INTRODUÇÃO À PROGRAMAÇÃO

PIAGET

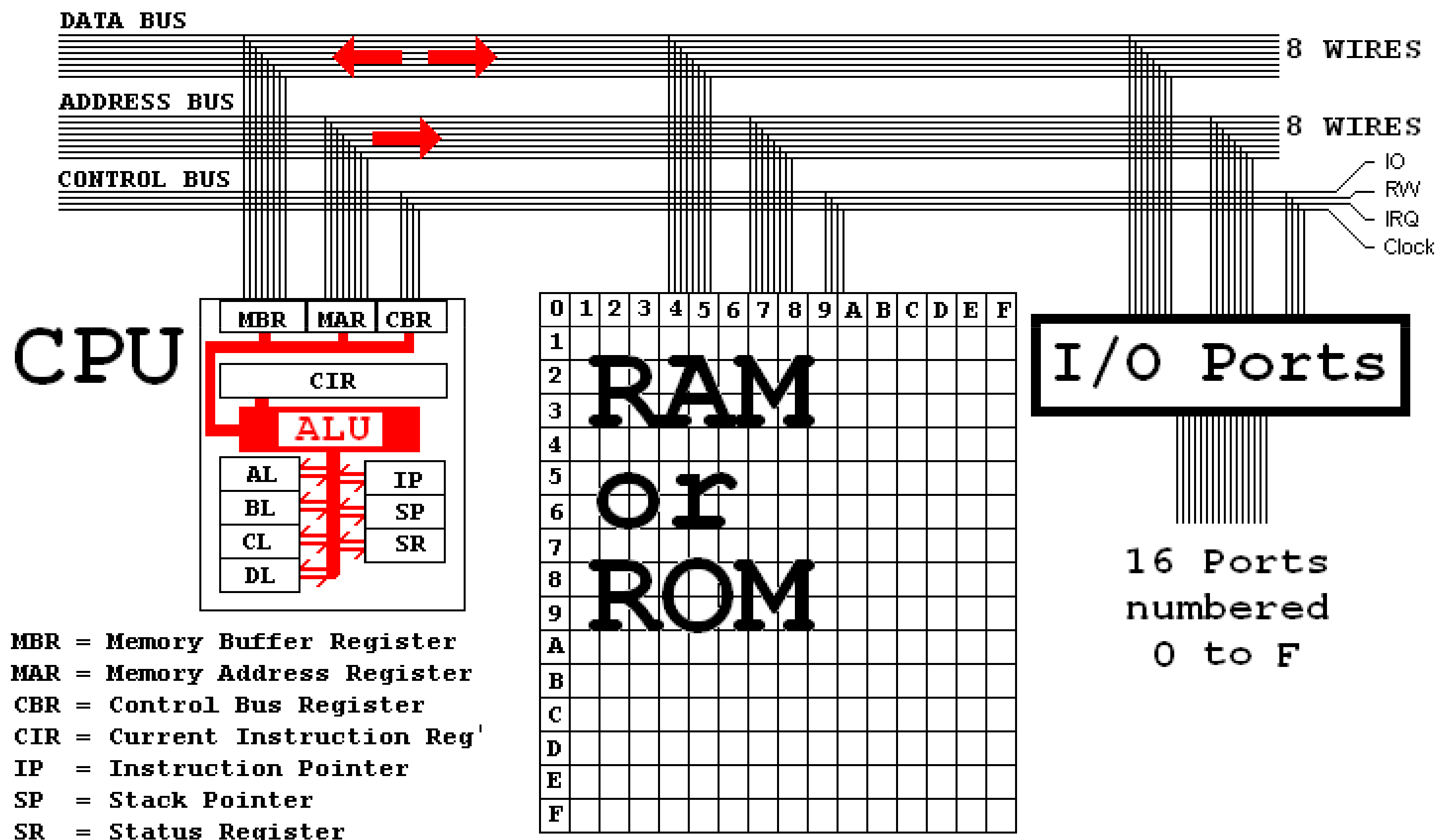
AULA 1

- O que é programar?
- A necessidade de modelar
- A necessidade de abstrair
- Evolução das Linguagens de programação.

Aula 2

- Arquitectura de computadores
- CPU
- A máquina de Van Neumann
- Noção de Registo e Memória
- Endereço e Dados
- Set de Instruções
- Funcionamento do CPU
  - Fase Fetch
  - Fase Execute
- Linguagem Assembly

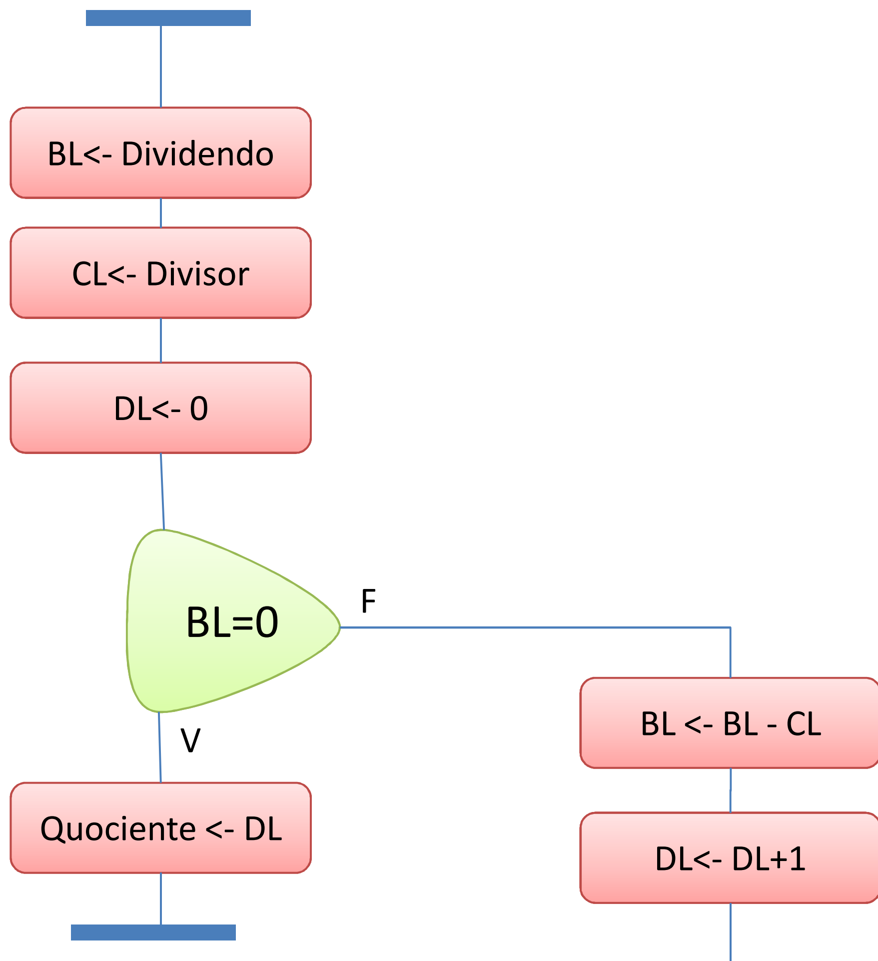
Aula 2 – Exemplo Arquitectura



Aula 3

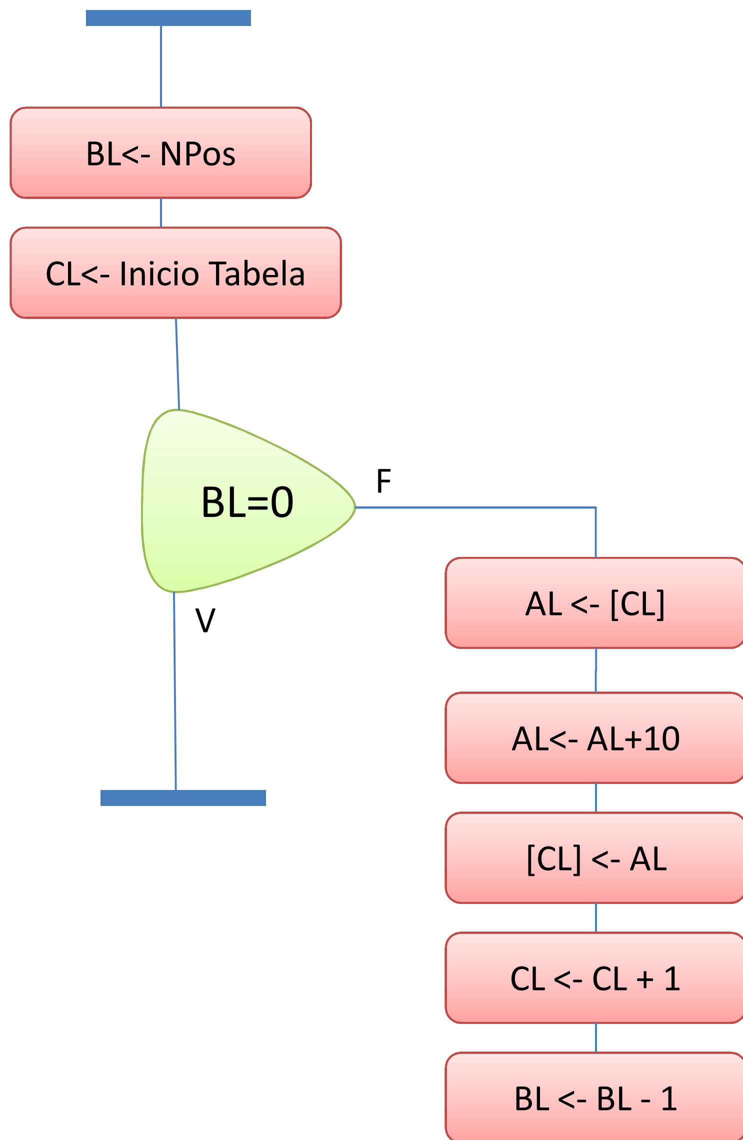
- Utilização do Set de instruções de um CPU com 8 bits de endereço e 8 de dados
- Descrição do Set de Instruções
- Primeiras representações gráficas de programas simples
- Resolução de Programas
  - Divisão Inteira
  - Iteração sobre tabelas em Memória
  - Semáforo

