Ponteiros e Alocação Dinâmica de Memória

DCC200 - Algoritmos II

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação

Conteúdo

- Ponteiros
  - Introdução
  - Exemplos
  - Aritmética de ponteiros
  - Vetores e funções
- Alocação Dinâmica de Memória
  - Introdução
  - Alocação dinâmica de memória
  - Operadores `new` e `delete`
  - Alocação dinâmica de vetores e matrizes
  - Funções

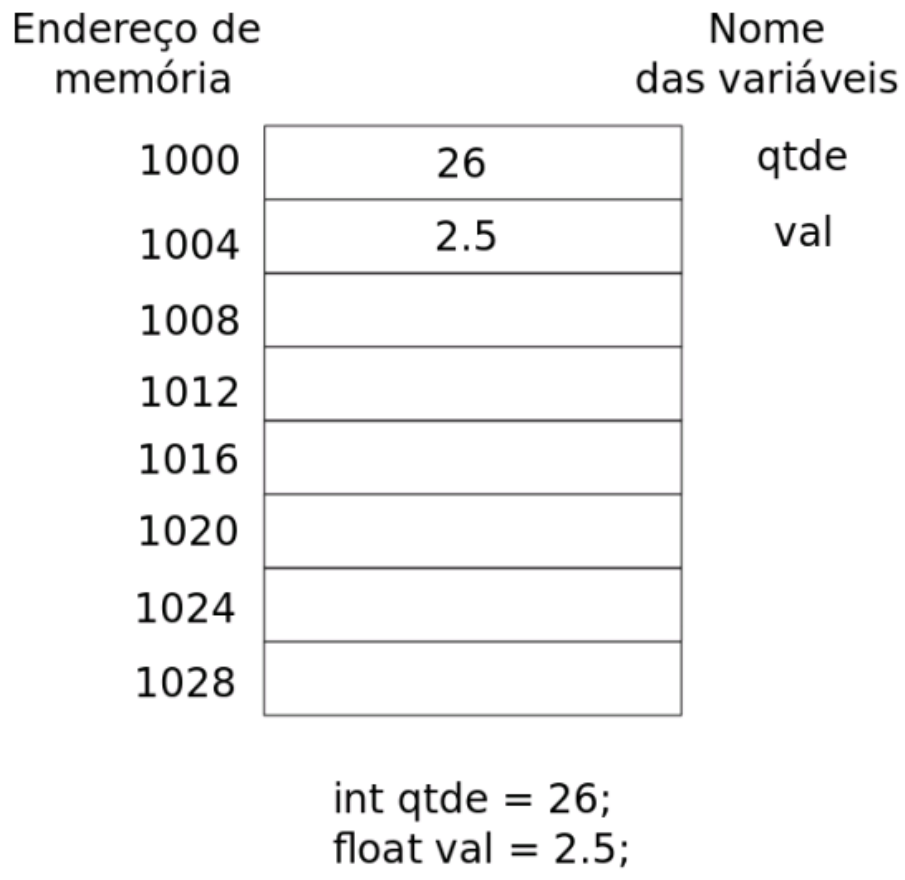
Ponteiros

Introdução

- Cada objeto (variável, string, vetor etc.) que reside na memória do computador ocupa um certo número de bytes:
  - `char`: 1 byte
  - `bool`: 1 byte
  - `short`: 2 bytes
  - `int`: 4 bytes
  - `float`: 4 bytes
  - `double`: 8 bytes

