BCC702 - Programação de Computadores II

Ponteiros
Alocação Dinâmica de Memória

Profª Valéria de Carvalho Santos

UFOP

decom
departamento
de computação

Variáveis

- Usamos nomes/identificadores de variáveis para armazenar valores
- Uma variável corresponde a um “espaço” na memória
- O nome de uma variável no programa refere-se ao conteúdo que esta variável armazena.

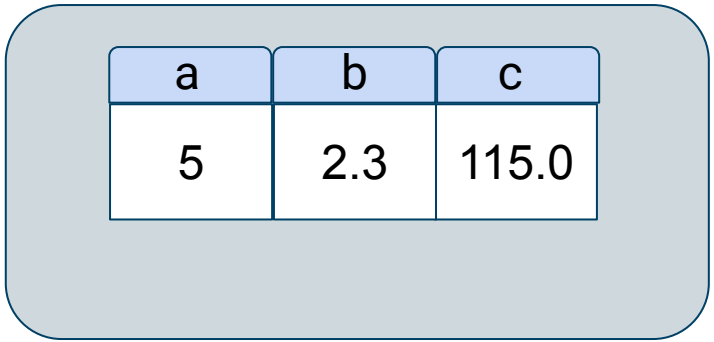
Variáveis

```
int main(){
	int a = 5;
	double b = 2.3;
	double c = a*b;
}
```

Variáveis

```
int main(){
	int a = 5;
	double b = 2.3;
	double c = a*b;
}
```

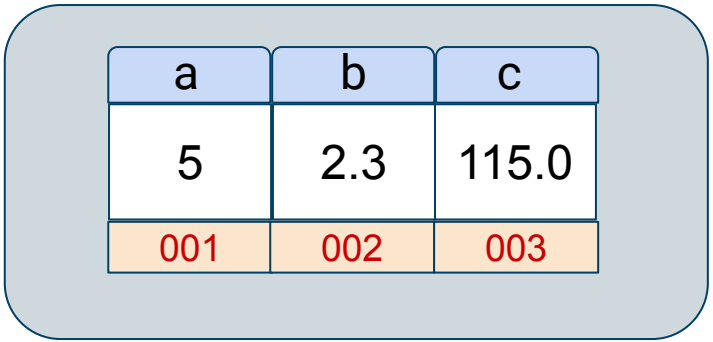
Memória



Variáveis

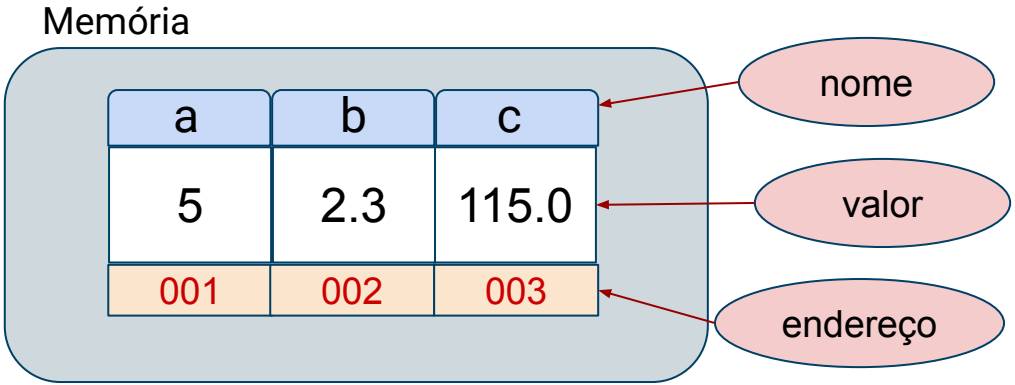
```
int main(){
	int a = 5;
	double b = 2.3;
	double c = a*b;
}
```

Memória



Variáveis

```
int main(){
	int a = 5;
	double b = 2.3;
	double c = a*b;
}
```



Ponteiro

- Armazena o endereço de uma variável
- Indica em que parte da memória a variável está alocada
- Proporciona um modo de acesso à variável sem referenciá-la diretamente

Ponteiro

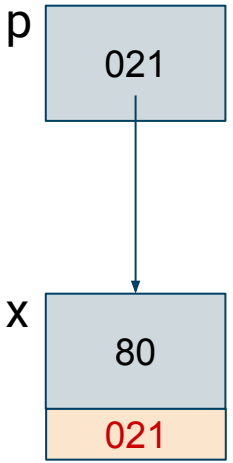
- Seja um ponteiro *p* que armazena o endereço de uma variável *x*, cujo valor é 80. Diz-se que *p* aponta para *x*.