Aula 3 – Divisão Inteira




```
; ----- EXAMPLE DIVISÃO INTEIRA --------------------------

        JMP     Start       ; Skip past the data table.

        DB      20          ; Dividendo.
        DB      8           ; Divisor
        DB      0           ; Resultado

Start:
        MOV     BL,[02]     ; Dividendo em BL
        MOV     CL,[03]     ; Divisor em CL
        MOV     DL,0        ; Quociente = 0
        OR      BL,BL
Loop:   JZ      Final
        INC     DL
        SUB     BL,CL
        JMP     Loop
Final:  MOV     [04],DL

        END
```

Aula 3 - Iteracção




```
; TABELA

        JMP     Start       ;

        DB      3           ; Numero Pos
        DB      20          ; Inicio Tabela.
        DB      30          ;
        DB      40          ;
        DB      50
        DB      60
        DB      70
        DB      80

Start:
        MOV     BL,[02]     ; NPOsicoes
        MOV     CL,03       ; Endereco em CL
        OR      BL,BL
Loop:   JZ      Final
        MOV     AL,[CL]
        ADD     AL, 10
        MOV     [CL], AL
        ADD     CL,02
        DEC     BL
        JMP     Loop

Final:  END
```

Aula 4

- Exercício Heater
- Exercício Semáforo
- Exercício Display 7 Segmentos

Conceitos Fundamentais na Programação

- Variáveis
- Blocos
  - Estrutura de decisão (IF)
  - Estruturas de Repetição
    - While
    - Repeat

Why Programming Languages?

- Increased capacity to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
  - www.tiobe.com/tiobe_index/index.htm
- Better understanding of the significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Criteria for Languages

Characteristics	Readability	Writability	Reliability
Simplicity	X	X	X
Orthogonality	X	X	X
Data Types	X	X	X
Syntax Design	X	X	X
Support for Abstraction		X	X
Expressivity		X	X
Type Checking			
Exception Handling			X
Restricted Aliasing			X

Readability Criteria

- How languages can be read and understood
- In the beginning the focus was efficiency
- Nowadays the focus is more on maintenance and therefore readability become much more important
- The probelm of the adequateness of the language constructs to the problem domain
- Overall Simplicity
  - Feature Multiciplity
    - *Count = Count + 1; Count += 1; Count++; ++Count*
  - Operator Overloading
- Orthogonality
  - Means that a relatively small set of primitive constructs can be conbined in arelatively small number of ways to builfd the control and structure of the language.
  - **IBM** – *A Reg1, memory_cell*; *AR Reg1, Reg2*
  - VAX – *ADDL oper1, oper2* ; can a register or a memory cell
  - C has two kinds of structured data types: structs and vectors. But functions can only return structs.
- Data Types
  - The presence of adequate facilities for defining data types and data structures in a language is a significant aid to readbility
  - Existence of Boolean type; timeOut = 1; vs timeOut = True
- Syntax Design
  - Identifier forms
    - Very short lengths detracts readability -> FORTRAN 77 Initial BASIC
  - Special Words (*while*, *class*, *for*)
    - Use of *end if* and *end loop* facilitates readability
  - Form and Meaning
    - Designing statements so that their appearence at least partially inidcates their purpose is an obvious aid to readability
    - Semantics or meaning should follow directly from syntax
    - *static* in C. Different meaning dependening on the context.If used inside a function it means the variable is created at compile time. If used outside all functions it means the variable is visible only in the file in which its definitions appears.

Writability Criteria

- Is a measure of how easily a language can be used to create programs for a chosen problem domain
- Direct relation between readability and writability. You have to read the program while programming
- Simplicity and Orthogonality
  - Smaller number os primitives and a consistent set of rules for combining them (orthogonality) is much better than many primitives
  - A programmer can design a solution to a complex problem after learning a simple set of primitive constructs
  - To much orthogonality can be error prone
- Support for Abstraction
  - Ability to define and then use complicated structures or operations in ways that allow many of the detais to be ignored
  - Two categories of abstraction: process and data
  - Process abstraction is th e use of subprograms to implement some algorithm (sort for instance) that is required several times in a program
  - More important than making the program longer is the advantage of having highlighted the algorithm (sort)
  - Data Abstraction: example of implementing a tree with Fortran. 3 arrays
- Expressitivity

Reliability Criteria

- A program is said to be reliable if it performs to its specifications under all conditions
- Type Checking
  - Testing for type errors in a given program
- Exception Handling
  - The ability of a program to intercept run-time errors, take corrective measures, and then continue is an obvious aid to reliability. ADA, C++ e Java
- Aliasing
  - Is having two or more distinct names to acess the same memory cell.
- Readability and Writability

Pseudo Linguagem

- Fazer um programa que a partir de uma variável do tipo inteiro indique o digito mais significativo
- A partir de uma variável indique o número de digitos
- Fazer um programa que a partir do valor colocado numa variável do tipo inteiro indique quais os digitos que fazem parte desse número
- Alterar o programa anterior por forma a garantir que os digitos são mostrados da esquerda para a direita. Utilizar vectores.

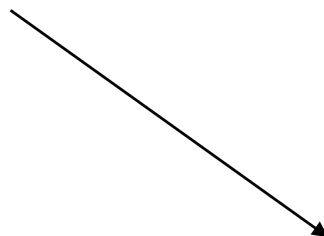
C – Ciclo de Desenvolvimento de Aplicações

- 4 fases

  1. Edição do código fonte
  2. Compilação do programa
  3. “*Linkagem*” dos objectos
  4. Execução do programa

C – Ciclo de Desenvolvimento de Aplicações

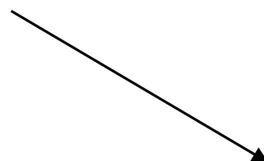
- **Edição do código fonte**
- Compilação do programa
- “*Linkagem*” dos objectos
- Execução do programa



O programador escreve o código em ficheiros com a extensão **.c**

C – Ciclo de Desenvolvimento de Aplicações

- Edição do código fonte
- **Compilação do programa**
- “*Linkagem*” dos objectos
- Execução do programa



O compilador verifica se há erros de sintaxe.

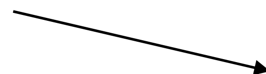
Pode detectar erros ou *warnings*

$ gcc –c prog.c

Cria um ficheiro **.o**

C – Ciclo de Desenvolvimento de Aplicações

- Edição do código fonte
- Compilação do programa
- **“*Linkagem*” dos objectos**
- Execução do programa



O ficheiro objecto é junto com outros e produz-se a ficheiro executável (**a.out**)

Se algum símbolo não for encontrado dá erro.

