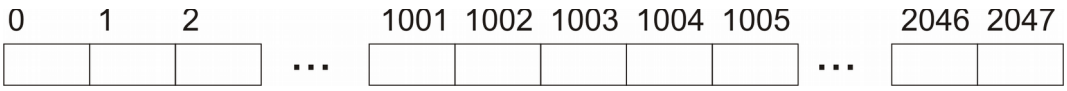
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
PRÓ-REITORIA DE ENSINO DE GRADUAÇÃO
Rua Dom Manoel de Medeiros, s/n – Dois Irmãos 52171-900 Recife-PE
Fone: 0xx-81-332060-40 proreitor@preg.ufrpe.br

Disciplina: Estrutura de Dados I
Professor: Igor M. Vanderlei

P O N T E I R O S

Ponteiros
• É uma das ferramentas mais poderosas da linguagem C *(para muitos, também é considerado a mais difícil).*
• Não é comum a existência de ponteiros em outras linguagens de programação. Ex. Java, Basic, Delphi.

Funcionamento da memória do computador.
A memória do computador consiste em milhares de bytes, onde cada byte é identificado pelo seu endereço físico. Os endereços dos bytes são organizados de forma seqüencial, variando do endereço 0 até o máximo disponível no sistema.



Ex. organização da memória em um sistema de 2KB

Quando você está utilizando o computador, o sistema operacional usa uma parte da memória. Quando qualquer programa é executado, o código (linguagem de máquina) e os dados(variáveis) do programa ocupam lugar na memória. Todas as instruções dos programas para ler ou modificar uma variável, utilizam o endereço da variável.

Por exemplo, ao fazer um programa que declara uma variável, o compilador em C reserva uma localização na memória, com um endereço único para acessar esta variável.

```
main() {
     int x;
...
```

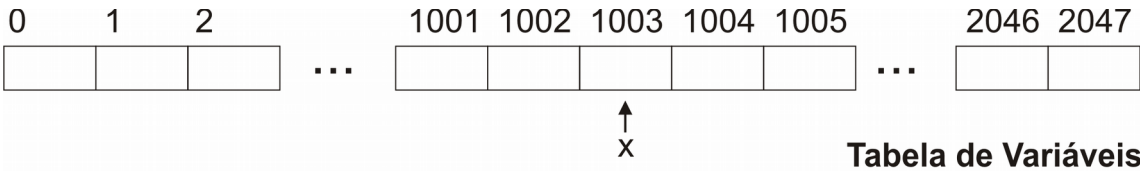


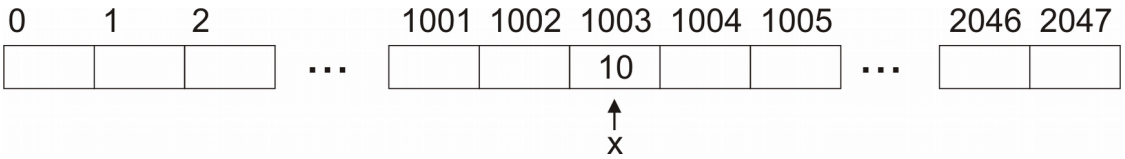
Tabela de Variáveis

Variável	Endereço
x	1003

```
}
```

A partir daí, qualquer operação de leitura ou escrita em uma variável será traduzida pelo compilador para uma instrução que acessa o endereço da variável.

```
main() {
     int x;
     x = 10;
}
```



O que são ponteiros?

• Ponteiro nada mais é do que uma variável que armazena o endereço de uma outra variável
• Proporciona uma maneira de acessar as variáveis, sem referenciá-las diretamente.

Por quê ponteiros são usados?

• Fornecer uma maneira para que as funções possam modificar os valores dos argumentos que ela recebe.
• Para passar matrizes e strings para funções de forma mais conveniente.
• Para manipular matrizes (ou strings), ou partes das matrizes (strings) de forma mais fácil.
• Para criar estruturas de dados complexas, listas, filas, árvores ...
• Para trabalhar com alocação dinâmica da memória. (declaração das variáveis a medida que são necessárias).
• Torna o código mais eficiente.

