteiros - diferença

- A operação de diferença entre dois ponteiros para elementos do mesmo tipo permite saber quantos elementos existem entre um endereço e outro.

- Por exemplo, o comprimento de uma *string* pode ser obtido através da diferença entre o endereço do caractere '\0' e o endereço do caractere original.

- **A diferença entre ponteiros só pode ser realizada entre ponteiros do mesmo tipo**

Aritmética de ponteiros - diferença

```
#include <stdio.h>
int strlen(char *s);
int main()
{
        char s[100];
        char *ptr = s; // aponta para o primeiro caractere de s
        printf("Digite uma string: ");
        gets(s);
        printf("%d\n",strlen(s);
}

int strlen(char *s)
{
        char *ptr = s;          // guardar o endereço inicial
        while (*s != '\0')      // enquanto não for fim de string
                s++;
        return (int) (s – ptr); // retorna a diferença entre os endereços
}
```

Aritmética de ponteiros - comparação

- É também possível a comparação de dois ponteiros para o mesmo tipo, utilizando os operadores relacionais (<, <=, >, >=, == e !=).

- **A diferença e a comparação entre ponteiros só podem ser realizadas entre ponteiros do mesmo tipo.**

Resumo das Operações sobre Ponteiros

Operação	Exemplo	Observações
Atribuição	ptr = &x	Podemos atribuir um valor (endereço) a um ponteiro. Se quisermos que aponte para nada podemos atribuir-lhe o valor da constante NULL.
Incremento	ptr = ptr + 2	Incremento de 2 * sizeof(tipo) de ptr.
Decremento	ptr = ptr – 10	Decremento de 10 * sizeof(tipo) de ptr.
Apontado por	*ptr	O operador asterisco permite obter o valor existente na posição cujo endereço está armazenado em ptr.
Endereço de	&ptr	Tal como qualquer outra variável, um ponteiro ocupa espaço em memória. Dessa forma podemos saber qual o endereço que um ponteiro ocupa em memória.
Diferença	ptr1 – ptr2	Permite-nos saber qual o número de elementos entre ptr1 e ptr 2
Comparação	ptr1 > ptr2	Permite-nos verificar, por exemplo, qual a ordem de dois elementos num vetor através do valor dos seus endereços.

Resumo das Operações sobre Ponteiros

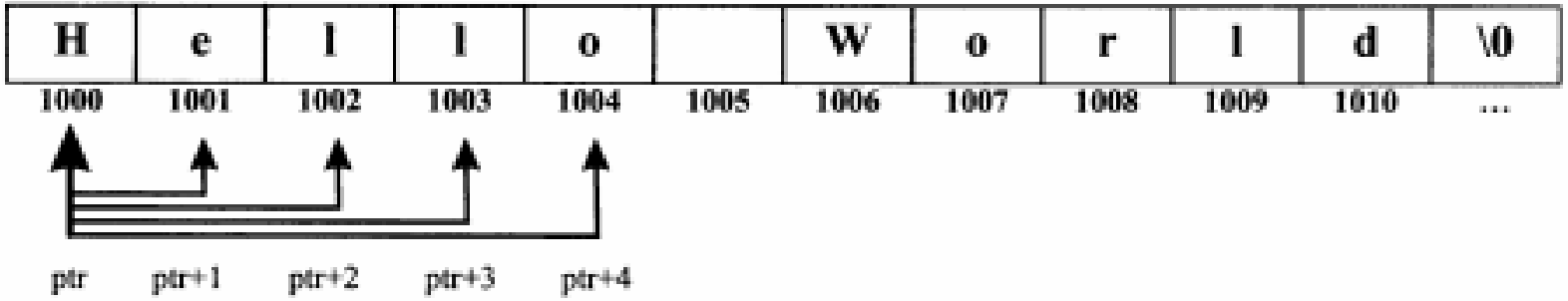
- Sendo o nome de um vetor o endereço do seu primeiro elemento, poderemos com ele realizar todas as operações a que temos acesso quando manipulamos ponteiros, desde que essas operações não alterem o seu valor, pois o nome de um vetor é uma constante.

- Exemplos:

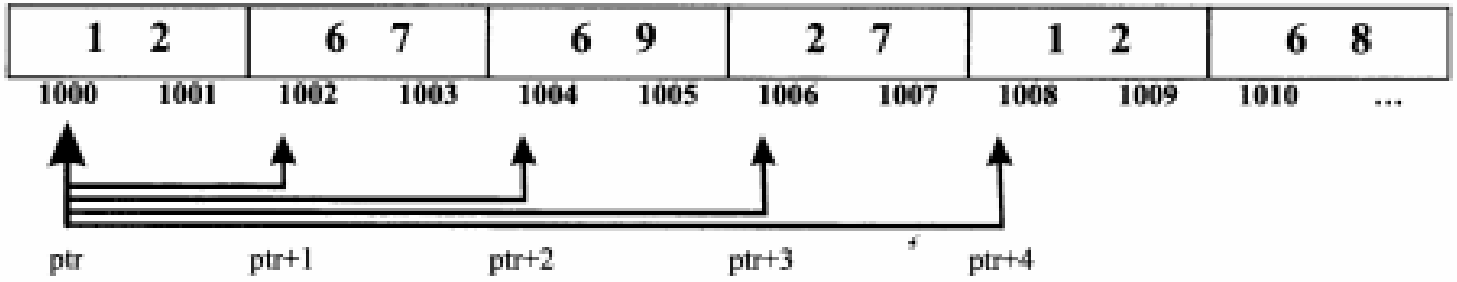
```
char s[20] = “Ola”;
s = “ole”;          // erro de compilação, usar strcpy
s++;                // erro: não podemos alterar s
s+1;                // OK: não estamos alterando s
*s;                 // OK
(*s)++;             // OK: não estamos alterando s
s = s – 2           // Erro: não podemos alterar s
s > s + 1;          // OK
s = 1 – s;          // OK
```