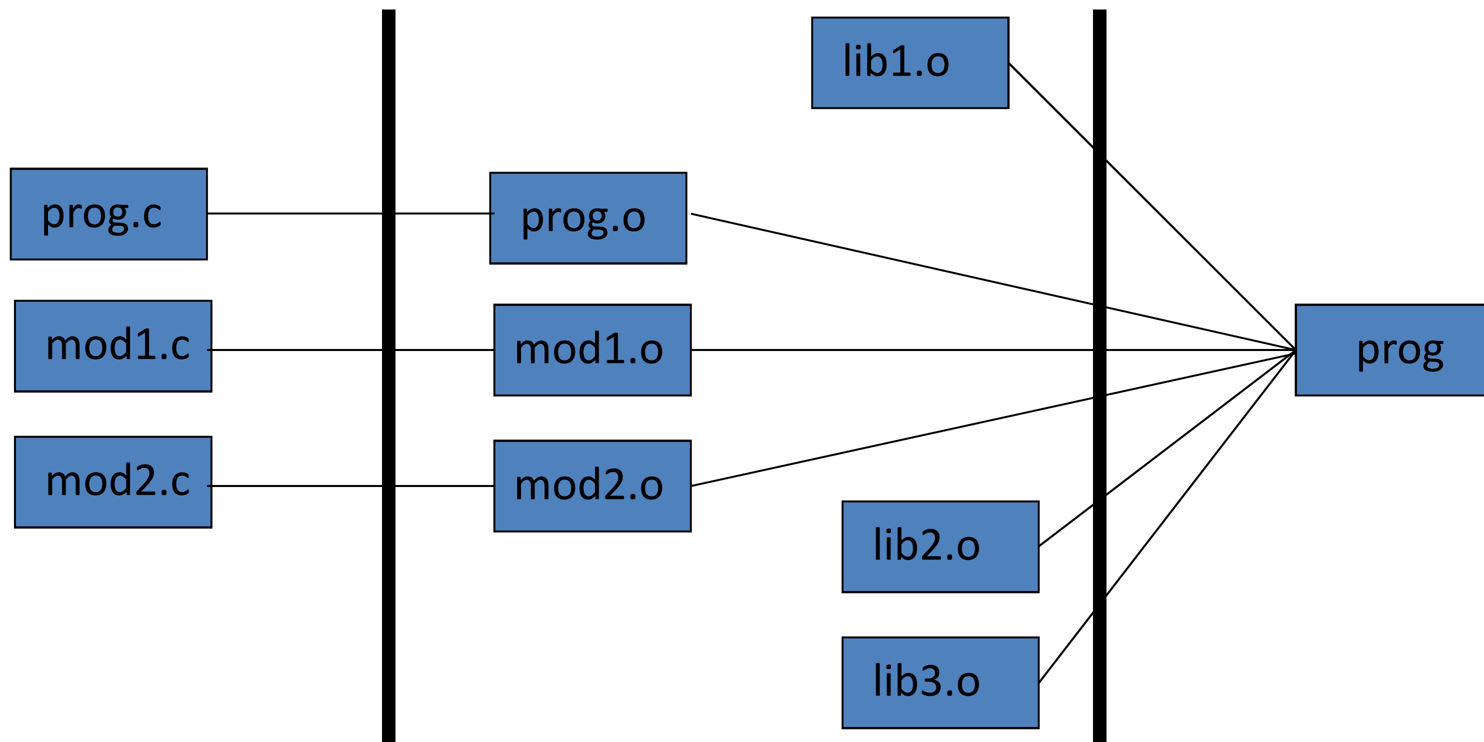
```
$ gcc prog.c –o prog
```

C – Ciclo de Desenvolvimento de Aplicações

- Edição do código fonte
- Compilação do programa
- “*Linkagem*” dos objectos
- **Execução do programa**

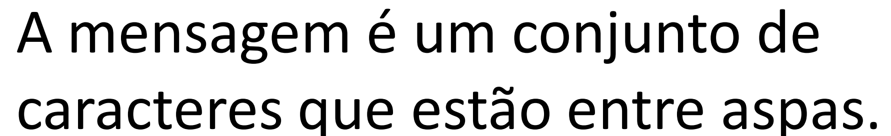
Colocar o programa a “correr”

C – Ciclo de Desenvolvimento de Aplicações



Programa – *Hello World*

```
# include <stdio.h>
main()
{
    printf ("Hello World");
}
```



Como escrever no écran

- É importante perceber que a escrita tem de ser feita também de acordo com o tipo de variável.
- C
  - printf
- C++
  - cout << “Ola Povo”<<“ de Portugal”<<endl
- JAVA
  - print(“ola povo”)
  - println(“O numero que vou” + “ escrever e “ + 2)
- PASCAL
  - write(“ola povo”)
  - writeln(“ola mundo”)

O símbolo \

O símbolo \ permite definir caracteres especiais ou tirar o efeito a algum especial (por exemplo \” são as aspas).

\n – *New Line*

\r – *carriage return*

\t – tabulação

\’ – plica

\” – aspas

\\ - caracter \

\? – ponto de interrogação

Formatadores do printf

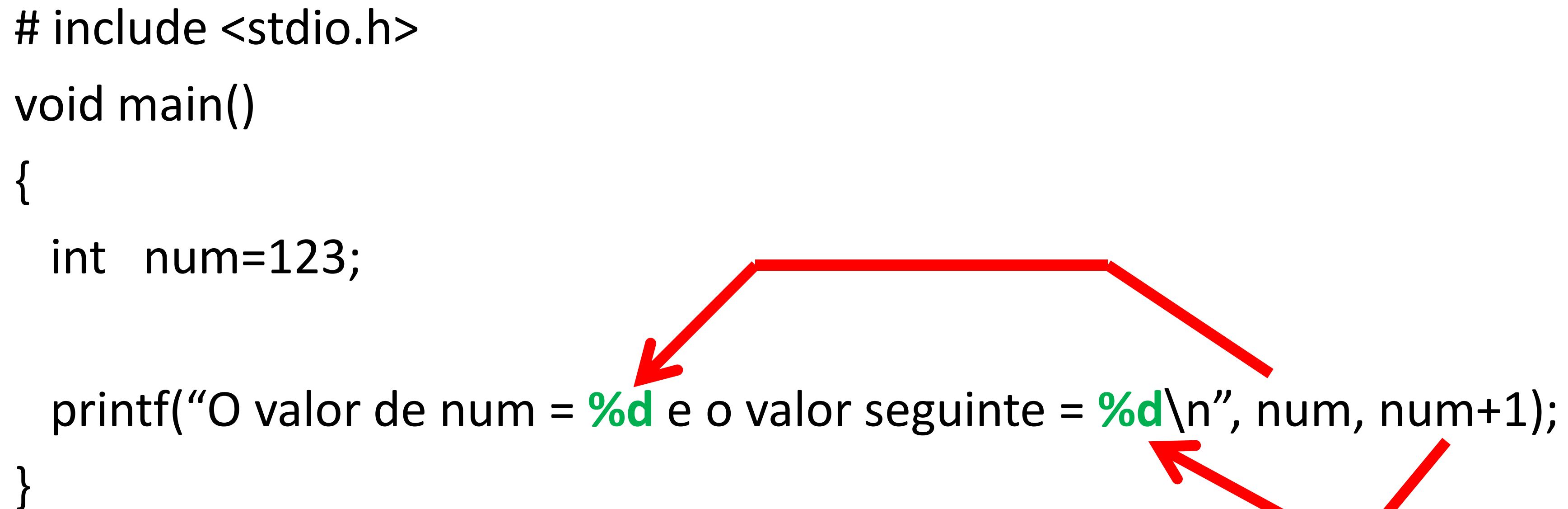
%	What it does
%d	Signed decimal integer
%o	Unsigned octal integer
%u	Unsigned decimal integer
%X	Unsigned Hexadecimal Integer
%f	Signed value of the form [-]dddd.dddd. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
%c	Single character
%s	String

O que Faz?

```
# include <stdio.h>
main()
{
    printf (“Isto\na\nfinal ate e\n\”facil\”\n”);
}
```

Inteiros – escrita

```
# include <stdio.h>
void main()
{
  int   num=123;

  printf("O valor de num = %d e o valor seguinte = %d\n", num, num+1);
}
```



Como Ler do Teclado

- A leitura é feita para variáveis e tem também de respeitar o tipo.
- C
  - scanf
- C++
  - cin >> variavel_raio
- JAVA
  - name = in.nextLine();
  - num = in.nextInt();
- PASCAL
  - read(var)

Comando scanf()

```
# include <stdio.h>
main()
{
  int   num;

    printf(“Introduza um Nº:  ”);
    scanf (“%d”, &num);
    printf(“O Nº introduzido foi %d\n”, num);
}
```

Notar na colocação de **&** antes do nome da variável

Comando scanf()

- A string de controlo do scanf é muito poderosa e permite fazer análise sintática.
- Caso faça *scanf("H:%d",&var);* tenho de colocar no input algo como ***H:234.***
- Caso não o faça o scanf devolverá o valor de 0 (false).

Char – A função getchar()

```
# include <stdio.h>
main()
{
  char   ch;

    printf (“Introduza um caracter: “);
    ch = getchar ();
    printf (“O caracter introduzido foi ‘%c’ \n”, ch);
}
```

evita-se o uso de **%c**

Comentários

- Um bom programa deve ter comentários para ajudar a sua compreensão e para no futuro percebermos algum passo menos trivial.