Criando um Ponteiro

O primeiro passo para a criação de um ponteiro é declarar uma variável que será utilizada para receber o endereço da outra variável. Utilizamos o operador * antes do nome da variável ponteiro.

```
main() {
     int x;
     int *p_x;
     x = 10;
     ...
}
```

Como era de se esperar, a própria variável ponteiro (p_x), precisa ser armazenada em algum lugar da memória. No momento da declaração, p_x aponta para qualquer lugar arbitrário, logo, antes de p_x ser efetivamente utilizado, precisamos inicializá-la.

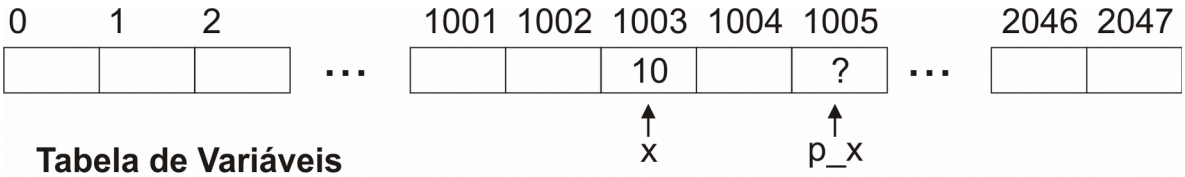


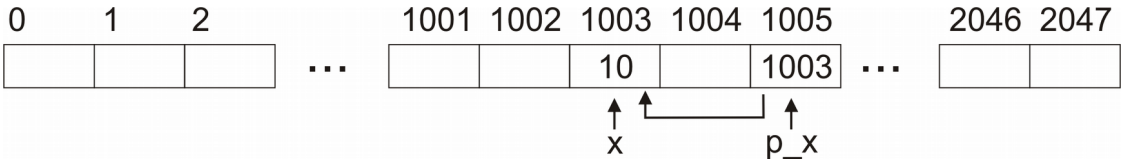
Tabela de Variáveis

Variável	Endereço
x	1003
p_x	1005

Inicializando o Ponteiro

Prosseguindo o exemplo, podemos fazer com que p_x aponte para a variável x (isto é, o valor armazenado na variável p_x será o endereço da variável x). Para realizar tal tarefa, utiliza-se o operador & (endereço de).

```
main() {
     int x;
     int *p_x;
     x = 10;
     p_x = &x;
}
```



Utilizando Ponteiros

O operador * (indireção) é utilizado para acessar o conteúdo da variável apontada por um ponteiro. Logo, o seguinte trecho de código exibirá na tela o valor 10 (conteúdo da variável x).

```
printf("%d", *p_x);
```

Exemplo. Declaração, inicialização e utilização de um ponteiro para inteiro.

```
main() {
     int x;
     int *p_x;
     p_x = &x;
     x = 10;
     printf("O endereço da variável x e: %\n", p_x);
     printf("O endereço da variável x e: %d\n", &x);
     printf("O valor da variável x e: %d\n", *p_x);
     printf("O valor da variável x e: %d\n", x);

     x = 20;
     printf("O valor da variável x e: %d\n", *p_x);
     printf("O valor da variável x e: %d\n", x);

     *p_x = 30;
     printf("O valor da variável x e: %d\n", *p_x);
     printf("O valor da variável x e: %d\n", x);
}
```

Como pode ser visto, agora existem duas maneiras diferentes de acessar o conteúdo da variável x. Quando a variável é referenciada pelo seu nome, diz-se que houve o acesso direto. Quando a variável é referenciada por um ponteiro, dá-se o nome de acesso indireto.

Exercícios.

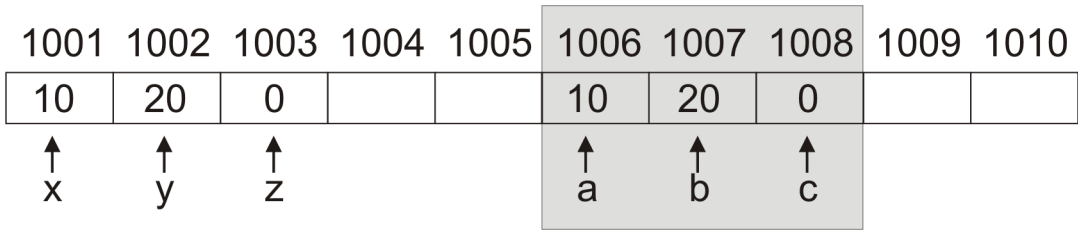
- Implemente o exemplo anterior e observe o resultado.
- Faça um programa na linguagem C que receba o valor de duas variáveis (tipo float) do teclado e troque o valor das variáveis através do acesso indireto.

