Introdução à Computação II – AULA 11

BCC Noturno - EMA896115B

Prof. Rafael Oliveira
olivrap@gmail.com

Universidade Estadual Paulista
“Júlio de Mesquita Filho”
UNESP

Rio Claro 2014 (Sem 2)

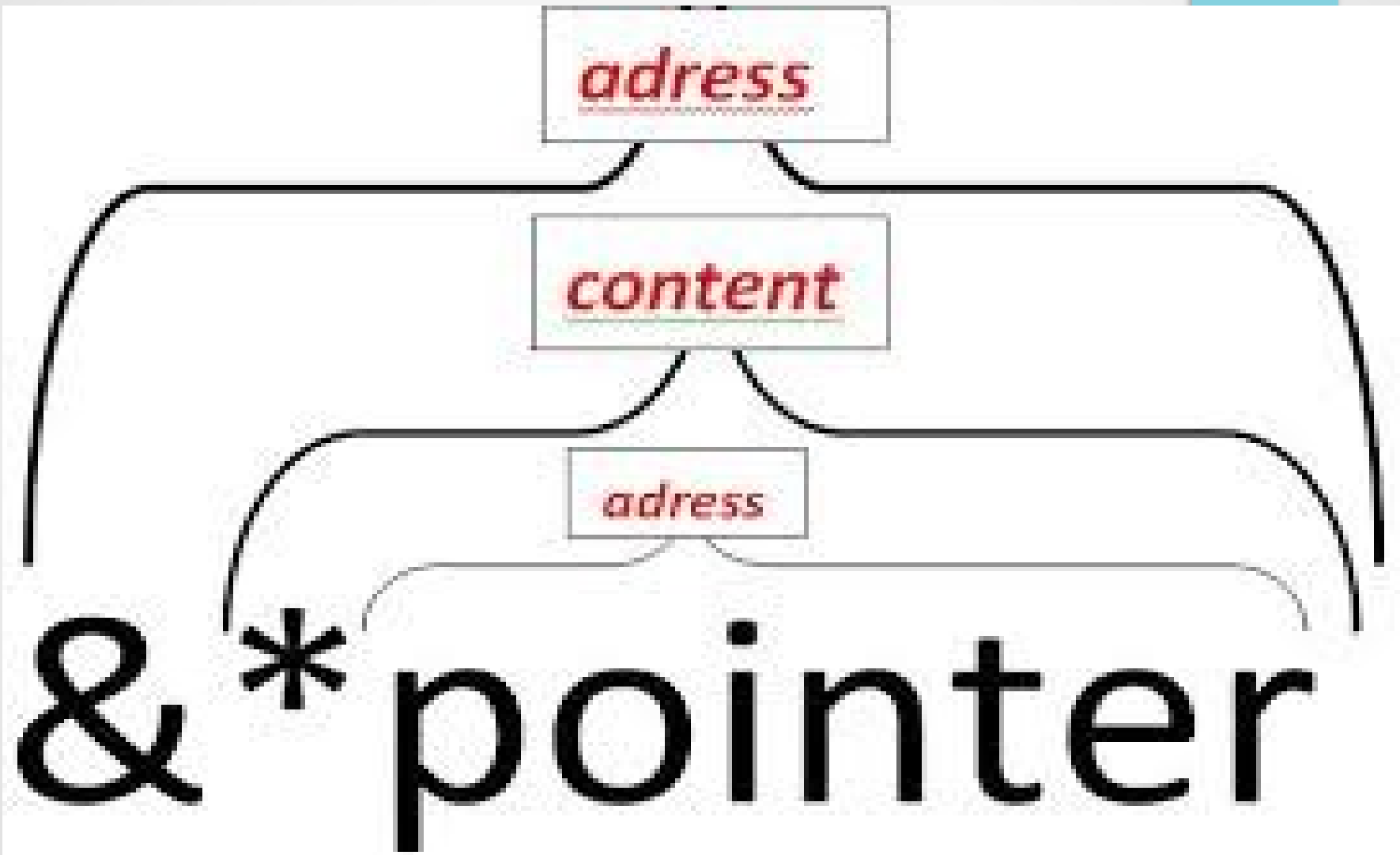
REVISAO

- Metodologia da disciplina:
  - Trabalhos em sala (avaliações contínuas);
  - Listas de exercícios (obrigatórias);
  - Projeto (em grupo) – Definição de grupos até próxima semana. Descrição do trabalho na próxima semana;
  - Provas (2 avaliações) 13/01 e 10/03