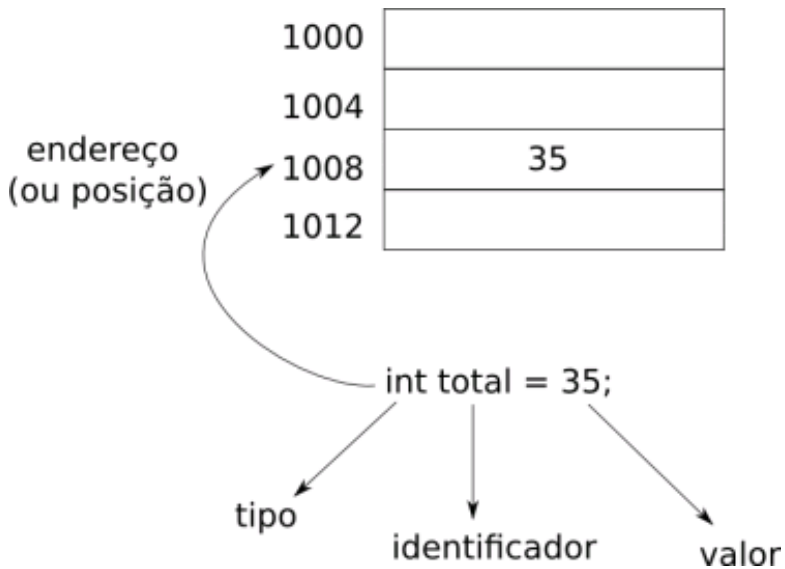
Introdução

- A memória de qualquer computador é uma sequência de bytes.
- Cada byte na memória é identificado por um endereço numérico que independe do seu conteúdo.
- Normalmente os endereços de memória são representados no formato hexadecimal:
  - 0x0065FD40
  - 0x0065FD44



Introdução

- Ou seja, sempre que declaramos uma variável, temos associados a ela:
  - um nome (ou identificador)
  - um endereço de memória
  - um valor
- Sempre que declaramos uma variável, o programa aloca espaço de memória para ela e sabe internamente onde ela está armazenada.



Endereços

- Antes de falar de ponteiros, vamos descobrir o endereço de variáveis.
- Para isso é preciso usar o operador **endereço de**, representado por &. Quando esse operador é aplicado a uma variável obtém-se o seu endereço de memória.
- Se `total` é a variável, então `&total` é o seu endereço.

```
int biscoitos = 6;
double peso = 4.5;
cout << "valor de biscoitos: " << biscoitos;
cout << " endereco de biscoitos: ";
cout << &biscoitos << endl;
cout << "valor de peso: " << peso;
cout << " endereco de peso: " << &peso << endl;
```

- Saída

```
valor de biscoitos: 6 endereco de biscoitos: 0x0065FD40
valor de peso: 4.5 endereco de peso: 0x0065FD44
```

Ponteiros

- Um ponteiro é simplesmente uma variável que armazena o endereço de outra variável ao invés do conteúdo desta.
- Tem-se assim um tipo de dado especial – o ponteiro – o qual armazena o endereço de um valor.
- O nome da variável de um tipo ponteiro representa a posição de memória daquele valor.
- Aplicando o operador *, conhecido como operador **conteúdo de**, recupera-se o valor armazenado naquela posição de memória.
- Isto é, se `pt_x` é um ponteiro, então `*pt_x` é o valor contido naquele endereço de memória.
- Através do operador *, pode-se alterar o valor que está armazenado em uma posição de memória:

```
*pt_x = 100;
```

Ponteiros

- Exemplo

```
int updates = 6;
int *p_updates;
p_updates = &updates;

// imprime valores
cout << "Valor: updates = " << updates;
cout << ", *p_updates = " << *p_updates;
cout << endl;

// imprime enderecos
cout << "Endereco: &updates = " << &updates;
cout << ", p_updates = " << p_updates << endl;

// usa ponteiro para alterar o conteudo
*p_updates = *p_updates + 1;

cout << "Valor atualizado: updates = ";
cout << updates << endl;
```

Ponteiros

- Declarando variáveis do tipo ponteiro

```
int *pt_var;
```

- Ou ainda declarar ponteiros para outros tipos de dados:

```
char *pt_var1;
float *pt_var2;
double *pt_var3;
```

- Pode-se declarar e inicializar um ponteiro

```
int numero = 10;
int *pt_num = &numero;
```

- É preciso tomar cuidado ao usar variáveis do tipo ponteiro!