Ponteiros e Funções

Considere o seguinte programa:

```
void soma(int a, int b, int c) {
     c = a + b;
     printf("A=%d; B=%d; C=%d", a, b, c);
}

main() {
     int x, y, z;

     x = 10; y=20; z = 0;
     soma(x, y, z);
     printf("X=%d; Y=%d; Z=%d", x, y, z);

}
```

A função soma recebeu três valores da função que a chamou (main) e armazena esses valores em um espaço privado da memória. Qualquer modificação nas variáveis a, b e c não irão afetar as variáveis originais x, y e z.



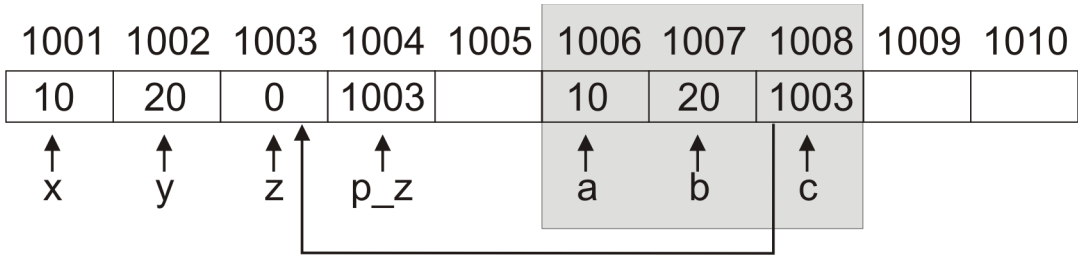
A área em destaque representa a parte da memória privada da função.

No momento em que a função é chamada, é feita uma cópia do valor das variáveis passadas como parâmetros (x, y e z) para as áreas internas da memória(a, b e c, respectivamente). Qualquer modificação das variáveis feitas dentro da função modificará apenas a cópia. Ao encerrar a função a área da memória interna da função é liberada (destruída) e o original permanece intacto.

Entretanto, quando a função precisa modificar o valor da variável REALMENTE, a mesma terá que ser passada através de ponteiros. Veja o exemplo modificado:

```
void soma(int a, int b, int *c) {
     *c = a + b;
     printf("A=%d; B=%d; C=%d", a, b, *c);
}

main() {
     int x, y, z;
     int *p_z = &z;

     x = 10; y=20; z = 0;
     soma(x, y, p_z); /* Opcionalmente soma(x, y, &z); */
     printf("X=%d; Y=%d; Z=%d", x, y, z);

}
```

O terceiro argumento da função é um ponteiro, ou seja, o endereço de uma outra variável.



Neste caso, a variável c apontará para a variável z. Logo, através do operador de indireção (* - na linha em destaque na função soma), o conteúdo da variável original z será modificado.

Exercícios.

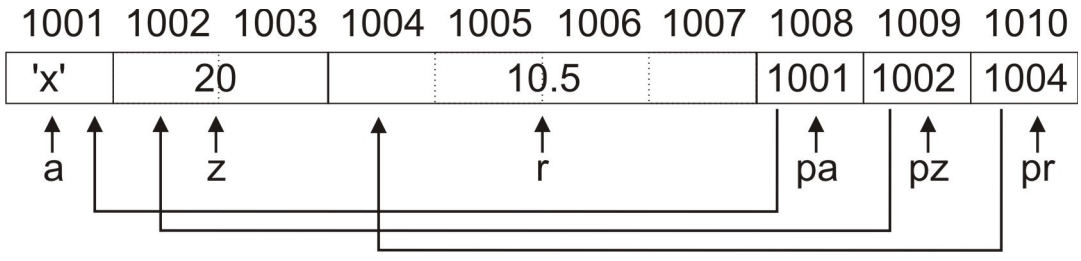
- Implemente os exemplos anteriores e observe o resultado.
- Faça um programa na linguagem C que receba o valor de duas variáveis (tipo char) do teclado, e, através de uma função, troque o valor das duas variáveis.