REVISAO

- Principais tópicos
  - Breve Introdução à Linguagem C
  - Estruturas de controle
  - Estruturas de repetição
  - Funções
  - Registros
  - Arquivos (texto e binário)

Introdução aos ponteiros



Outline

- Computer Memory Structure
- Addressing Concept
- Introduction to Pointer
- Pointer Manipulation
- Summary

Computer Memory Revisited

- Computers store data in memory slots
- Each slot has an *unique address*
- Variables store their values like this:

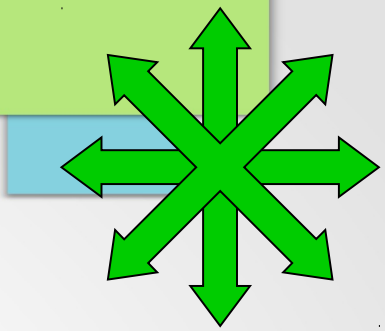
Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 37	1001	j: 46	1002	k: 58	1003	m: 74
1004	a[0]: ‘a’	1005	a[1]: ‘b’	1006	a[2]: ‘c’	1007	a[3]: ‘\0’
1008	ptr: 1001	1009	…	1010		1011	

Computer Memory Revisited

- Altering the value of a variable is indeed changing the content of the memory
  - e.g. i = 40; a[2] = ‘z’;

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 46	1002	k: 58	1003	m: 74
1004	a[0]: ‘a’	1005	a[1]: ‘b’	1006	a[2]: ‘z’	1007	a[3]: ‘\0’
1008	ptr: 1001	1009	…	1010		1011	

Addressing Concept



- Pointer stores the **address** of another entity
- It **refers** to a memory location

```
int i = 5;
int *ptr;                          /* declare a pointer variable */
ptr = &i;                          /* store address-of i to ptr */
printf("*ptr = %d\n", *ptr);       /* refer to referee of ptr */
```

Why do we need Pointer?

- Simply because it's there!
- It is used in some circumstances in C

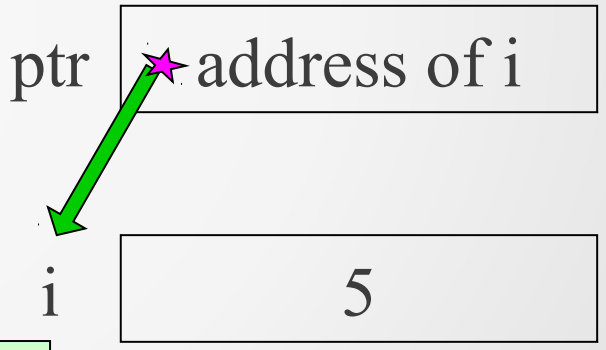
Remember this?

```
scanf("%d", &i);
```

What actually *ptr* is?

- **ptr** is a variable storing **an address**
- ptr is **NOT** storing the actual value of i

```
int i = 5;
int *ptr;
ptr = &i;
printf("i = %d\n", i);
printf("*ptr = %d\n", *ptr);
printf("ptr = %p\n", ptr);
```



Output:

```
i = 5
*ptr = 5
ptr = effff5e0
```

value of ptr = address of i in memory

Twin Operators

- &: Address-of operator
  - Get the *address* of an entity
    - e.g. `ptr = &j;`

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 33	1002	k: 58	1003	m: 74
1004	ptr: 1001	1005		1006		1007	

Twin Operators

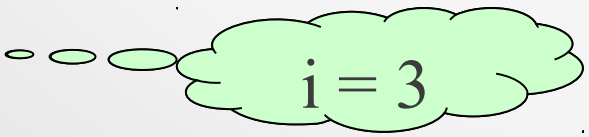
- *: De-reference operator
  - Refer to the *content* of the referee
    - e.g. `*ptr = 99;`

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 99	1002	k: 58	1003	m: 74
1004	ptr: 1001	1005		1006		1007	