- Os comentários são texto normal incluído entre

  /*   */

```
/***********************************************************
 * Programa: Fazer a subtracção de dois números   *
 * Autor: João Semana                             *
 * Data:     07/10/2007                           *
 ***********************************************************/
```

Tipos de dados

- Existem 4 tipos de dados básicos:
  - Caracteres ***char***
  - Inteiros ***int***
  - Reais ***float***
  - Apontadores (para ver mais tarde)

- Tipos Compostos
  - Vectores
  - Estruturas

Inteiros - Operações

Operação	Descrição	Exemplo	resultado
+	Soma	21 + 4	25
-	Subtracção	21 – 4	17
*	Multiplicação	21 * 4	84
/	Divisão	21 / 4	5
%	Resto da divisão inteira (Mod)	21 % 4	1

Variáveis

- Como já vimos correspondem a posições de memória
- Nas linguagens de alto nível o nome de um avariável "esconde" do programador o modo como o acesso é feito a memória. O programador apenas utiliza o nome e não se preocupa com o detalhe de saber onde está a variável representada na memória.
- Cada uma delas está definida de acordo com um tipo
- O tipo da variável define o conjunto de operações que podem ser aplicadas sobre elas

Nomes das variáveis

- Letras do alfabeto, dígitos e o *underscore*
- O primeiro caracter não pode ser dígito
- *Case sensitive*

- O nome deve ser descritivo do que armazena
- Existem regras a seguir que são universalmente usadas pelos programadores de C.
- Exemplo:

```
nome_principal_cliente
nomePrincipalCliente
```

Exemplos de definição

```
int       i;
char       ch1,   novo_char, aux_char;
float     pi, raio, perimetro;
double    total;
```

ATRIBUIÇÃO de VALOR a uma Variável

- A instrução de afectação é uma das instruções fundamentais de qualquer linguagem de programação.
- Esta acção destina-se a que a variável passe a ter o conteúdo que se pretende.
- A uma variável pode-se atribuir um valor ou o valor de outra variável.
- C e JAVA
  - raio **=** 2; raio **=** raio + 2;
- PASCAL
  - raio **:=** 2; raio **:=** raio + 2;

Exercícios Em C e JAVA

- Fazer um programa que a partir de uma variável do tipo inteiro indique o digito mais significativo
- A partir de uma variável indique o número de digitos
- Fazer um programa que a partir do valor colocado numa variável do tipo inteiro indique quais os digitos que fazem parte desse número
- Alterar o programa anterior por forma a garantir que os digitos são mostrados da esquerda para a direita. Utilizar vectores.

Exer 1

```c
#include <stdio.h>

void  main ()
{
    int num = 5466;
    int quo = 0;

    do {
              quo = num % 10;
              num = num / 10;
    } while (num != 0);

    printf("O Dig. Mais Sig. e o %d",quo);
}
```

```java
public class mSD {
   public static void main(String[] args) {
      int num = 4568;
      int quo;

      do {
              quo = num % 10;
               num = num / 10;
       } while (num != 0);
       System.out.println("O Dig. Mais Sig. e " + quo);
   }
}
```

Exer 2

```c
#include <stdio.h>

void  main ()
{
        int num = 5466;
        int index = 0;
        int vect[10];

        do {
            vect[index++] = num % 10;
            num = num / 10;
        } while (num != 0);
        printf("Os Digitos sao ");
        do {
            printf("%d ",vect[--index]);
        } while (index != 0);
}
```

```java
public class mSDArrays {
    public static void main(String[] args) {
        int num = 4999;
        int index = 0;
        int[] vect;

        vect = new int[10];

        do {
            vect[index++] = num % 10;
            num = num / 10;
         } while (num != 0);
        System.out.print("Os digitos sao ");
        do {
            System.out.print(vect[--index] + " ");
        } while (index != 0);
    }
}
```

Operadores relacionais

Operador	Nome	Exemplo
==	Igualdade	a == b
>	Maior que	a > b
>=	Maior ou Igual que	a >= b
<	Menor que	a < b
<=	Menor ou Igual que	a <= b
!=	Diferente de	a != b

Operadores Lógicos

- Quando é preciso **mais do que uma** condição para tomar a decisão

<table>
<tr><th>Operador</th><th>Significado</th><th>Exemplo</th></tr>
<tr><td>&&</td><td>AND (E lógico)</td><td>x>=1 && x<=19</td></tr>
<tr><td>||</td><td>OR (OU lógico)</td><td>x==1 || x==2</td></tr>
<tr><td>!</td><td>NOT (Negação Lógica)</td><td>! (x == 3)</td></tr>
</table>

Operador *switch*

```
switch  (expressão)
{
     case  constante1  :   instrução1;
     case  constante2  :   instrução2;
     case  constante3  :   instrução3;
     [ default : instruções ]
 }
```

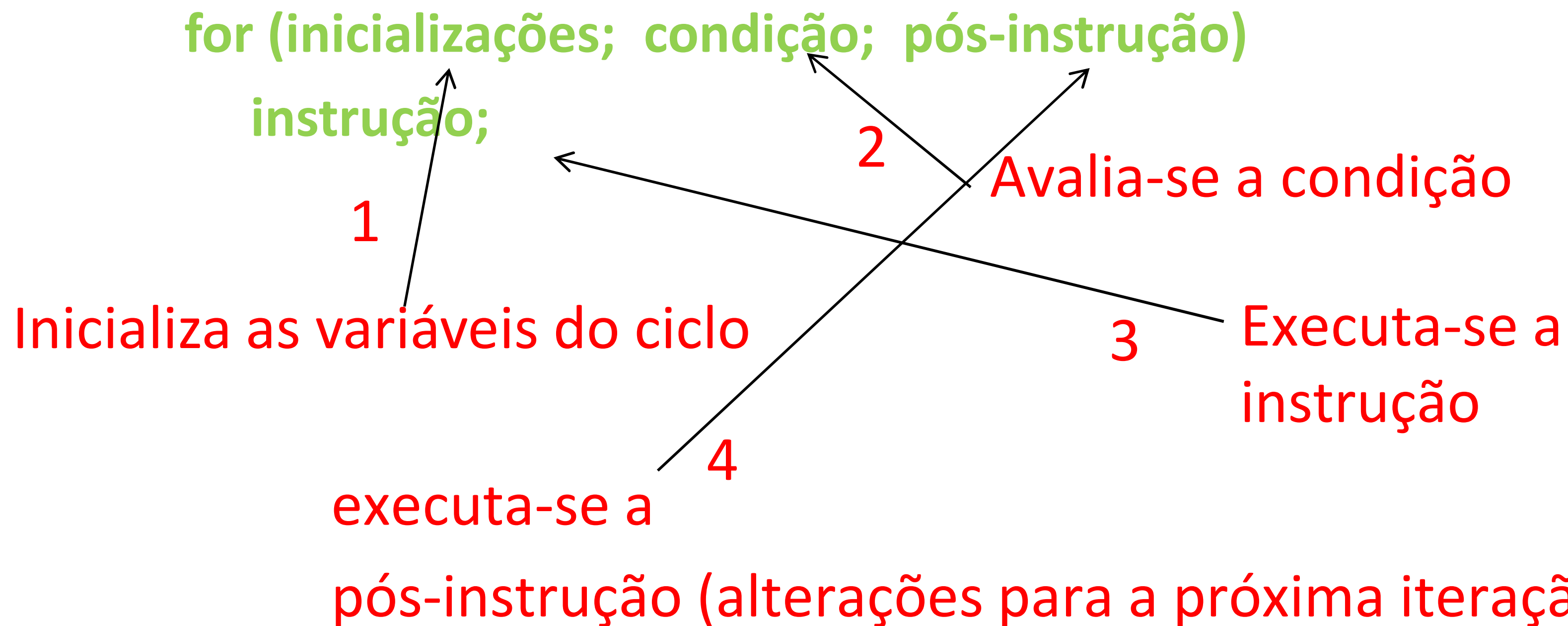
A expressão dá um resultado **char**, **int** ou **long**.

A instrução da constante que for igual a esse valor é executada

Ciclo *for*

Os ciclos permitem a repetição de instruções

O ciclo *for* permite a repetição da instrução quando o número de iterações é conhecido à partida



Escrever a tabuada do 1 ao 5

```
# include <stdio.h>
main()
{
  int    n, num;

    for ( n = 1; n <= 5; n = n + 1)
        {
            for (num = 0; num <= 10; num = num + 1)
                printf (“%2d   *    %2d   =   %2d\n “, n, num, n*num);
         }
      /*   Passar à próxima tabuada    */
}
```

Ciclos *while* e *for*

Em C qualquer ciclo **for** pode ser escrito como um ciclo **while**

```
for (inicializações;  condição;  pós-instrução)
    instrução;

inicializações;
while (condição)
  {
    instrução;
    pós-instrução
  }
```

Ciclos *do...while*

O teste é realizado no fim do corpo

Este ciclo é sempre executado pelo menos uma vez

```
do
    instrução;
while (condição);
```

É bom para se processarem menus (em que o teste se faz no fim...)

Instrução *break*

Termina o ciclo, continuando o programa na instrução imediatamente a seguir ao ciclo.

O que faz este programa?

```
# include <stdio.h>
main()
{
  int    i;

    for ( i = 1; i <= 100; i = i + 1)
       if (i == 30)
          break;
       else
          printf (“%2d\n “, 2 * i);
     printf (“Fim do ciclo\n“);
}
```

Instrução *continue*

Termina a execução do bloco de instruções passando logo para a próxima iteração

Operador *switch*

```
switch  (expressão)
{
    case  constante1  :   instrução1;
    case  constante2  :   instrução2;
    case  constante3  :   instrução3;
    [ default : instruções ]
 }
```

A expressão dá um resultado **char**, **int** ou **long**.