Ponteiros e Matrizes (Arrays)

Existe um forte relacionamento entre ponteiros e matrizes na linguagem C. Qualquer código escrito na linguagem C, tem suas matrizes convertidas em ponteiros no momento da compilação. Quando utilizamos a notação para identificar os subscritos das matrizes (ex. a[5] = 10;) na verdade estamos usando ponteiros, mesmo sem se dar conta

Devemos ainda lembrar que cada tipo de variável ocupa um determinado tamanho na memória, isto é, uma variável pode ocupar mais de um endereço adjacente na memória. Entretanto, o ponteiro para a variável armazenará apenas o primeiro endereço ocupado pela variável.

```
main() {
     char a, *pa;
     int z, *pz;
     float r, *pr;

     pa=&a;  pz=&z; pr=&r;
}
```



Quando declaramos uma matriz, na verdade estamos reservando um espaço adjacente na memória para armazenar uma determinada quantidade de variáveis de um certo tipo. E o nome da matriz sem os colchetes é um ponteiro para o primeiro elemento da matriz.

```
main() {
     int v[3];
     int *pv;
     pv = v; //Observe que não utilizamos pv = &v ...
}
```

Observe que cada elemento da matriz está ocupando dois bytes na memória (duas posições), pois o tipo inteiro tem o tamanho de 2 bytes.

```
&v[0] é igual a 1001
&v[1] é igual a 1003
&v[2] é igual a 1005
```

Exercício: Execute e verifique o resultado do seguinte trecho de código

```
main() {
     int v[10], i;
     float f[10];
     printf("%d = %d\n", v, &v[0]);
     for(i=0; i < 10; i++)
          printf("Endereco de v[%d] = %d\n", i, &v[i]);
     for(i=0; i < 10; i++)
          printf("Endereco de f[%d] = %d\n", i, &f[i]);
}
```

Acessando os elementos da matriz através da notação de Ponteiros

```
main() {
     int v[10], pv, i;
     pv = v;
     for(i=0; i < 10; i++)
          printf("[%d]", *(pv + i));
}
```

Como interpretar ***(pv + i)** ?
Suponha que o primeiro elemento da matriz está na posição 1000.
Logo, os demais elementos estarão nas posições: 1002, 1004, 1006, 1008 ...

Sabemos que pv é o endereço para o primeiro elemento da matriz, então *(pv + 0) = *(1000 + 0) será conteúdo do primeiro elemento da matriz.
A partir de i=1 teremos o seguinte cálculo:
*(pv + 1) = ***(1000 + 1) = *(1002) ...** ou seja, o segundo elemento da matriz. A operação de atribuição com ponteiros soma a quantidade de bytes necessário para pular a quantidade de números desejada. Seguindo ...
*(pv + 2) = ***(1000 + 2) = *(1004)**

*(pv + 3) = ***(1000 + 3) = *(1006)**
...

Considere agora a seguinte alteração no código

```
main() {
     float v[10], pv;
     int i;
     pv = v;
     for(i=0; i < 10; i++)
          printf("[%d]", *(pv + i));
}
```

Novamente supondo que a matriz inicia no endereço 1000, teríamos:
*(pv + 0) = *1000
*(pv + 1) = ***(1000 + 1) = *(1004)**
*(pv + 2) = ***(1000 + 2) = *(1008)**
*(pv + 3) = ***(1000 + 3) = *(1012)**

Pois, cada elemento do tipo float ocupa 4 bytes.

Alocação Dinâmica de Memória
É uma técnica de programação que consiste em alocar(reservar) a memória somente no momento em que a mesma será utilizada. Geralmente, a alocação dinâmica é feita quando não sabemos, no momento da codificação, o tamanho que as matrizes devem ter.

A função malloc( )
É uma das funções de memória da linguagem C. Ao chamar esta função, devemos passar a quantidade de BYTES que desejamos alocar. A função malloc localizará um bloco na memória com um espaço suficiente e retornará o endereço do primeiro byte deste bloco.