Variável	x	p
Valor	80	021
Endereço	021	028

Ponteiro

- Seja um ponteiro *p* que armazena o endereço de uma variável *x*, cujo valor é 80. Diz-se que *p* aponta para *x*.

Variável	x	p
Valor	80	021
Endereço	021	028



Por que usar ponteiros?

- Modificar parâmetros de funções (passagem por referência)
- Alocar e desalocar memória dinamicamente do sistema
- Criar estruturas de dados complexas
- Passagem de funções como parâmetro (*não veremos aqui*)

Declaração de ponteiros

- Tipo da variável que ele armazena do endereço, seguido do operador * e o nome do ponteiro.

```
tipo  *nome;
```

- Podem ser declarados para qualquer tipo de dados, incluindo registros (*struct*)

Operadores de ponteiros

- Para manipulação de endereços de memória, a linguagem C/C++ oferece dois operadores:
  - O operador **&** (“endereço de”)
  - O operador ***** (“conteúdo de”)

Operadores &

- **O operador & (“endereço de”):** quando aplicado a uma variável resulta no endereço dessa variável na memória do computador.

Operador &

- Exemplo

Variável	a
Valor	5
Endereço	1084

Operador &

- Exemplo

Variável	a
Valor	5
Endereço	1084

```
int a;
a = 5;
cout << “Valor de a: ” << a << endl;
cout << “Endereço de a: ” << &a <<
endl;
```

Operador &

- Exemplo

Variável	a
Valor	5
Endereço	1084

```
int a;
a = 5;
cout << “Valor de a: ” << a << endl;
cout << “Endereço de a: ” << &a <<
endl;
```

```
Valor de a: 5
Endereço de a: 1084
```

Operador *

- **O operador * (“conteúdo de”):** é aplicado a ponteiros, permitindo acessar o conteúdo do endereço de memória armazenado por ele

Operador *

- Exemplo

Variável	a	ptr
Valor		
Endereço	1084	1087

```
int a, *ptr;
a = 5;
ptr = &a;
*ptr = 25;
```

Operador *

- Exemplo

Variável	a	ptr
Valor	5	
Endereço	1084	1087

```
int a, *ptr;
a = 5;
ptr = &a;
*ptr = 25;
```

Operador *

- Exemplo

Variável	a	ptr
Valor	5	1084
Endereço	1084	1087

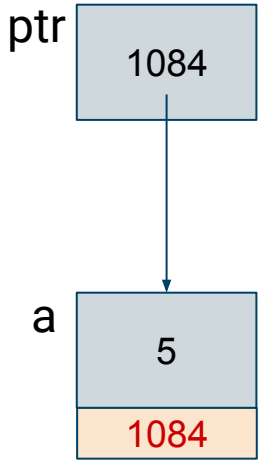
```
int a, *ptr;
a = 5;
ptr = &a;
*ptr = 25;
```

Operador *

- Exemplo

Variável	a	ptr
Valor	5	1084
Endereço	1084	1087

```
int a, *ptr;
a = 5;
ptr = &a;
*ptr = 25;
```



Operador *

- Exemplo

Variável	a	ptr
Valor	25	1084
Endereço	1084	1087

```
int a, *ptr;
a = 5;
ptr = &a;
*ptr = 25;
```

Operadores

- Qual é a saída desse programa?

```
int main(){
        int i = 1;
        cout << i << “, ”;
        int *ptr = &i;
        *ptr = 2;
        cout << i << “, ”;
        i = 1;
        cout << *ptr << endl;
        return 0;
}
```

Operadores

- Qual é a saída desse programa?

```
int main(){
	int i = 1;
	cout << i << “, ”;
	int *ptr = &i;
	*ptr = 2;
	cout << i << “, ”;
	i = 1;
	cout << *ptr << endl;
	return 0;
}
```