A instrução da constante que for igual a esse valor é executada

O que faz este programa?

```
# include <stdio.h>
main()
{
  int    i;

    for ( i = 1; i <= 100; i = i + 1)
       if (i == 60)
          break;
       else
          if (i % 2 == 1)    /*  Se i for impar  */
            continue;
          else
            printf (“%2d\n “, i);
}
```

Primos (Brute Force)

```
/* Programa para Calcular os Numeros Primos por Brute Force */
#include <stdio.h>

void  main ()
{
        int num; int j;
        int nPrimo = 0;       /*FALSE*/
        int qte = 0;

        for (num=2; num<102; num++) {
                for (j=2; j<num; j++) {
                        if ((num % j) == 0) {
                                nPrimo = 1;
                                break;
                        }
                }
                if (!nPrimo) {  printf("%d\n",num); qte++;  }
                nPrimo = 0;
        }
        printf("Existem %d primos",qte);
}
```

EXERCÍCIOS

- Fazer um programa que leia da linha de comando os seguintes tipos de comandos:
  - h23 m56 x
  - m45 h34 x
- No final da entrada deve escrever o número de horas e o número de minutos

Exercício

- Fazer um algoritmo capaz de calcular o desvio padrão:
  - A fórmula é $\sigma = \sqrt{\frac{1}{4.0}\sum_{i=1}^{5}\left(x_i - \bar{x}\right)^2}$
  - Os 5 números devem ser introduzidos através do teclado.

Funções e Procedimentos em C

- As funções servem para construir programas de uma forma mais modular
- Já se utilizaram funções em **Programação de Microprocessadores**
- Ex: *printf*, *scanf*, *getchar*, *putchar*, etc.

Características de uma função

- Tem de ter um nome único
- Pode ser invocada a partir de outras
- Deve comportar-se como uma caixa negra: não interessa como funciona, mas apenas interessa o seu resultado
- O seu código deve ser o mais independente possível do resto do programa
- Pode retornar um valor como resultado do seu trabalho

Argumentos e parâmetros

Cada função necessita de saber o tipo de cada um dos parâmetros.

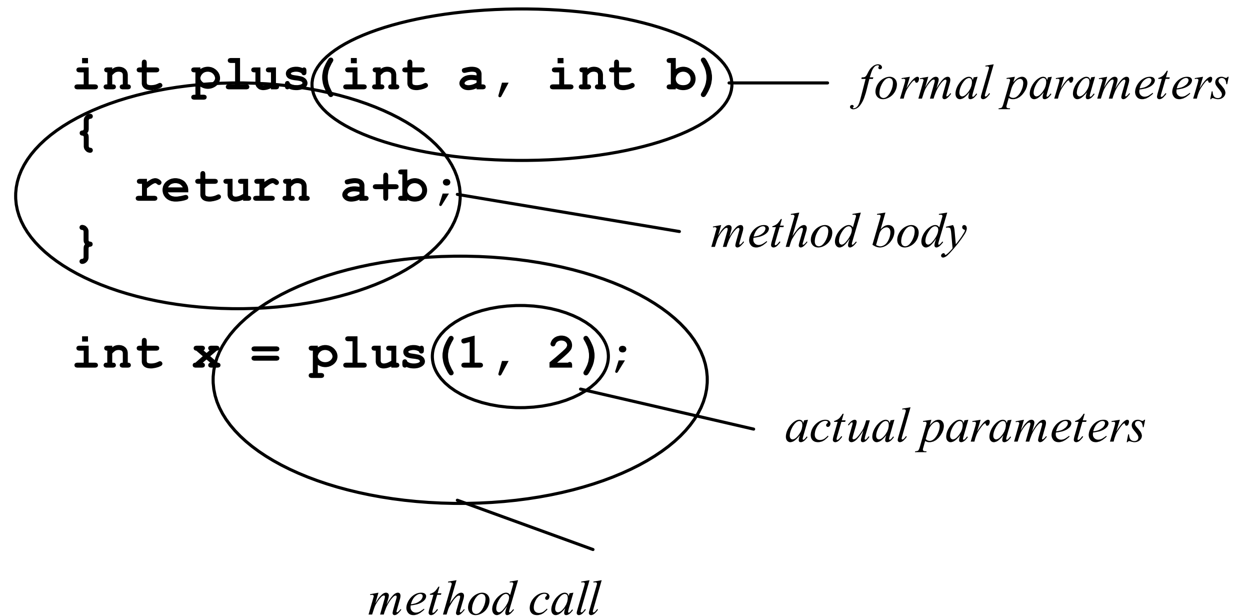
Um parâmetro não é mais do que uma variável local da função

Qualquer expressão válida de C pode ser enviada como parâmetro para uma função

Argumentos e parâmetros

```
funcao (char ch, int n, float x)
{
...
}

main ()
{
    ...
    funcao ('A', 123, 23.456);
    ...
}
```

Passagem Parâmetros



- How are parameters passed?
- Looks simple enough…
- We will see seven techniques

Outline

- 18.2 Parameter correspondence
- Implementation techniques
  - 18.3 By value
  - 18.4 By result
  - 18.5 By value-result
  - 18.6 By reference
  - 18.7 By macro expansion
  - 18.8 By name
  - 18.9 By need
- 18.10 Specification issues

Parameter Correspondence

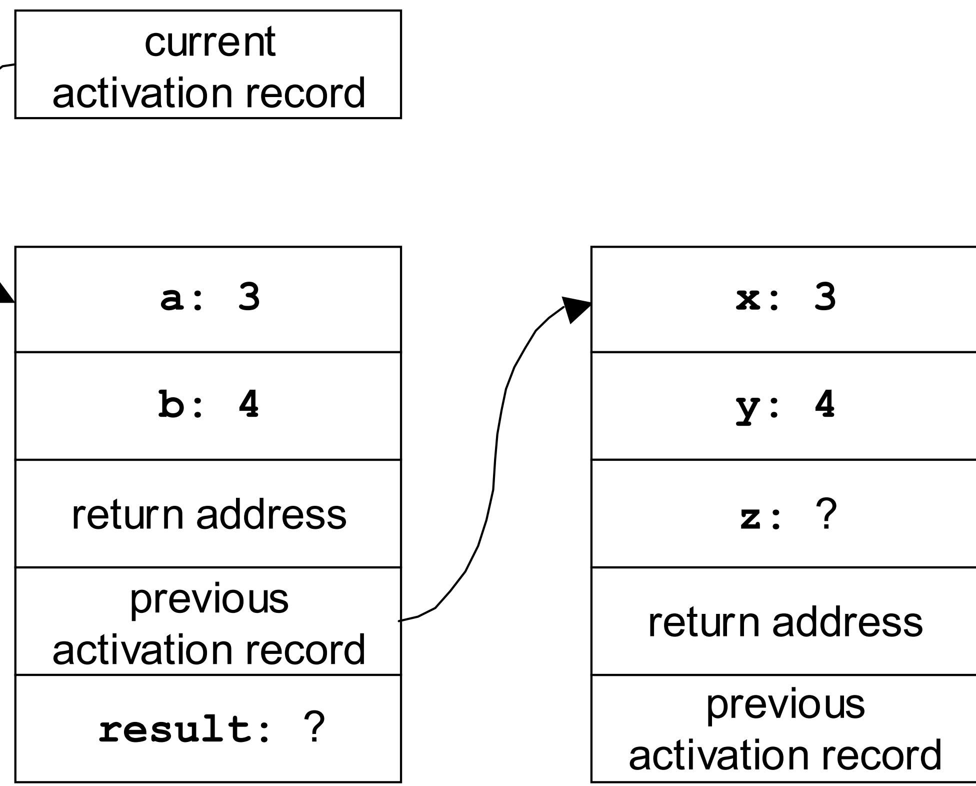
- A preliminary question: how does the language match up parameters?
- That is, which formal parameters go with which actual parameters?
- Most common case: *positional parameters*
  - Correspondence determined by positions
  - *n*th formal parameter matched with *n*th actual

By Value

For by-value parameter passing, the formal parameter is just like a local variable in the activation record of the called method, with one important difference: it is initialized using the value of the corresponding actual parameter, before the called method begins executing.

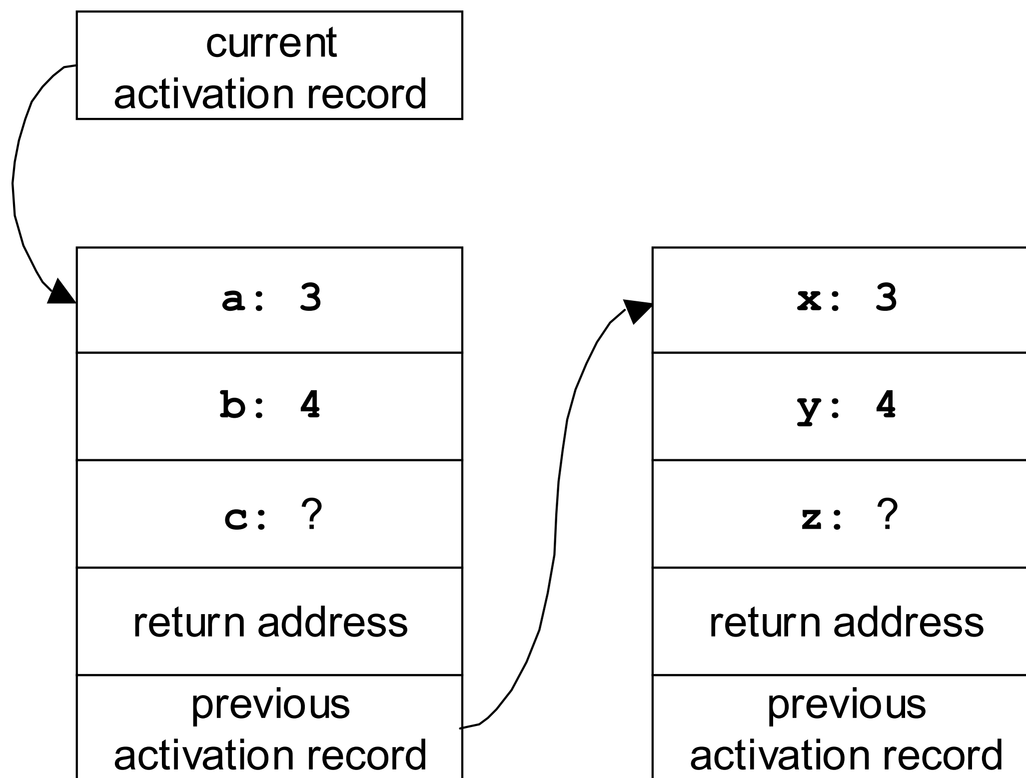
- Simplest method
- Widely used
- The only method in real Java

```
int plus(int a, int b) {
  a += b;
  return a;
}

void f() {
  int x = 3;
  int y = 4;
  int z = plus(x, y);
}
```