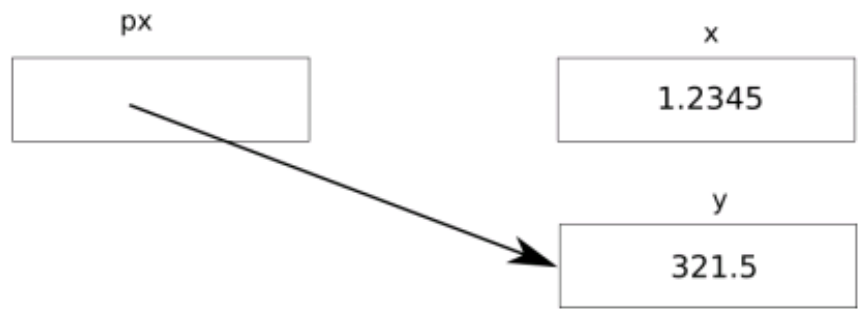
```
// erro
int *pt_var;
*pt_var = 2058; // ponteiro nao inicializado
```

Ponteiros

```
double * px;
double x = 5.1234;
double y;
px = &x;
```


```
// altera o valor do objeto apontado por px
*px = 1.2345;
```


```
// faz px apontar para outro objeto
px = &y;
*px = 321.5;
```



Aritmética de Ponteiros

- Somar 1 à uma variável inteira incrementa o seu valor em uma unidade.
- Somar 1 à uma variável do tipo ponteiro incrementa seu valor dependendo do tipo do ponteiro.

```
int numero = 10;
int *pt_num = &numero;
pt_num = pt_num + 1;
```

  - Nesse exemplo a adição soma 4 ao valor numérico do endereço contido em `pt_num`, pois este é um ponteiro para inteiro e um inteiro ocupa 4 bytes na memória.
  - Por outro lado, somar 1 a um ponteiro para double adiciona 8 ao valor numérico do endereço que o ponteiro armazena.
  - Ou seja, ao somar 1 a um ponteiro, você está indo para o endereço do próximo elemento do tipo de dado do ponteiro.
- De forma geral: `pt_tipo = pt_tipo + tam(tipo)*n;`

Aritmética de Ponteiros

- Exemplo

```
int x;
int *ap;
cout << "Digite x: ";
cin >> x;
ap = &x;
cout << "Endereco de x: " << &x << endl;
cout << "Valor de ap  : " << ap << endl;
cout << "Valor de ap+1: " << ap+1 << endl;
cout << "Valor de ap+2: " << ap+2 << endl;
```

- Saída

```
Endereco de x: 0x7FFFE5B7B53C
Valor de ap  : 0x7FFFE5B7B53C
Valor de ap+1: 0x7FFFE5B7B540
Valor de ap+2: 0x7FFFE5B7B544
```

Ponteiros e Arrays

- Em C e C++, o nome de um vetor (array) é interpretado como o endereço de memória do primeiro elemento do vetor.
- Ou seja, o nome de um vetor é um ponteiro.

```
int vet[3] = {10, 20, 30};
cout << "enderecos" << endl;
cout << vet << endl;
cout << vet+1 << endl;
cout << vet+2 << endl;

cout << "valores" << endl;
cout << vet[0] << endl;
cout << vet[1] << endl
cout << vet[2] << endl;
```

Ponteiros e Arrays

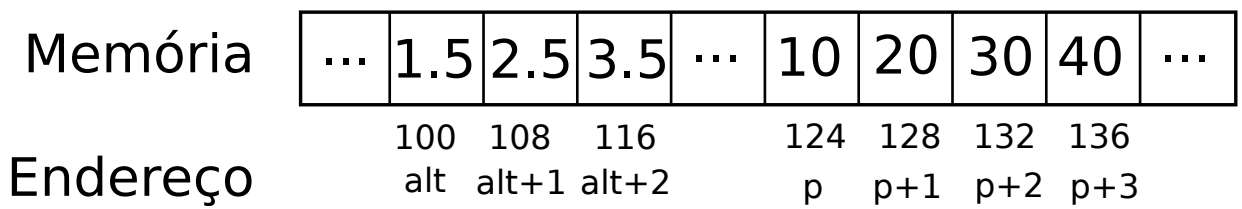
- Exemplo

```
double alt[3] = {1.5, 2.5, 3.5};
int vet[4] = {10, 20, 30, 40};

int *p;

p = vet;
// ou
p = &vet[0];
```