1, 2, 1

Ponteiros para registros

- Quando ponteiros apontam para registros, o acesso aos campos desses registros pode ser realizado de duas formas:
  - usando o operador * (“conteúdo de”)
  - usando o operador -> (“seta”)

Ponteiros para registros

```
struct Ponto{
	int x, y;
};
int main(){
	Ponto p1, *ptr;
	p1.x = 2;
	p1.y = 3;
	ptr = &p1;
	cout << p1.x << “, ” << p1.y << endl;
	cout << ptr->x << “, ” << ptr->y << endl;
	cout << (*ptr).x << “, ” << (*ptr).y << endl;
	return 0;
}
```

Ponteiros e parâmetros de funções

- Vimos que há duas formas de passagem de parâmetros para funções:
  - **por valor:** os valores dos argumentos da chamada da função são ***copiados*** para os parâmetros da função
  - **por referência:** os valores dos argumentos da chamada da função são ***vinculados*** aos parâmetros da função

Ponteiros e parâmetros de funções

- Passagem de ponteiros para funções: esse é um caso específico de passagem por valor, em que o valor copiado é o endereço da variável
  - Dessa forma, os valores também podem ser alterados (como na passagem por referência)

Ponteiros e parâmetros de funções

```
void troca(int *a, int *b){
        int temp = *a;
        *a = *b;
        *b = temp;
}
int main(){
        int x = 2, y = 3;
        cout << “x =  ” << x << “y =  ” << y << endl;
        troca(&x, &y);
        cout << “x =  ” << x << “y =  ” << y << endl;
        return 0;
}
```

Ponteiros e vetores

- Existe uma associação forte entre vetores e ponteiros
- Na declaração

```
int v[10];
```

na verdade a variável *v* armazena o endereço inicial do vetor

- Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor.

Ponteiros e vetores

```
void imprime_vetor1(int *v, int n){
	for(int i=0; i<n; i++)
		cout << v[i] << endl;
}

void imprime_vetor2(int v[ ], int n){
	for(int i=0; i<n; i++)
		cout << v[i] << endl;
}
```

```
int main(){
	int v[ ] = {1, 2, 3};
	imprime_vetor1(v, 3);
	imprime_vetor2(v, 3);
	return 0;
}
```

Alocação dinâmica de memória

- Alocação estática de memória
  - O espaço reservado para uma variável existe enquanto o programa estiver sendo executado
  - Para uma variável declarada em uma função, o espaço reservado existe apenas enquanto a função estiver sendo executada

Alocação dinâmica de memória

- Alocação dinâmica de memória
  - O espaço de memória é requisitado em tempo de execução
  - O espaço permanece reservado até que seja explicitamente liberado
  - O espaço alocado e não liberado explicitamente será automaticamente liberado ao final da execução

Alocação dinâmica de memória

- Por que usar alocação dinâmica de memória?
  - Permite melhor controle de uso da memória do computador (quando alocar memória, o quanto alocar, por quanto tempo deixar alocado)
  - Não se restringe ao tamanho da pilha de execução (*stack*) reservada ao programa, uma vez que o espaço é alocado em outra região da memória, denominada *heap*.

Alocação dinâmica de memória

- A alocação dinâmica de memória é feito pelo uso do operador ***new***:
  - recebe o tipo de dado a ser alocado;
  - encontra o bloco de memória com o tamanho necessário
  - retorna o endereço (da primeira posição) do bloco alocado

Alocação dinâmica de memória

- Exemplo:

```
int *ptr;
ptr = new int;
*ptr = 13;
```

Alocação dinâmica de memória

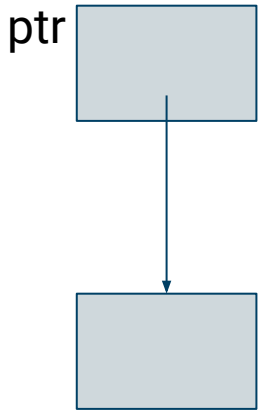
- Exemplo:

```
int *ptr;
ptr = new int;
*ptr = 13;
```

Alocação dinâmica de memória

- Exemplo:

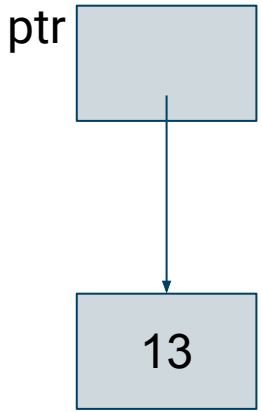
```
int *ptr;
ptr = new int;
*ptr = 13;
```



Alocação dinâmica de memória

- Exemplo:

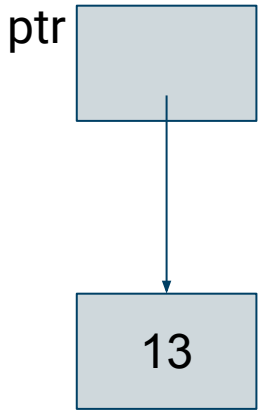
```
int *ptr;
ptr = new int;
*ptr = 13;
```



Alocação dinâmica de memória

- Exemplo:

```
int *ptr;
ptr = new int;
*ptr = 13;
```



- Alocou-se um bloco de memória para armazenar um inteiro e o endereço dessa alocação é armazenado no ponteiro ptr.

Alocação dinâmica de memória

- Quando um espaço é alocado com o operador *new*, este espaço não é desalocado automaticamente
- A desalocação deste espaço de memória é responsabilidade do programador
- Para desalocar espaço de memória é utilizado o comando ***delete***

Alocação dinâmica de memória

- Exemplo:

```
int *ptr;
ptr = new int;
*ptr = 13;
delete ptr;
```

Alocação dinâmica de memória

- Alocação dinâmica de vetor:

```
int *ptr = new int[N];
```

- Os elementos de vetor alocado podem ser acessados normalmente, como os de um vetor comum:

```
ptr[0] = 5;
ptr[1] = -27;
```

- Para desalocar o vetor é necessário usar [ ] após delete:

```
delete[ ] ptr;
```

Alocação dinâmica de memória

- Outro motivo para usar ponteiros e alocação dinâmica de memória é a possibilidade de retornar vetores a partir de funções:

```
int* criar_vetor(int tam){
        int *meu_vetor = new int[tam];
        return meu_vetor;
}
```

```
int main(){
        int *vetor = criar_vetor(2);
        vetor[0] = 2;
        vetor[1] = 6;
        delete[ ] vetor;
        return 0;
}
```

Alocação dinâmica de memória

- Exemplo: alocando e desalocando registro

```
Ponto *ptr = new Ponto;
ptr ->x = 2;
ptr ->x = 3;
delete ptr;
```

Cuidados com o uso de ponteiros

- Ponteiros soltos: um ponteiro solto é aquele que contém o endereço de memória de uma região que já foi liberada

Cuidados com o uso de ponteiros

- Ponteiros soltos: um ponteiro solto é aquele que contém o endereço de memória de uma região que já foi liberada

```
int *ptr = new int;
delete ptr;
*ptr = 2;
```

Cuidados com o uso de ponteiros

- Variáveis dinâmicas perdidas: uma variável que foi alocada dinamicamente, mas não está mais acessível, pois não ponteiros que apontem para ela.

Cuidados com o uso de ponteiros

- Variáveis dinâmicas perdidas: uma variável que foi alocada dinamicamente, mas não está mais acessível, pois não ponteiros que apontem para ela.

```
int *ptr1 = new int;
int *ptr2 = new int;
ptr1 = ptr2;
```

Cuidados com o uso de ponteiros

- Variáveis dinâmicas perdidas: uma variável que foi alocada dinamicamente, mas não está mais acessível, pois não ponteiros que apontem para ela.

```
int *ptr1 = new int;
int *ptr2 = new int;
ptr1 = ptr2;
```

O espaço alocado para ptr1 ficou perdido: não pode mais ser liberado explicitamente, nem pode ser reutilizado no programa.

Exercício

Faça um programa que declare duas variáveis do tipo double e solicite os valores das mesmas ao usuário. Em seguida, guarde o valor da soma em memória alocada dinamicamente. Por fim, exiba o valor da soma.