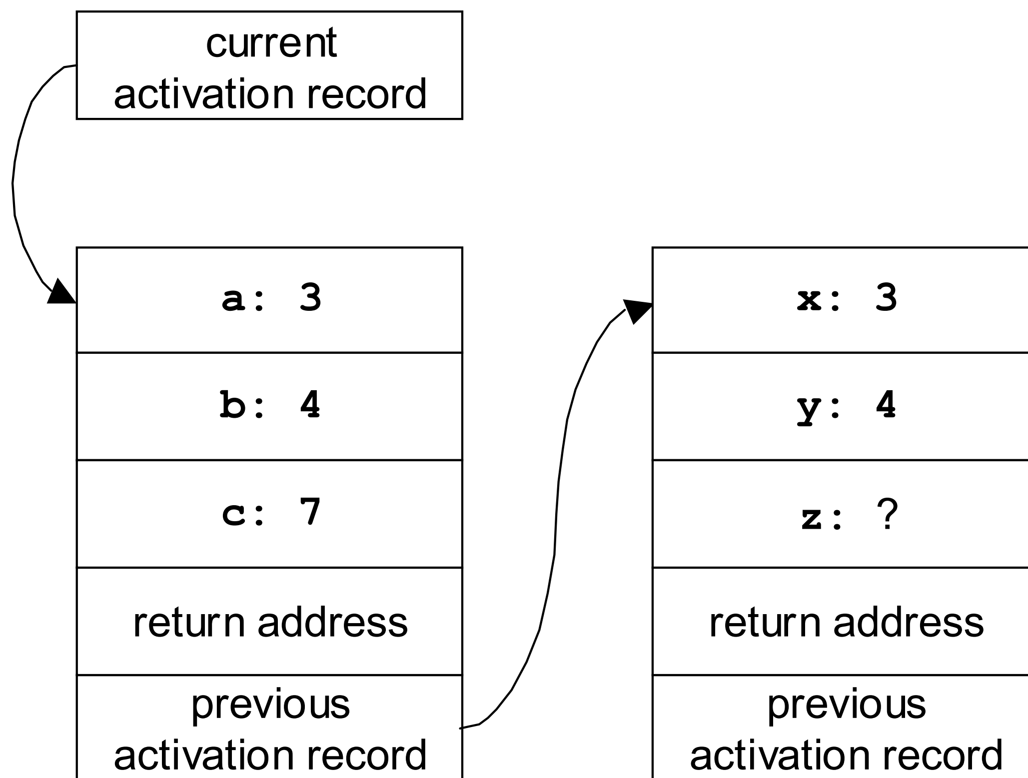
When **plus** is starting

```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



When **plus** is starting

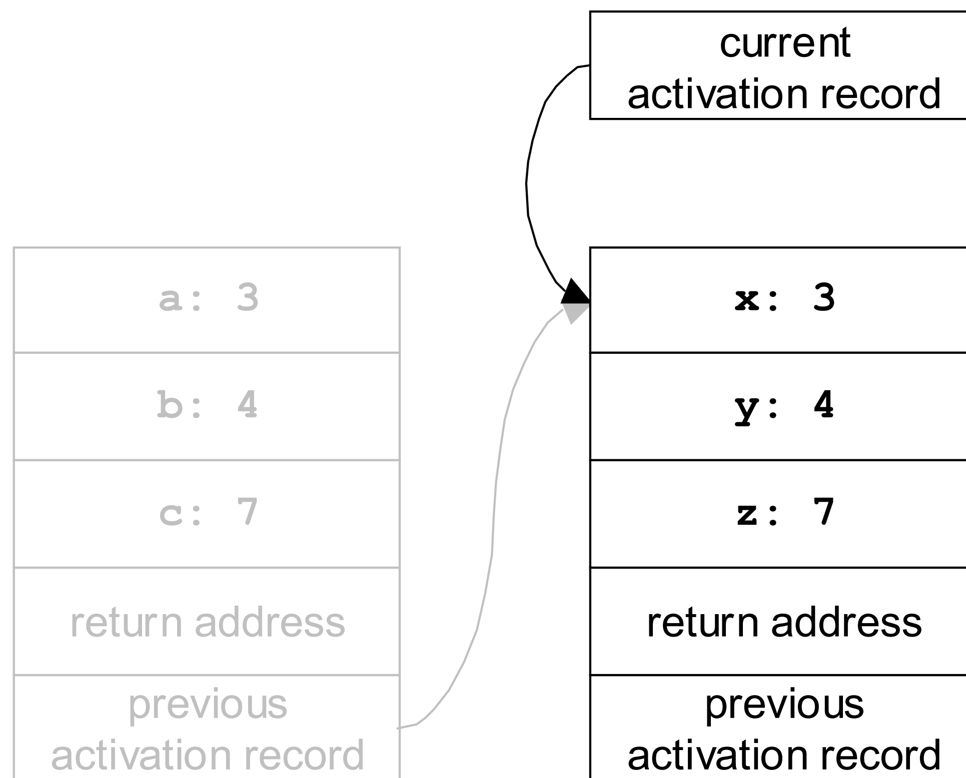
```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```



When **plus** is ready to return

```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```

When **plus** has returned



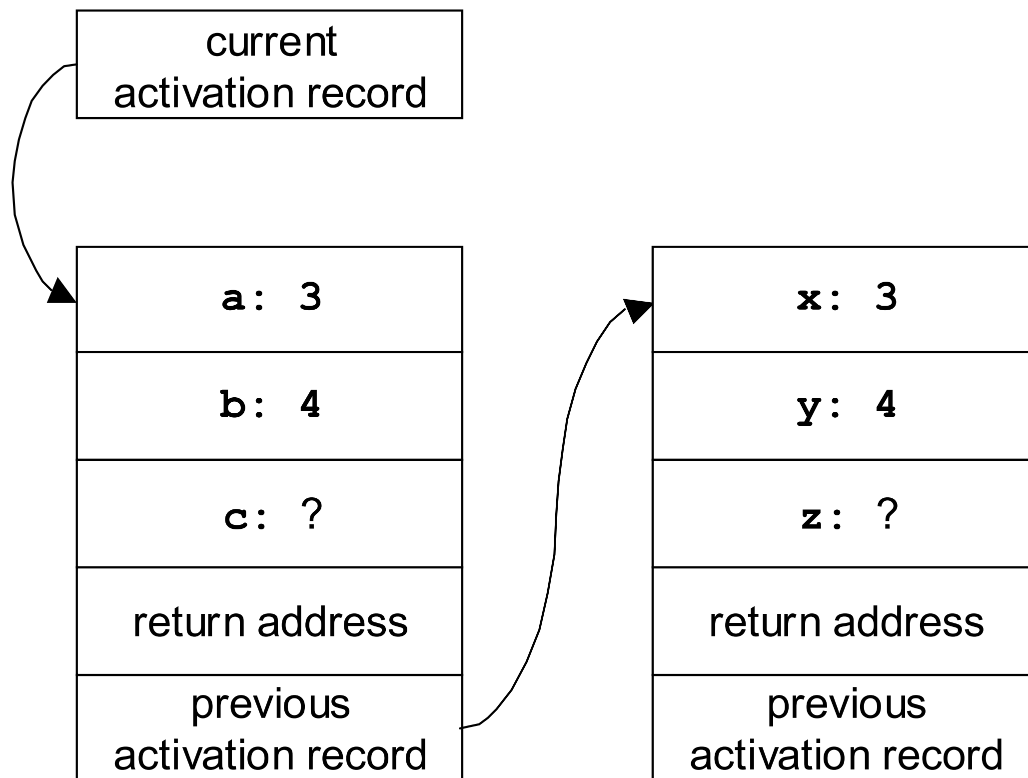
By Result

For by-result parameter passing, the formal parameter is just like a local variable in the activation record of the called method—it is uninitialized. After the called method finished executing, the final value of the formal parameter is assigned to the corresponding actual parameter.

- Also called *copy-out*
- Actual must have an lvalue
- Introduced in Algol 68; sometimes used for Ada

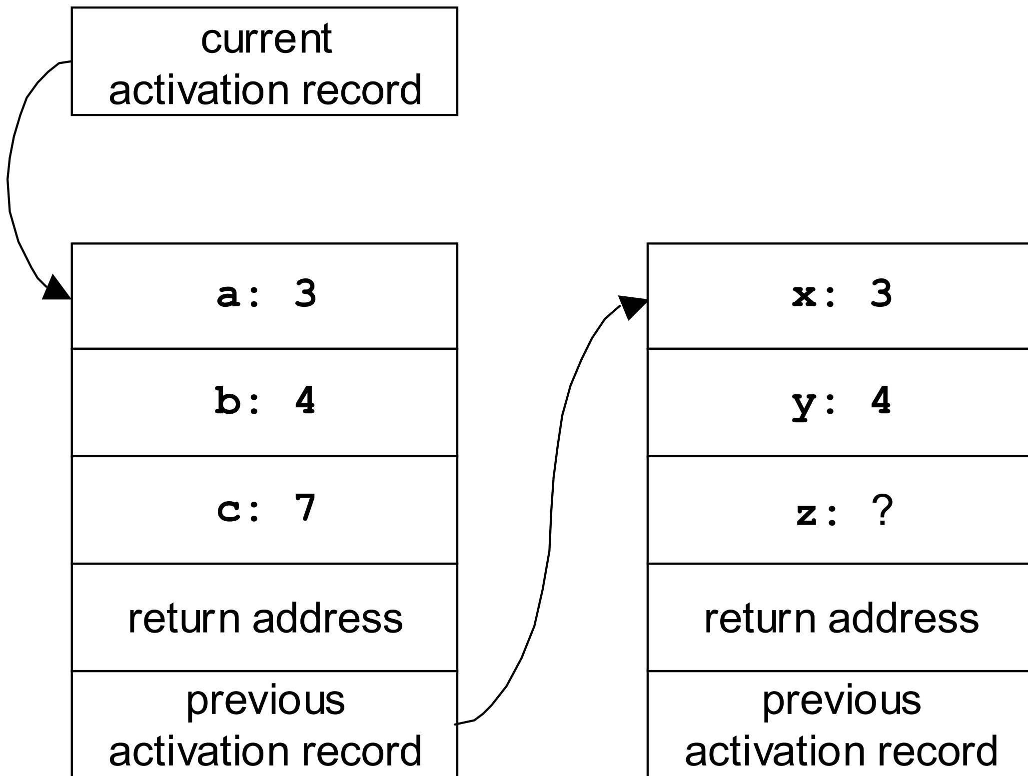
```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```