- Representação esquemática da memória



Ponteiros e Arrays

- Ou seja, pode-se acessar os elementos de um vetor de duas formas.
- Usando o operador `[]`
  - `vet[2]`
  - `vet[i]`
  - `vet[i+1]`
- Usando aritmética de ponteiros
  - `*(vet+2)`
  - `*(vet+i)`
  - `*(vet+i+1)`

Ponteiros e o valor NULL

- Assim como variáveis normais, os **ponteiros** não são inicializados quando são instanciados. Portanto, a menos que um valor seja atribuído ao ponteiro, este irá apontar para algum lixo de memória.
- Além de **endereços de memória**, existe um valor adicional que um ponteiro pode armazenar: o valor **NULL**.
- O valor **NULL** é um valor especial que significa que o ponteiro está apontando para nada.

```
int * ptr = NULL;
// ou
int * ptr = 0;

int *ptr2; // ptr2 nao foi inicializado
ptr2 = 0;  // ptr2 agora aponta pra null
```

Ponteiros e Funções

- Em C/C++ existem dois tipos de passagem de parâmetros para funções: passagem por valor e passagem por referência.
- Passagem por valor
  - Uma cópia do valor é passado para a função.
  - Mesmo que a função altere o valor, esta alteração não permanecerá no parâmetro original após o retorno da função.
- Passagem por referência
  - Se a função alterar o valor do objeto passado, essa alteração será realizada no objeto original.

Ponteiros e Funções

- Exemplo errado

```
void troca(int a, int b)
{
  int aux = b;
  b = a;
  a = aux;
}

int main()
{
  int x=2, y=30;
  troca(x, y);
  cout << "x = " << x << " , ";
  cout << "y = " << y << endl;
  return 0;
}
```

- Saída: x=2, y=30
- Saída esperada: x=30, y=2

Ponteiros e Funções

- Para alterar o conteúdo de uma variável passada para uma função como argumento é preciso usar passagem por referência.
- Passamos então o endereço (ponteiro) do objeto que desejamos que a função altere o conteúdo
- Na função manipulamos o conteúdo do objeto usando o operador * (”conteúdo de”) para alterar o seu valor.
- O protótipo correto da função é:

```
void troca(int *a , int *b);
```

Ponteiros e Funções

- Versão correta da função `troca`

```
void troca(int *a, int *b)
{
  int aux = *b;
  *b = *a;
  *a = aux;
}

int main()
{
  int x=2, y=30;
  troca(&x, &y);
  cout << "x = " << x << " , ";
  cout << "y = " << y << endl;
  return 0;
}
```

Ponteiros e Funções

- Como arrays são ponteiros, então a passagem de arrays para funções é **sempre por referência**.
- Os elementos do array não são copiados, apenas o endereço do primeiro elemento do array.
- Sendo assim, pode-se alterar os valores dos elementos do array dentro da função.

```
void incr_vet(int tam, int vet[]) {
  int i;
  for(i=0; i<tam; i++)
    vet[i] = vet[i] + 1;
}

int main() {
  int v[]={10, 20, 5};
  incr_vet(3, v);
  cout << v[0] <<" "<<v[1]<<" "<<v[2]<< "\n";
  return 0;
}
```

Ponteiros e Funções

- Podemos usar os seguintes protótipos para declarar uma função que recebe um array:

```
void incr_vet(int tam, int vet[]);
void incr_vet(int tam, int vet[3]);
void incr_vet(int tam, int *vet);
```

- Todas são equivalentes, e no final das contas o que é passado para a função é o endereço do primeiro elemento do array.
- Por fim, vale lembrar que funções também podem retornar ponteiros.
- Mais detalhes serão apresentados adiante.

