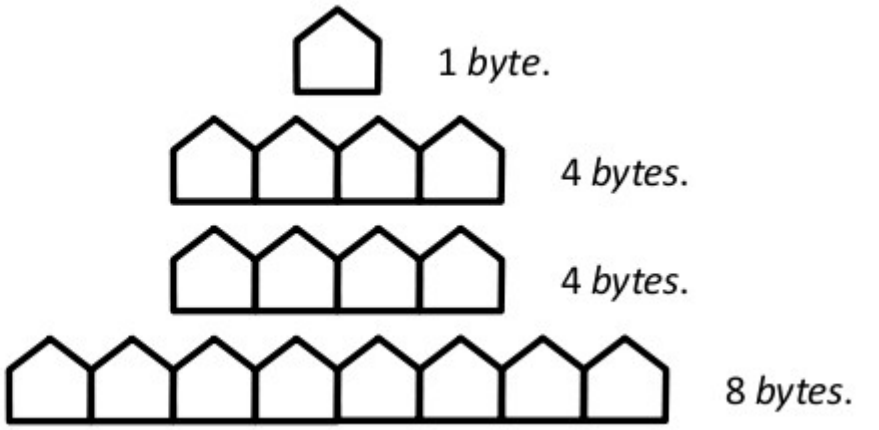
Ponteiros

Prof. Edwar Saliba Júnior
Outubro de 2012

Identificadores e Endereços

- Na linguagem C, uma declaração de variável faz associação entre um identificador e endereços de memória;
- O número de *bytes* de memória depende do tipo utilizado na declaração.

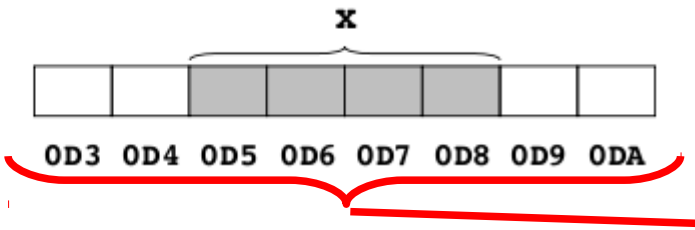
```
char c;
int x;
float x;
double y;
```



Identificadores e Endereços

- Considere a declaração de uma variável do tipo `float`, identificada por:

```
float x;
```



Endereços de memória são representados utilizando-se numeração hexadecimal.

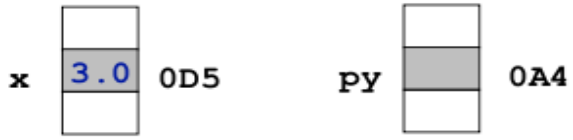
- Após a declaração, os 4 *bytes* que se iniciam no endereço `0D5` estão reservados e são acessados pelo programa através do identificador `x`.

```
printf("Endereço de x: %X", &x);  // irá exibir 05D
```

Ponteiros

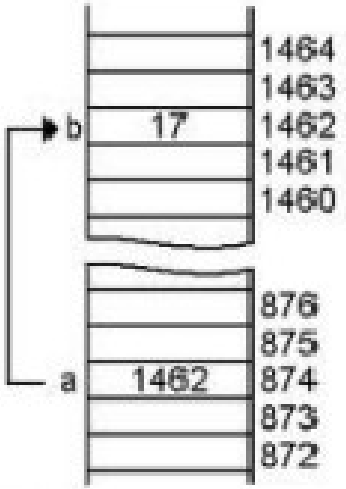
- Outra forma de acesso a memória é através da utilização do endereço dos *bytes*. Para isto é necessário utilizar uma entidade chamada de ponteiro;
- Definição:
  - Ponteiros são variáveis que armazenam endereços de memória;
- Para declarar uma variável do tipo ponteiro, basta utilizar o símbolo * entre o identificador e o tipo desta.

```
float x = 3.0;    // x é uma variável do tipo float
float *py;        // py é ponteiro para variáveis do tipo float
```



Conceito

- Um apontador é uma variável que contém o endereço de uma variável;
- Considere um tipo `t`:
  - `t*` é o tipo “apontador para `t`”;
  - Exemplos:
    - `int *`
    - `float *`
    - `char *`
    - etc.