Resumo das Operações sobre Ponteiros

Se ptr for um ponteiro para um vetor de caracteres, então

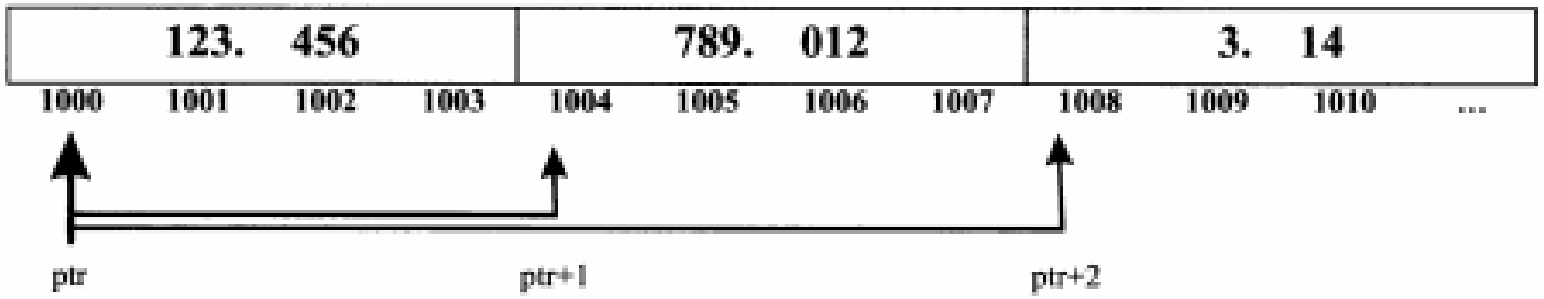


Se ptr for um ponteiro para um vetor de inteiros (dois *bytes* cada), então



Resumo das Operações sobre Ponteiros

Se ptr for um ponteiro para um vetor de *floats* (quatro *bytes* cada), então



- Como se pode verificar pelos esquemas apresentados, cada ponteiro se move o número de *bytes* que ocupa o tipo para o qual aponta.

- Dessa forma, evita-se que o programador tenha de indicar qual o número exato de *bytes* que o ponteiro deve avançar ou retroceder, sendo apenas necessário indicar o número de elementos a avançar.

Ponteiros e Vetores – acesso aos elementos

```
char  s[] = “OlaOleOli”;
char *ptr = s;                    // *prt fica com o &s[0]
```

- Como podemos acessar o caractere ‘a’ presente na string?

S[2] caractere existente na posição 2 do vetor

*(ptr+2) como ptr contém o endereço do primeiro caractere, se lhe adicionarmos 2 obtermos o endereço do caractere ‘a’. Para obter o caractere ‘a’ basta usar o operador *(apontado por).

*(s+2) Se s == &s[0] pode –se usar a mesma estratégia que foi usada no exemplo anterior.

Ptr[2] o endereçamento de elementos através de colchetes pode ser realizado também por ponteiros, como se tratasse de um vetor.

Ponteiros e Vetores – acesso aos elementos

vetor[0] == *(vetor)
vetor[1] == *(vetor + 1)
vetor[2] == *(vetor + 2)
- - - - - -
vetor[n] == *(vetor + n)

Pois o nome de um vetor é um ponteiro para o primeiro elemento desse mesmo vetor, o qual poderá ser acessado por *vetor. Em relação aos outros elementos, bastará utilizar os conceitos de aritmética de ponteiros, posicionando-se no endereço do elemento pretendido, através de somas ou subtrações e utilizando o operador ***(Apontado por)** para obter o conteúdo dessa posição do vetor.

Ponteiros – considerações finais

1. Um ponteiro é uma variável que não tem memória própria associada (apenas possui o espaço para conter um endereço), apontando normalmente para outros objetos já existentes.
2. Embora seja possível utilizá-los como vetores, os ponteiros não possuem memória própria. Só se pode utilizar o endereçamento através de um ponteiro depois que este está apontando para algum objeto existente.
3. Não se deve fazer cargas iniciais de objetos apontados por um ponteiro que ainda não tenha sido iniciado.

Exemplo:

```
int *p;       // p fica com lixo no seu interior – aponta para algum lugar
*p = 234;     // vamos colocar 234 no local para onde p aponta. Pode-se perder dados
```

4. Por segurança, inicie sempre os seus ponteiros. Se não souber para onde apontá-los, inicie-os com NULL.
5. Nunca se deve confundir ponteiros com vetores sem dimensão. Se não sabemos qual a dimensão de que necessitamos, como o compilador poderá saber?
6. Em uma declaração de um ponteiro com carga inicial automática **int *p = NULL;** é o ponteiro **p** que é iniciado, e não ***p**, embora a atribuição possa por vezes sugerir o contrário.