Exercícios

1. Ponteiros e aritmética de ponteiros. Faça esse exercício com auxílio do computador e verifique o entendimento das operações. Sejam `i` e `j` são variáveis inteiras e `p` e `q` ponteiros para inteiros. Quais das seguintes expressões de atribuição são **incorretas**?

```
a) p = &i;
b) *q=&j;
c) p=&*&i;
d) i=(*&)j;
e) i=*&*&j;
f) q=&p;
g) i=(*p)++ + *q;
h) if(p==i)i++;
```

Exercícios

2. Faça um **programa** que realize as seguintes operações:
   - Declare uma variável inteira `val`.
   - Declare um ponteiro para inteiro `ptr`.
   - Faça com que `ptr` aponte para `val`.
   - Imprima o valor de `val` e o endereço de memória dela.
   - Imprima o valor de `ptr` e o valor apontado por ele.
   - Modifique o valor de `val` através do ponteiro `ptr`.
   - Crie um novo ponteiro `ptr2` e atribua a ele o mesmo valor de `ptr`.
   - Imprima o valor de `ptr2` e o valor apontado por ele.
   - Modifique o valor apontado por `ptr2`.
   - Imprima o valor de `val`.

Exercícios

3. O que fazem as seguintes funções:

```
void func( ) {
  int mat[ ] = {1, 10, 100};
  for(int j=0; j<3; j++)
    cout << *(mat+j) << endl;
}
```

```
void func( ){
  int mat[ ] = {1, 10, 100};
  for(int j=0; j<3; j++)
    cout << (mat+j) << endl;
}
```

```
void func( ) {
  int mat[ ] = {1, 10, 100}; int *p=mat;
  for(int j=0; j<3; j++)
    cout << (*p)++ << endl;
}
```

Exercícios

4. Implemente uma única função que recebe um vetor de números inteiros (vet) e o seu tamanho (tam) e
   - conte o total de elementos pares
   - conte o total de elementos impares
   - conte o total de elementos negativos
   - e por fim, retorne verdadeiro se existirem números negativos no vetor, ou retorne falso, caso contrário.

   Considere o seguinte protótipo:

```
bool func(int tam, int vet[],
          int *par, int *imp, int *neg);
```

Exercícios

5. Crie uma função que recebe como parâmetros um vetor de números inteiros `vet` e seu tamanho `n`. A função deve trocar o maior valor do vetor com o valor da primeira posição. Se houver mais de um valor maior, considerar a primeira ocorrência. A troca deve ser realizada utilizando aritmética de ponteiros. O valor do maior elemento do vetor deve ser armazenado em `m`.
   Considere o seguinte protótipo:

```
void trocaMaior(int vet[], int n, int *m);
```