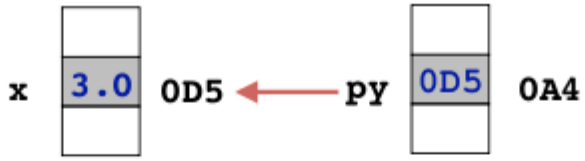
- Uma variável do tipo `t*` contém o endereço de uma variável do tipo `t`;
- Dizemos que o apontador aponta para uma variável daquele tipo.

Armazenando Endereços em Ponteiros

- Para armazenar um endereço em um ponteiro, basta atribuir-lhe o endereço de uma variável de mesmo tipo base:

```
py = &x;          // py aponta para x
```

- Esquematicamente:

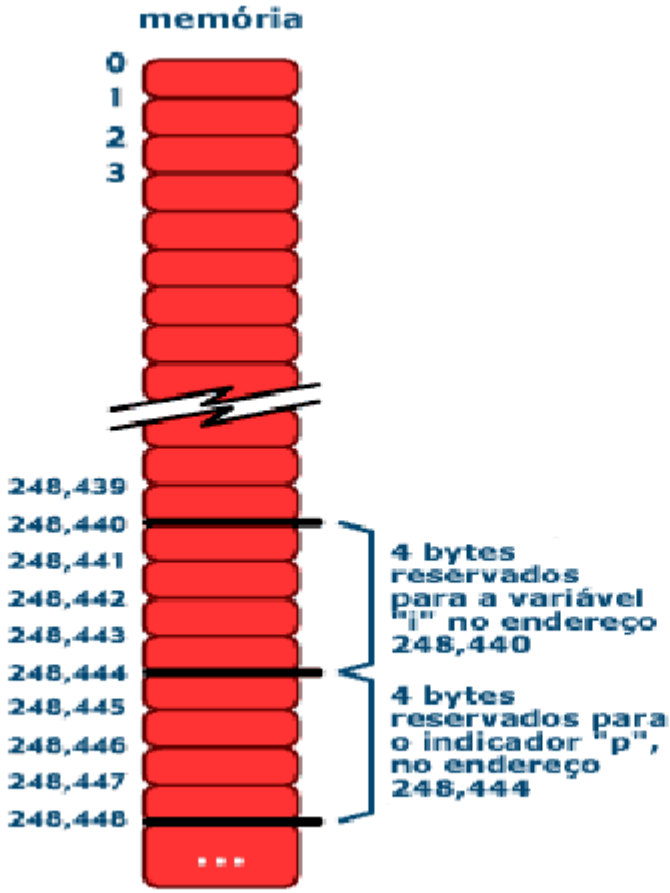


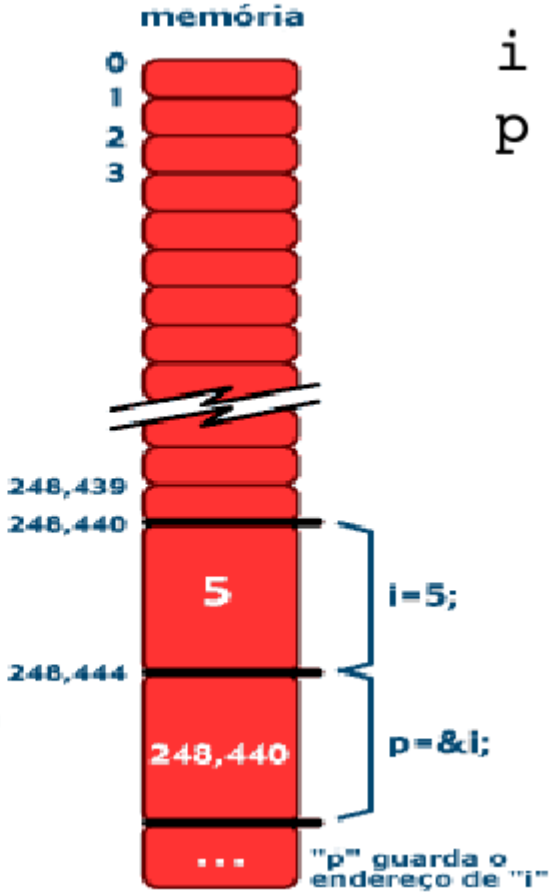

```
char   c, *pc;
int    i, *pi;
double x, *px;

pc = &c;
printf("Tamanho de c: %dB\tEndereco de c: %p\t Próximo endereco: %p\n", sizeof(c), pc, pc+1);
pi = &i;
printf("Tamanho de i: %dB\tEndereco de i: %p\t Próximo endereco: %p\n", sizeof(i), pi, pi+1);
px = &x;
printf("Tamanho de x: %dB\tEndereco de x: %p\t Próximo endereco: %p\n", sizeof(x), px, px+1);
```