Para utilizar a função malloc devemos incluir a biblioteca stdlib.h

```
#include <stdio.h>
#include <stdlib.h>
main() {
     int *v, i, tamanho;
     printf("digite o tamanho desejado para o vetor");
     scanf("%d", &tamanho);

     v = malloc(tamanho * 2); /* 2 é o tamanho ocupado      |
                              |  por cada variável inteira */
     for(i=0; i < tamanho ; i++)
          scanf("%d", &v[i]);

     for(i=0; i < tamanho ; i++)
          printf("[%d] ", v[i]);
```

```
}
```

A função malloc retorna um ponteiro para o tipo void, pois o tipo void é compatível com qualquer outro tipo de dados, assim, deveremos realizar um cast para informar o tipo de dado que estaremos utilizando. A linha em destaque no código anterior deve ser modificada para:

```
v = (int *) malloc(tamanho * 2);
```

O operador sizeof().
Este operador retorna a quantidade de bytes utilizada para armazenar o tipo da variável passada como parâmetro. Com o sizeof não precisaremos decorar a tabela de bytes por tipos de dados.

```
v = (int *) malloc(tamanho * sizeof(int));
```

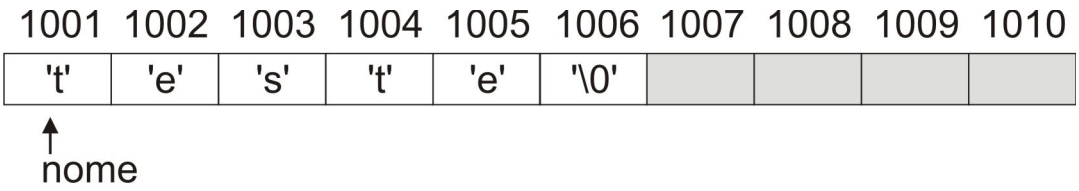
Exemplos:

```
/*Aloca memória para uma matriz de 50 floats*/
float *f;
f = (float *) malloc (50 * sizeof(float));

/*Aloca memória para uma matriz de 10 caracteres*/
char *c;
c = (char *) malloc (10 * sizeof(char));

/*Aloca memória para uma matriz de 15 inteiros*/
int *i;
i = (int *) malloc (15 * sizeof(int));
```

Ponteiros e Strings
Podemos utilizar os conceitos de ponteiros e alocação dinâmica para trabalhar com strings sem desperdício de espaço. O primeiro ponto que deve ser observado é que os strings nada mais são do que matrizes de caracteres unidimensionais, desta forma, tudo o que foi visto sobre matrizes é válido para strings.

Devemos também lembrar que os strings na linguagem C são terminados com o caractere especial ‘\0’.

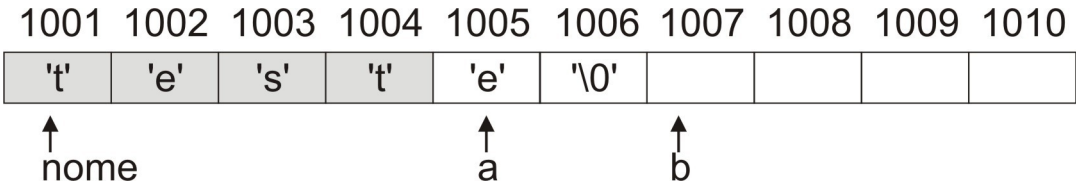
```
main() {
     char nome[10];
     gets(nome); //suponha que foi digitado teste
}
```



Como o string foi declarado com o tamanho 10, os endereços 1007, 1008, 1009 e 1010 estão reservados para a variável nome e não estão sendo utilizados, gerando assim um desperdício de memória.

Por outro lado, se declararmos o String com o tamanho menor, corremos o risco de escrever em uma área de memória utilizada por outra variável, ou mesmo, gerar um erro de violação no acesso à memória.

```
main() {
     char nome[4];
     int a, b;
     gets(nome); //suponha que foi digitado teste
}
```



Este impasse é solucionado através da utilização da alocação dinâmica de memória.