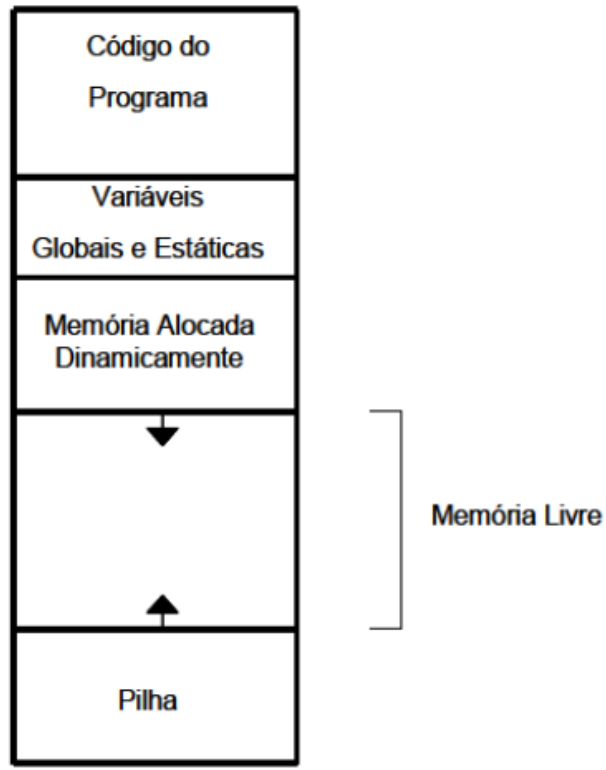
Alocação Dinâmica de Memória

Alocação Dinâmica de Memória

- Quando um programa é executado, a memória do processo é dividida em quatro áreas:
  - **Instruções**: armazena o código C/C++ compilado e montado em linguagem de máquina.
  - **Pilha**: nela são criadas as variáveis locais.
  - **Estática**: onde variáveis globais e estáticas (`static`) são armazenadas.
  - **Heap**: alocada dinamicamente em tempo de execução. Memória "manuseável" através do uso de ponteiros.

Alocação Dinâmica de Memória

- Organização da memória de um processo



- Heap: embora seu tamanho seja desconhecido, o *heap* geralmente contém uma quantidade razoavelmente grande de memória livre.

Alocação Dinâmica de Memória

- As variáveis da **pilha** e da **memória estática** precisam ter **tamanho conhecido** antes do programa ser compilado.

```
int x;
float y;
double vet[10];
```

- A **alocação dinâmica de memória** permite reservar espaços de memória de tamanho arbitrário e acessá-los através de ponteiros.
- Desta forma, podemos escrever programas mais flexíveis, pois nem todos os tamanhos devem ser definidos ao escrever o programa.

```
int N;
cin >> N;

// alocar vet com N elementos
```

Alocação Dinâmica de Memória

- Em C++ a alocação dinâmica de memória é feita usando os operadores `new` e `delete`.
- `new` é usado para alocar memória em tempo de execução.
  - aloca um bloco de bytes consecutivos na memória
  - retorna um ponteiro para o endereço de onde um objeto pode ser armazenado
  - sempre retorna um ponteiro para o tipo que segue o operador new
- `delete` libera o uso de um bloco de memória, permitindo que este espaço seja reaproveitado
  - O operador delete deve ser usado com um endereço de memória que foi originalmente alocado com o operador new.
  - É recomendado que o uso do new seja sempre balanceado com o uso do delete, caso contrário você estará perdendo memória que poderia ser usada.

Alocação Dinâmica de Memória

Operador `new`

- Sintaxe do operador `new`

```
// para alocar uma variavel
new tipoDeDado
// ou para alocar um array
new tipoDeDado[tamanho]
```

- Se existir memória disponível na heap, o operador `new` aloca memória suficiente para uma variável do `TipoDeDado` ou para um array deste tipo, e retorna um ponteiro para (endereço da) área de memória alocada.
- Caso contrário, o programa termina automaticamente com uma mensagem de erro.
- O objeto alocado dinamicamente existe até que o operador `delete` seja usado para destruí-lo.