Exemplo de Ponteiros

```
int i;
int *p;
```






```
i = 5;
p = &i;
```

Aritmética de Ponteiros

- Seja pa uma variável do tipo ponteiro para inteiros, que está armazenando o seguinte valor: `DA3`. O comando:

```
pa = pa + 1;     // ou pa++;
```

- soma 4 *bytes* (tamanho do tipo) no valor de `pa` que passa a valer: `DA7`
- Ou seja, `pa` passa a apontar para outro endereço de memória.

Aritmética dos Ponteiros

- Apenas os operadores aritméticos + e – podem ser usados com ponteiros;
- Incrementos ou decrementos no valor de um ponteiro fará com que este aponte para endereços imediatamente posteriores ou anteriores ao endereço atual, respectivamente;
- Desta forma, pode-se utilizar ponteiros para percorrer toda a memória de um computador, apenas usando incrementos inteiros de ponteiros.

Operadores

- Existem 3 operações básicas com apontadores. São elas:
  - Declarar ponteiros:
    - Utiliza-se o operador *;
  - Acessar o conteúdo de uma variável apontada:
    - Utiliza-se o operador unário *, também conhecido como operador de indireção ou indeferência;
  - Copiar o endereço de uma variável:
    - Utiliza-se o operador &.

Exemplos de Apontadores

- int x = 1, y = 2, z[10];
- int *px; // Declaração do ponteiro.
- px = &x; // px aponta para x, ou seja, recebe // o endereço de x.
- y = *px; // y recebe o conteúdo do endereço // contido em px. Neste momento y // passa a valer 1.
- *px = 0; // Neste momento x passa a valer 0.
- px = &z[0]; // px aponta para z[0].
- px = z; // px = &z[0];

Opera-ções idên-ticas.

Apontadores

- Exemplos:

```
int x = 1, y = 2, z[10];
int *ip; // Declaração de apontador.
ip = &x; // ip aponta pra x ou ip recebe o
         // endereço de x.
y = *ip; // y recebe o conteúdo da variável
         // apontada por ip, ou seja, y agora
         // passa a valer 1.
*ip = 0; // Altera o conteúdo da variável
         // aponta por ip para 0 (zero), ou
         // seja, x passa a valer 0.
ip = &z[0]; // ip aponta para z[0].
```

Apontadores

- Conceito:
  - Se `ip` aponta para `x`, então `*ip` pode ocorrer em qualquer lugar onde o `x` é permitido:

```
int x = 1;
int *ip = &x;
*ip = *ip + 10;     // x = x + 10;
y = *ip + 1;        // y = x + 1;
*ip += 5;           // x += 5;
++ *ip;             // ++x;
(*ip)++;            // x++;
```

Parênteses necessários devido a ordem de precedência das operações.