When **`plus`** is starting



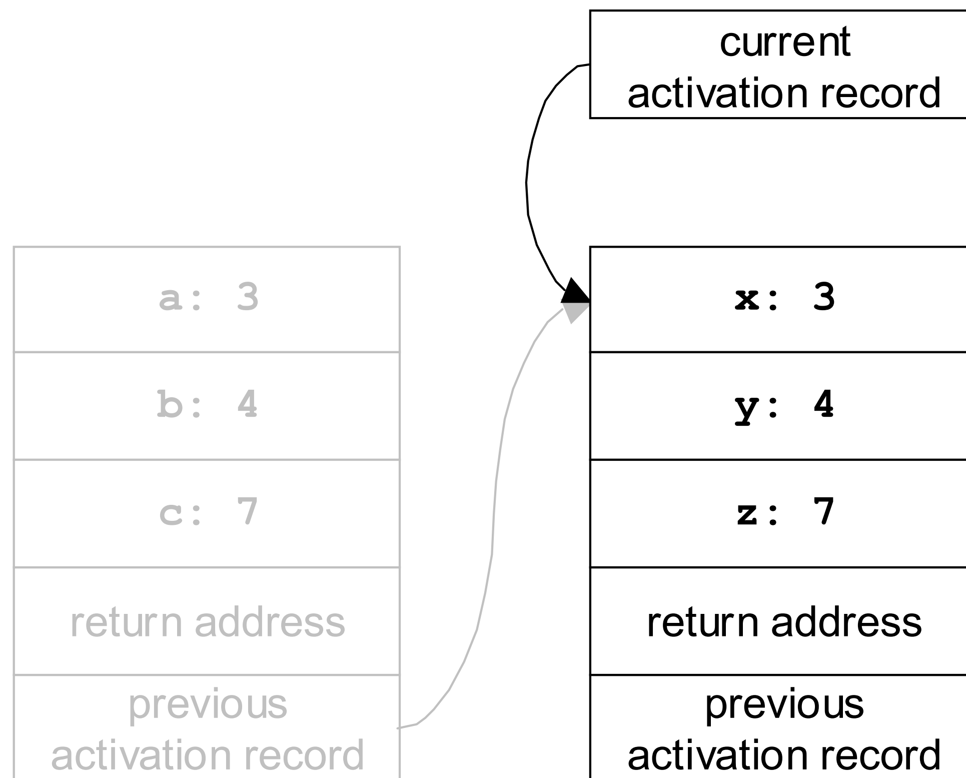
```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```

When **plus** is ready to return



```
void plus(int a, int b, by-result int c) {
  c = a+b;
}
void f() {
  int x = 3;
  int y = 4;
  int z;
  plus(x, y, z);
}
```

When **plus** has returned

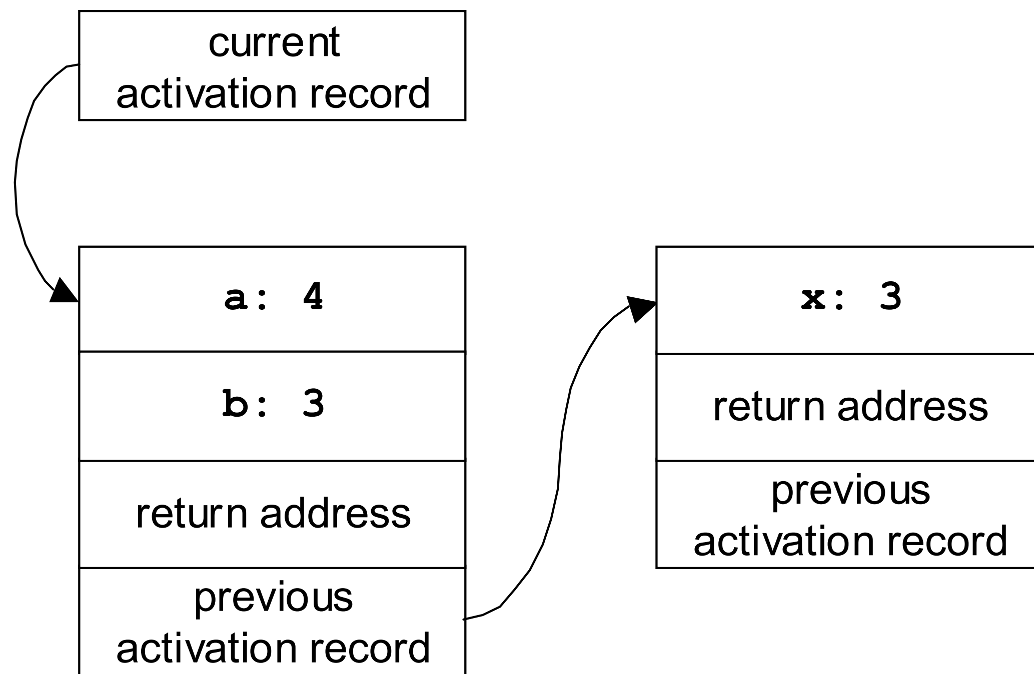


By Value-Result

For passing parameters by value-result, the formal parameter is just like a local variable in the activation record of the called method. It is initialized using the value of the corresponding actual parameter, before the called method begins executing. Then, after the called method finishes executing, the final value of the formal parameter is assigned to the actual parameter.

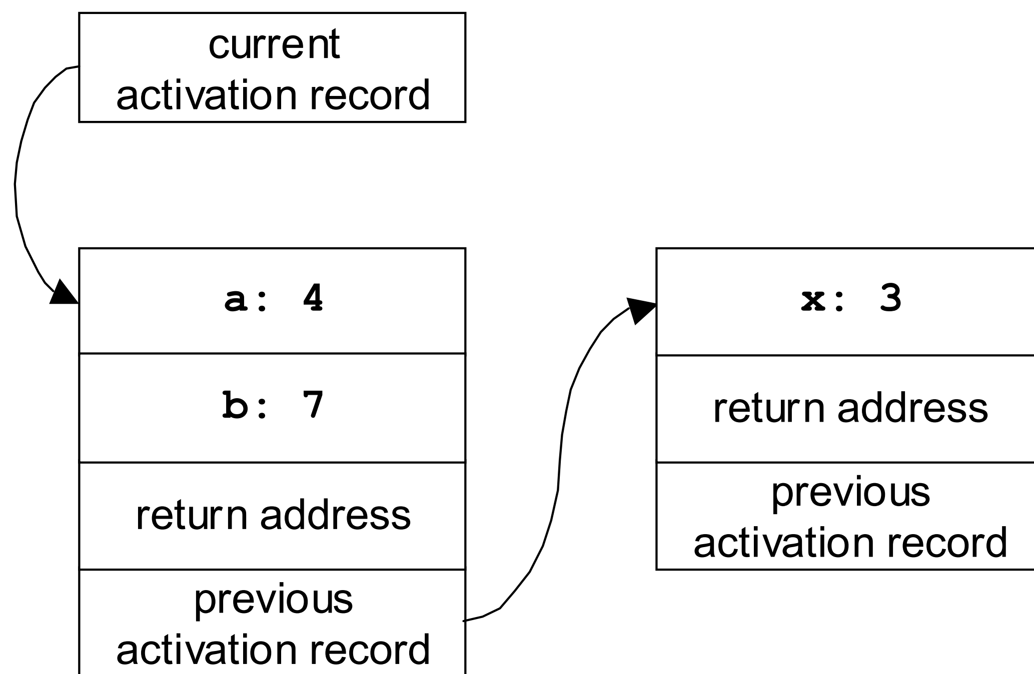
- Also called *copy-in/copy-out*
- Actual must have an lvalue

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



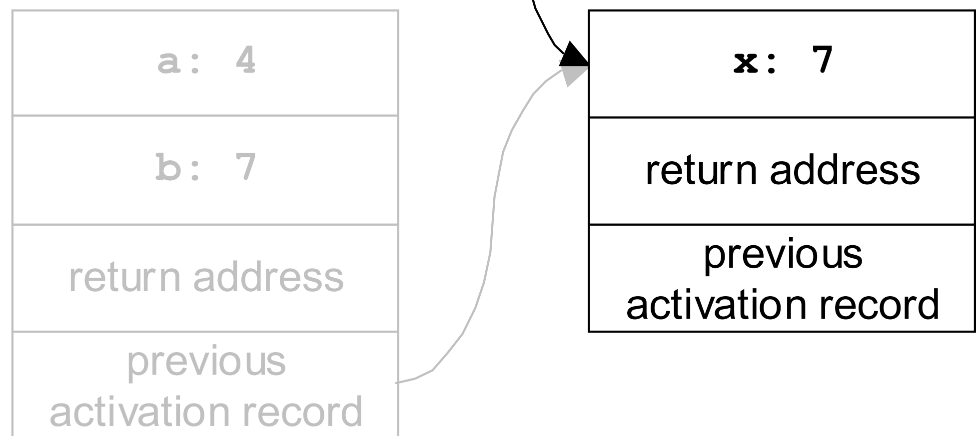
When **plus** is starting

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



When **plus** is ready to return