Em primeiro lugar é preciso declarar uma variável temporária do tipo string, com um tamanho razoavelmente grande (chamaremos esta variável de buffer), em seguida, faremos a leitura (gets ou scanf) utilizando esta área temporária, e finalizamos com a criação de um string exatamente do tamanho necessário e copiamos o valor temporário para o novo string. Depois de realizar a cópia para a variável definitiva, o buffer pode ser reutilizado em outras operações de leitura de string, ou pode ter seu espaço de memória liberado.

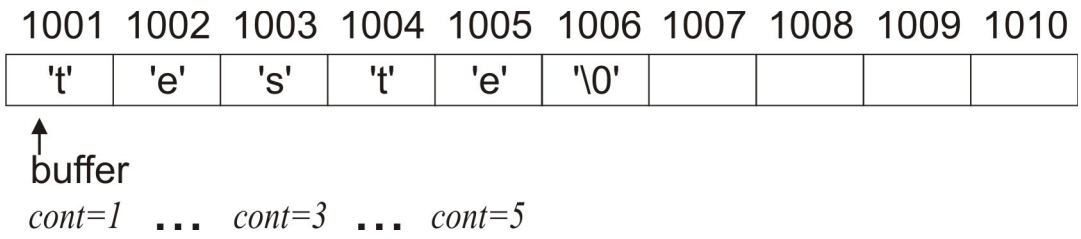
```
main() {
     char *buffer; //variável para leitura temporária
     char *nome; //variável definitiva

     buffer = (char *) malloc (100 * sizeof(char));
     //Considerando o tamanho 100 um valor razoavelmente
     //grande, pode ser qualquer valor, depende do tipo
     //de entrada esperado

     gets(buffer); //A leitura é feita na variável temporária
```

```
     nome = (char *) malloc (tamanho_de_buffer *
sizeof(char));

     copiarString(buffer, nome);
}
```

A única preocupação agora consiste em descobrir o tamanho do string digitado e como copiar um string para outro, visto que o operador de atribuição não é válido em casos de strings (ou matrizes).

Exercício:
1- Fazer uma **função** que calcula o tamanho do String.
2- Fazer uma **função** que copia os caracteres de um string para outro.

Dica para solucionar o exercício: Todo string na linguagem C termina com o caracter '\0', logo, basta criar um contador e percorrer o string, incrementando o contador, até encontrar o '\0';



Para copiar um string para outro podemos utilizar também o caracter '\0' como condição de encerramento do laço.

Felizmente a linguagem C possui uma função para realizar cada uma das tarefas acima, o que veremos na próxima atualização da apostila. ;)

3 – Faça um programa para armazenar gerenciar uma agenda telefônica (nome, telefone) seguindo as seguintes regras.
a) Implementação dos métodos inserir pessoa e listar agenda;
b) Utilização de alocação dinâmica para armazenamento dos nomes;
c) Utilização de estrutura (struct).
d) Telas do programa

Tela Inicial	Ação
AGENDA TELEFONICA Digite a quantidade de contatos que serão armazenados:	O array de contatos será criado dinamicamente com o tamanho informado e o programa ira para a tela Menu.

Tela Menu	Ação
AGENDA TELEFONICA Digite 1 para inserir um contato Digite 2 para listar os contatos Digite 3 para sair	Se o usuário digitar 1 o programa vai para a tela de inserir contato; Se digitar 2 para a tela de listar contatos; Se digitar 3 o programa é encerrado; Se digitar qualquer outra coisa deve aparecer a mensagem “opção inválida” e retorna a tela de Menu.

Tela Inserir Contato	Ação
AGENDA TELEFONICA INSERIR CONTATO Digite o nome: <lê o nome> Digite o telefone: <lê o telefone>	Realiza a leitura dos dados, insere o contato na lista e retorna a tela de Menu. OBS. Caso a quantidade de contatos ultrapasse o tamanho da agenda informado na tela inicial o programa deve exibir a mensagem “lista cheia” e retorna a tela Menu.

Tela Listar	Ação
AGENDA TELEFONICA LISTA DE CONTATOS Atanagilda: 45123567 Fulano: 1234567 Sicrano: 5456213215 Godofredo: 456132187 Digite qualquer tecla para continuar ...	Lista todos os contatos cadastrados em seguida volta para a tela de Menu.