Definições em Sequência

- Atenção:

```
int a, b, c;      // 3 variáveis inteiras.
int *pa, pb, pc; // 1 apontador para inteiro
                  // "pa" e duas variáveis
                  // inteiras "pb" e "pc".
```

  - Portanto, para evitar confusão e erro:

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

Ponteiros e Vetores

- Trabalhar com ponteiros em estruturas sequenciais (*strings*, vetores e matrizes) pode melhorar o desempenho de programas.
- O endereço inicial de um vetor corresponde ao nome do mesmo.

```
char s[80], *ps, c;

ps = s; // Equivalente a: ps = &s[0];
```

- Atribuir o elemento da posição 4 da *string* `s` à variável `c`:

```
c = s[4];          // Indexação de vetores ou

c = *(ps + 4);     // aritmética de ponteiros.
```

Vetores e Apontadores

- Existe uma relação intrínseca entre *arrays* e apontadores:

```
int a[10];
```

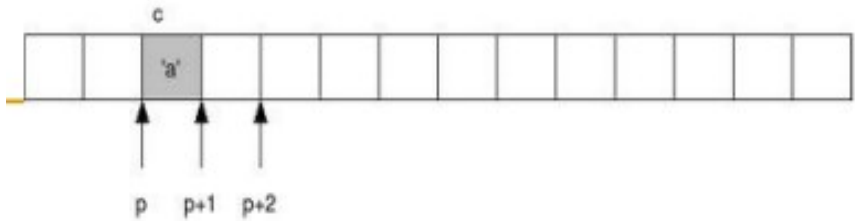
  - define um *array* a de tamanho 10, que é alocado em um bloco contíguo de memória de tamanho suficiente para conter 10 objetos: `a[0]`, `a[1]`, `a[2]`, `a[3]`, …, `a[9]`.
  - O nome do vetor é o endereço do seu primeiro elemento.

```
int *pa;  // Apontador para um inteiro.
pa = a;
pa = &a[0];
```

Comandos equivalentes.
`a == &a[0]`

Vetores: Aritmética de Ponteiros

- Formas de acesso:




```
int a[10];
int *pa;
pa = a;
int x = *pa; // Copia o valor de a[0] em x.
```

`(pa + i)` refere-se ao endereço de `a[i]`

`*(pa + i)` refere-se ao valor armazenado em `a[i]`

`(a + i)` refere-se ao iésimo elemento do *array* `&a[i]`

`*(a + i)` equivale a escrever `a[i]`

`pa++` Aponta para o próximo elemento.

`pa--` Aponta para o elemento anterior.

Vetores e Apontadores

- Considere:

```
int a[10];
int *pa;
pa = a;
```

- Diferença importante entre "`a`" e "`pa`":
  - o nome do vetor não é uma variável e não pode ser alterado;

```
pa = a;  pa++; // Comandos válidos.
a = pa;  a++; // Comandos inválidos.
```

Duas Formas de Percorrer Um *Array*

- Percorrer um *string* usando o índice:

```
void percorreComIndice(char v[]){
  int i;
  for(i = 0; v[i] != 0; i++)
    use(v[i]);
}
```

- Percorrer um *string* usando um apontador:

```
void percorreComApontador(char v[]){
  char *p;
  for(p = v; *p != 0; p++)
    use(*p);
}
```

Alocação Contígua de Memória

- Demonstração

```c
#include <stdio.h>
#include <stdlib.h>
#define LIMITE 100

int main()
{
    const int MAX = 10;
    int a[MAX];
    int i;

    for(i = 0; i < MAX; i++){
        a[i] =  rand() % LIMITE;
    }
    for(i = 0; i < MAX; i++){
        printf("\n[%d] = %d no endereço %x", i, a[i], a+i);
    }

    return 0;
}
```

```
D:\Programming\Win32\C\Aulas\IFTM\Exemplos\AlocacaoContiguaDeMe
[0] = 41 no endereço 28fe90
[1] = 67 no endereço 28fe94
[2] = 34 no endereço 28fe98
[3] = 0 no endereço 28fe9c
[4] = 69 no endereço 28fea0
[5] = 24 no endereço 28fea4
[6] = 78 no endereço 28fea8
[7] = 58 no endereço 28feac
[8] = 62 no endereço 28feb0
[9] = 64 no endereço 28feb4
Process returned 0 (0x0)   execution time : 0.207 s
Press any key to continue.
```