Alocação Dinâmica de Memória

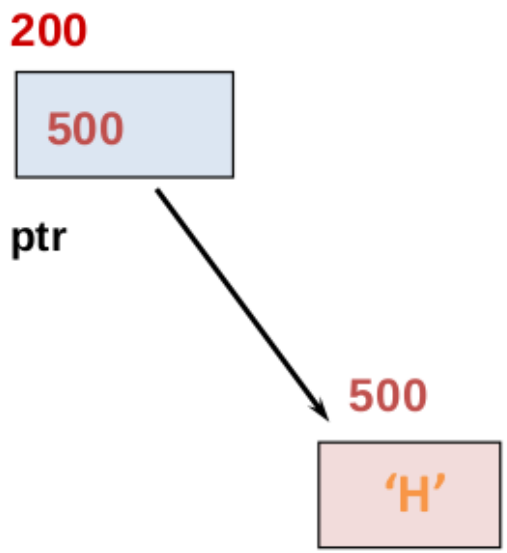
Operador `new`

- Exemplo

```
char *ptr;
ptr = new char;

*ptr = 'H';

cout << *ptr;
```



Alocação Dinâmica de Memória

Operador `new`

```
int main()
{
  int * pt = new int;          // aloca espaco p/ int
  *pt = 202;                   // armazena valor
  cout << "valor do int = " << *pt;
  cout << ": endereco = " << pt << endl;
  // ...

  double * pd = new double; // aloca espaco
  *pd = 10000001.0;            // armazena valor
  cout << "valor do double =" << *pd;
  cout << ": endereco = " << pd << endl;
  cout << "endereco do ponteiro pd: ";
  cout << &pd << endl;
  // ...

  return 0;
}
```

Alocação Dinâmica de Memória

Operador `delete`

- Sintaxe do operador `delete`

```
// para desalocar uma variavel
delete ponteiro;

// ou para desalocar um array
delete [] ponteiro;
```

- O objeto ou array apontado por `ponteiro` é desalocado e o seu valor passa a ser indefinido. A memória é disponibilizada para uso novamente na heap.
- Colchetes [] são usados para desalocar memória de um array alocado dinamicamente.

Alocação Dinâmica de Memória

Operador `delete`

- Atenção!
- É recomendado que o uso do `new` seja sempre balanceado com o uso do `delete`, caso contrário você estará perdendo memória que poderia ser usada.
- Você não deve tentar usar o `delete` para liberar memória que já foi previamente desalocada. O resultado dessa operação é indefinido.
- Você não deve usar o `delete` para liberar memória de uma variável ordinária que não foi alocada dinamicamente.
- Ou seja, não é possível desalocar memória que não foi alocada, assim como não é possível desalocar o mesmo bloco de memória duas vezes.

Alocação Dinâmica de Memória

Operador `delete`

```
int main()
{
  int * pt = new int;          // aloca espaco p/ int
  *pt = 202;                   // armazena valor
  cout << "valor do int = " << *pt;
  cout << ": endereco = " << pt << endl;
  delete pt;                   // libera memoria

  double * pd = new double; // aloca espaco
  *pd = 10000001.0;            // armazena valor
  cout << "valor do double =" << *pd;
  cout << ": endereco = " << pd << endl;
  cout << "endereco do ponteiro pd: ";
  cout << &pd << endl;
  delete pd;                   // libera memoria

  return 0;
}
```

Alocação Dinâmica de Memória

Alocando e desalocando arrays

- Até então os operadores `new` e `delete` foram usados para alocar variáveis ordinárias.
- Para alocar um array de elementos do mesmo tipo, basta usar os comandos:

```
tipo *pt = new tipo[tamanho];

// ...

delete [] pt;
```

- onde
  - `tamanho` é um número inteiro
  - `tipo` é o nome de um tipo de dados (`int`, `float`, ...)
  - `pt` é o ponteiro retornado pelo `new`

Alocação Dinâmica de Memória

Alocando e desalocando arrays

- Exemplo

```
int i;

float *dados;
dados = new float[5]; // aloca

for(i = 0; i < 5; i++)     // processamento
{
  dados[i] = (i+1)*(i+1);
  cout << "valor de dados[i] = " << dados[i];
  cout << endl;
}

delete [] dados;       // desaloca
```

Alocação Dinâmica de Memória

Alocando e desalocando arrays

- Exemplo que determina o tamanho do array em tempo de execução

```
int i, N;                    // determina tamanho
cout << "Digite o tamanho N" << endl;
cin >> N;

float *dados;
dados = new float[N]; // aloca

for(i = 0; i < N; i++)      // processamento
{
  dados[i] = (i+1)*(i+1);
  cout << "valor de dados[i] = " << dados[i];
  cout << endl;
}

delete [] dados;          // desaloca
```

Alocação Dinâmica de Memória

Operadores `new` e `delete`

- Resumo
- Alocou memória com `new tipo;`
- Desaloca com `delete;`
- Alocou memória com `new tipo[tam];`
- Desaloca com `delete [];`

Alocação Dinâmica de Memória

Funções

- Função para somar 2 vetores.
- Errado.

```
float* somaVetores(float u[], float v[])
{
  // alocado de forma estatica
  float r[3];

  r[0] = u[0] + v[0];
  r[1] = u[1] + v[1];
  r[2] = u[2] + v[2];

  return r;
}
```

- Endereço de variável local (no caso `r`) sendo retornada.