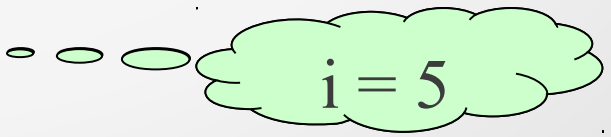
Example: Pass by Reference

- Modify behaviour in argument passing

```
void f(int j)
{
  j = 5;
}
void g()
{
  int i = 3;
  f(i);
}
```




```
void f(int *ptr)
{
  *ptr = 5;
}
void g()
{
  int i = 3;
  f(&i);
}
```



An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10

Ponteiro de Ponteiro

Ponteiro para ponteiro

- **Podemos declarar um ponteiro que guarda o endereço de memória de outro ponteiro**

tipo **nome;

- **Neste caso,**

– *nome é o conteúdo do ponteiro intermediário (um endereço)

– **nome é o conteúdo da região final apontada (um valor)

PORTANTO: Podemos ter ponteiro para ponteiro para ponteiro...

basta aumentar o número de asteriscos.

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10

An Illustration

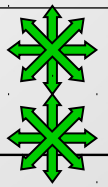
```
int i = 5, j = 10;
int *ptr;     /* declare a pointer-to-integer variable */
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;  /* declare a pointer-to-pointer-to-integer variable */
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

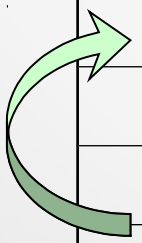
Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	
pptr	int **	integer pointer pointer variable	



Double Indirection

An Illustration

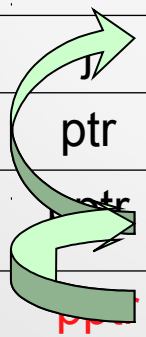
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;     /* store address-of i to ptr */
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	
*ptr	int	de-reference of ptr	5

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr; /* store address-of ptr to pptr */
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	3
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	3

An Illustration

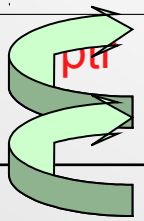
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	7

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

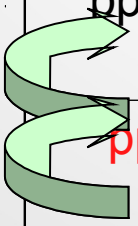
Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	10



An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int	de-reference of de-reference of pptr	9



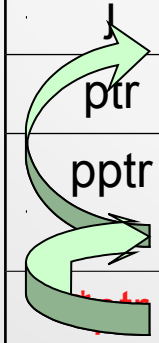
An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	-2
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	-2

Pointer Arithmetic

- What's `ptr + 1`?

➔ The next memory location!

- What's `ptr - 1`?

➔ The previous memory location!

- What's `ptr * 2` and `ptr / 2`?

➔ Invalid operations!!!

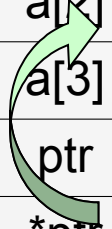
Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```



Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	9.0

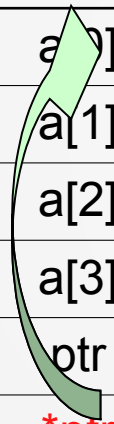
Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```



Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	6.0

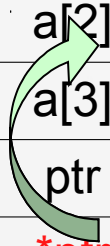
Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```



Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	7.0
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	7.0

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

- Type of `a` is `float *`
- `a[2]` ↔ `*(a + 2)`
  - `ptr = &(a[2])`
  - → `ptr = &(*(a + 2))`
  - → `ptr = a + 2`
- `a` is a memory address *constant*
- `ptr` is a pointer *variable*

More Pointer Arithmetic

- What if `a` is a `double` array?
- A `double` *may* occupy more memory slots!
  - Given **`double *ptr = a;`**
  - What's **`ptr + 1`** then?

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	a[0]: 37.9	1001	…	1002	…	1003	…
1004	a[1]: 1.23	1005	…	1006	…	1007	…
1008	a[2]: 3.14	1009	…	1010	…	1011	…

More Pointer Arithmetic

- Arithmetic operators + and – *auto-adjust* the address offset
- According to the *type* of the pointer:
  - 1000 + sizeof(double) = 1000 + 4 = 1004

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	a[0]: 37.9	1001	…	1002	…	1003	…
1004	a[1]: 1.23	1005	…	1006	…	1007	…
1008	a[2]: 3.14	1009	…	1010	…	1011	…

Advice and Precaution

- Pros
  - Efficiency
  - Convenience
- Cons
  - Error-prone
  - Difficult to debug

- Other pointer declarations that you may find and can make you confused are listed below.

Pointer declaration	Description
`int *x`	`x` is a pointer to `int` data type.
`int *x[10]`	`x` is an `array[10]` of pointer to `int` data type.
`int *(x[10])`	`x` is an `array[10]` of pointer to `int` data type.
`int **x`	`x` is a pointer to a pointer to an `int` data type – double pointers.
`int (*x)[10]`	`x` is a pointer to an `array[10]` of `int` data type.
`int *funct()`	`funct()` is a function returning an integer pointer.
`int (*funct)()`	`funct()` is a pointer to a function returning `int` data type – quite familiar constructs.
`int (*(*funct()) [10])()`	`funct()` is a function returning pointer to an `array[10]` of pointers to functions returning `int`.
`int (*(*x[4])())[5]`	`x` is an `array[4]` of pointers to functions returning pointers to `array[5]` of `int`.

Roteiro

- **Ponteiros**
  - Definição.
  - Operadores.
  - Ponteiros e Variáveis.
  - Ponteiros e Vetores.
  - Ponteiros e Funções.
- **Exercícios**

Ponteiros - Definição

- Ponteiros são tipos de dados que referenciam (ou “apontam” para) endereços de memória.
- Em algumas linguagens com maior abstração, ponteiros não existem expostos ao programador (como em Java) ou tem alternativas mais seguras em outros tipos de dados (como o tipo “referência” em C++).
- Acessar o valor nesse endereço é chamado de “desreferenciar” o ponteiro.
- Em C, um ponteiro é um numero inteiro, referindo-se ao endereço na memória.

Ponteiros - Definição

- Ponteiros são tipos de dados que referenciam (ou “apontam” para) endereços de memória.
- Em algumas linguagens com maior abstração, ponteiros não existem expostos ao programador (como em Java) ou tem alternativas mais seguras em outros tipos de dados (como o tipo “referência” em C++).
- Acessar o valor nesse endereço é chamado de “dereferenciar” o ponteiro.
- Em C, um ponteiro é um numero inteiro, referindo-se ao endereço na memória.

Ponteiros – Operadores

- Para a atribuição de valores para ponteiros, usamos o operador “=“, como fizemos com qualquer outro tipo, MAS:
  - Mesmo sendo um inteiro, não se atribui (normalmente*) valores arbitrários a ponteiros, pois são raras as ocasiões em que é necessário usar um endereço constante para todas as execuções do programa.
  - Para contornar isso, precisamos saber o endereço de variáveis em tempo de execução. Conseguimos isso através do operador “&” (“endereço de”):