Alocação Dinâmica de Memória

- Um **vetor** de inteiros com tamanho definido pelo usuário em tempo de execução será declarado com um ponteiro de inteiros.

```
int *x, n, i;
printf("Digite o tamanho do vetor: ");
scanf("%d",&n);
// Aloca n posições de tamanho "int".
x = (int *) calloc(n, sizeof(int));
for(i = 0; i < n; i++){
  printf("Digite um valor: ");
  scanf("%d",&x[i]);
}
…
free(x);
```

Importante!
A memória alocada dinamicamente deve ser liberada pela função "free()".

Alocação Dinâmica de Memória

- Para **matrizes** a alocação deverá ser feita em duas etapas:

```
float **m; // Uma matriz de reais.
int r, c;  // Suas dimensões.
int i, j;  // Seus índices.
```

- **Alocar memória para as linhas.**

```
printf("Digite a quantidade de linhas: ");
scanf("%d",&r);
m = (float **) calloc(r, sizeof(float*));
```

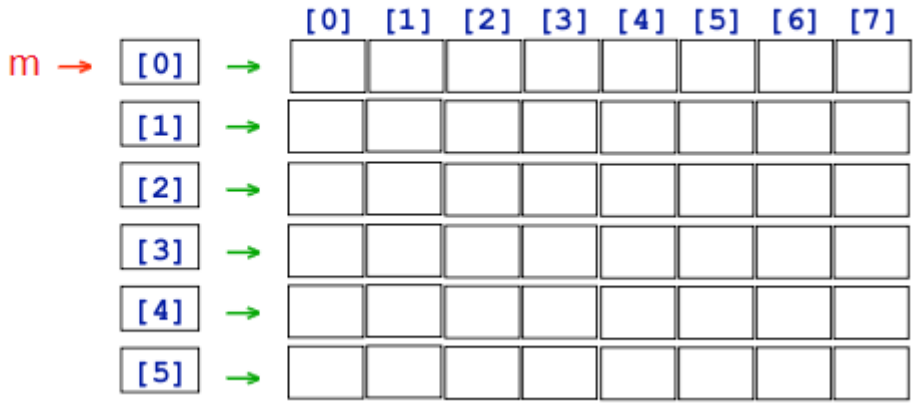
- **Para cada linha alocar memória para as colunas.**

```
printf("Digite a quantidade de colunas: ");
scanf("%d",&c);
for(i = 0; i < r; i++){
   m[i] = (float *) calloc(c, sizeof(float))
}
```

Representação da Matriz

Esquematicamente:

r = 6 e c = 8



- Para liberar a memória:

```
for(i = 0; i < r; i++){
  free(m[i]);
}
free(m);
```

Verificando a Alocação de Memória

- Seja “p” um ponteiro para um tipo qualquer. O seguinte teste:

```
if(p == NULL){
  puts(“Memória insuficiente para a alocação de \”p\”.”);
  exit(0);
}
```

- Poderá ser utilizado para verificar se a alocação de memória para o ponteiro “p” foi bem sucedida. Caso contrário o programa poderá retornar uma mensagem de erro e encerrar a execução.

Bibliografia

- LAUREANO, Marcos. **Programação em C para ambiente Linux**. Disponível em: <http://br-c.org/doku.php>. Acesso em: 06 fev. 2011.
- MURTA, Cristina Duarte. *Slides* da disciplina de Programação de Computadores I. CEFET-MG, 2010.
- SENNE, Edson Luiz França. **Primeiro Curso de Programação em C**. 2. ed. Florianópolis: Visual Books, 2006.
- SOARES, Gabriela Eleutério. *Slides* da disciplina de Programação de Computadores I. CEFET-MG, 2011.