Alocação Dinâmica de Memória

Funções

```
float* somaVetores(float u[], float v[])
{
  float *r = new float[3]; // alocacao dinamica
  r[0] = u[0] + v[0];
  r[1] = u[1] + v[1];
  r[2] = u[2] + v[2];
  return r;
}
```

- Programa

```
int i, dim=3;
float vecU[3] = {1.0,1.0,1.0};
float vecV[3] = {2.0,1.0,-1.0};
float *vecRes = somaVetores(vecU, vecV);
for(i = 0; i < dim; i++) {
  cout << "resultado " << i << " = ";
  cout << vecRes[i] << endl;
}
delete [] vecRes;
```

Alocação Dinâmica de Memória

- Estática: tamanhos devem ser conhecidos em tempo de compilação

```
int meuArray[10];
meuArray[3] = 99;
```

- Dinâmica: tamanhos podem ser determinados em tempo de execução

```
int n;
cin >> n;    // assumindo n >= 4
int *ptr;
ptr = new int[n];
ptr[3] = 99;
// ...
delete [] ptr;
```

Exercícios

1. Faça um programa que leia um número inteiro N e que aloque dinamicamente um vetor com N elementos reais e faça a leitura dos seus valores. Em seguida, calcule a média dos valores do vetor e imprima na tela. Por fim, libere a memória alocada de forma dinâmica.

2. Modifique o exercício anterior e crie uma função para realizar a tarefa de calcular a média dos elementos do vetor. Protótipo:

```
float calcMedia(int n, float vet[]);
```

3. Modifique o exercício anterior e crie agora uma função para alocar vetores de números reais de tamanho N de forma dinâmica. Protótipo:

```
float* alocaVetor(int n);
```

Exercícios

4. Dadas as declarações e inicializações abaixo:

```
int a = 1, b = 2, c = 3, *v, *px, *py, *pz;
```

Faça o que se pede nos itens a seguir:

- Faça `px` e `py` apontarem, respectivamente, para `a` e `b`.
- Incremente o valor de `a`.
- Atribua o valor de `c` ao local apontado por `py`.
- Aloque um vetor de 3 elementos inteiros e armazene o endereço resultante em `v`.
- Preencha a última posição do vetor com o valor 10.
- Faça `pz` apontar para a primeira posição do vetor.
- Usando aritmética de ponteiros, preencha as duas primeiras posições do vetor com o valor de `b` e o valor apontado por `px`, nesta ordem.
- Imprima os conteúdos das variáveis `a`, `b` e `c` e o conteúdo do vetor.
- Libere a memória alocada.

Exercícios

5. Crie uma função retorne quantos elementos de um vetor `vet` de inteiros, de tamanho `n`, são maiores do que um valor `val`. Essa função deve imprimir uma mensagem conforme exemplo abaixo para todos os elementos de `vet` que são maiores que `val`. Em seguida, crie uma função para alocar um vetor dinamicamente, copiar os elementos do vetor `vet` que são maiores que `val` para esse novo vetor criado e, ao final, retornar esse vetor criado de forma dinâmica. Se o vetor não possuir nenhum elemento maior que `val`, retornar `NULL`. Protótipos:

```
int func1(int n, int vet[], int val);
int* func2(int n, int vet[], int val, int tam);
```

```
Exemplo de saida da func1:
posicao 0   valor 10   endereco 0x7fff9575c050
posicao 2   valor 33   endereco 0x7fff9575c058
...
```