```
int inteiro;
int* pInt = &inteiro;
```

  - Como vet ão sem precisar achar o endereço com o “&”:

```
int vetorInt[5];
int* pVetor = vetorInt;
```

*: Exceção para o etornado por funções em ca ugar nenhum.

Ponteiros e Variáveis

- Podemos, usando os operadores apresentados, fazer um ponteiro apontar para um endereço o qual armazena uma variável.

- Com isso, podemos usar o ponteiro para modificar o valor de uma variável:

```
int inteiro = 0;   //inteiro = 0
int* pInt = &inteiro; //pInt aponta para
                      // o endereço de inteiro
*pInt = 42;           //inteiro = 42
```

Ponteiros e Vetores

- Como já foi dito, ponteiros guardam endereços. Vetores também. Podemos acessar os valores do vetor usando ponteiros? Sim:

```
char string[20];
char* pChar = string;
```

- Para acessar cada elemento:

```
for(i = 0; i < 20; i++)
{    //Notação de vetor
     aux = pChar[i];
     //Notação de ponteiro
     *(pChar + i) = funcao();
}
```

Ponteiros e Funções

- Ponteiros, por serem endereços, permitem que acessemos e modifiquemos dados externos à função, de dentro da função, contornando a passagem de variáveis por cópia*:

```
void copiar(int* a, int b){
    *a = b;
}
int main(){
    int foo = 2;
    int bar = 5;
    //o endereço de foo é passado
    //como parametro da função
    copiar(&foo,bar);
    //foo == 5
    return 0;
}
```

*: A