```
void plus(int a, by-value-result int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



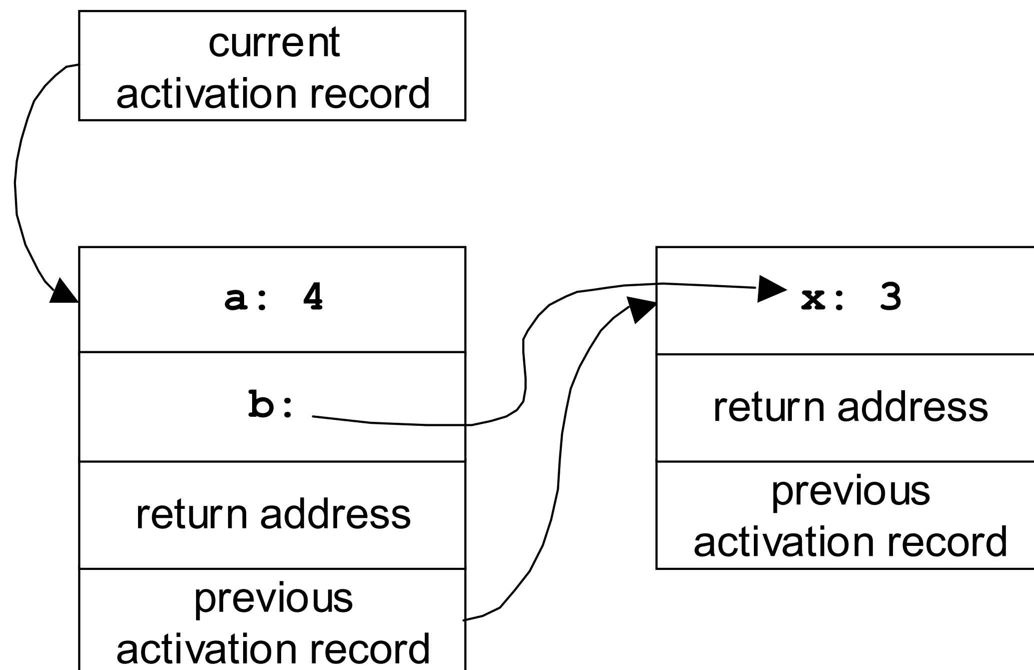
When **plus** has returned

By Reference

For passing parameters by reference, the lvalue of the actual parameter is computed before the called method executes. Inside the called method, that lvalue is used as the lvalue of the corresponding formal parameter. In effect, the formal parameter is an alias for the actual parameter—another name for the same memory location.

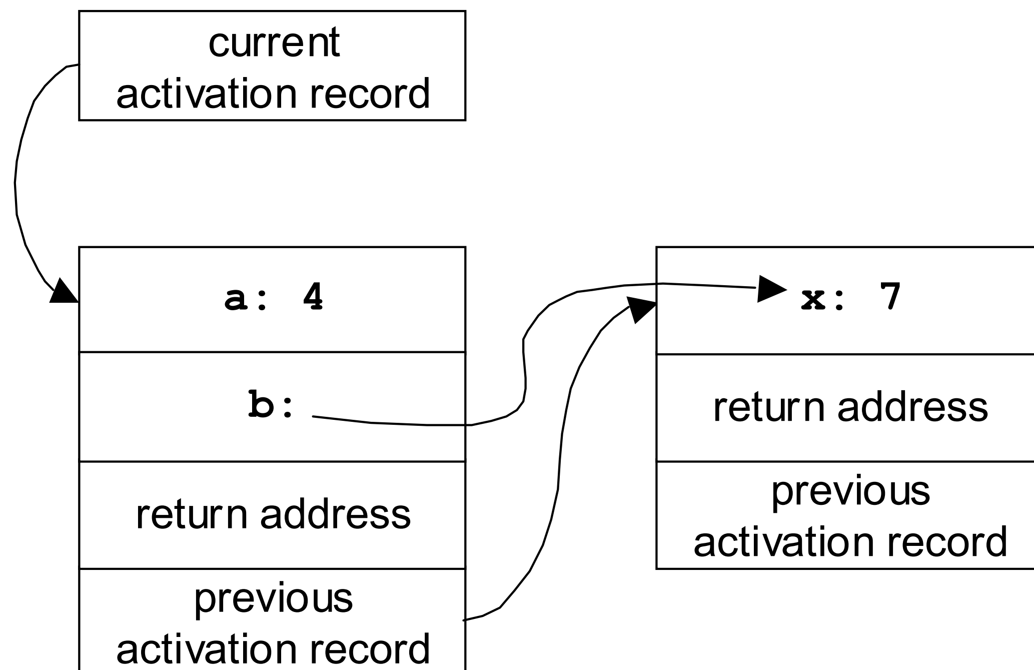
- One of the earliest methods: Fortran
- Most efficient for large objects
- Still frequently used

```
void plus(int a, by-reference int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



When **plus** is starting

```
void plus(int a, by-reference int b) {
  b += a;
}
void f() {
  int x = 3;
  plus(4, x);
}
```



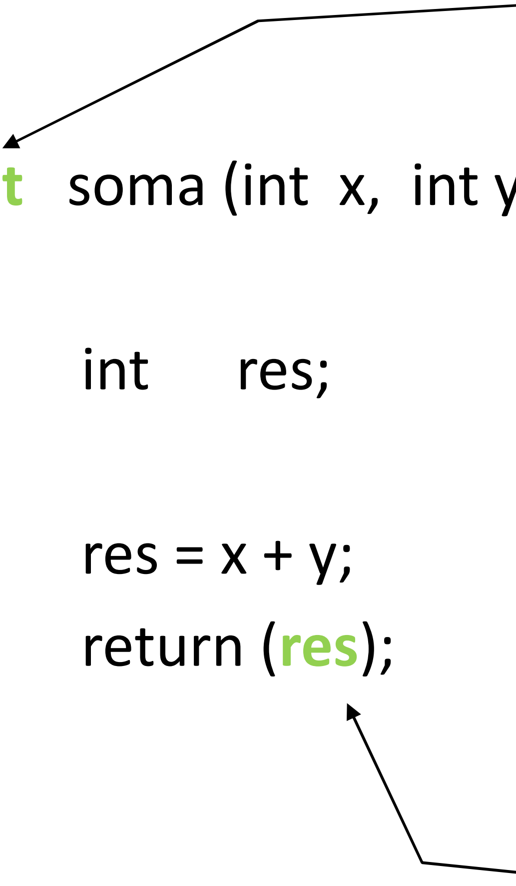
When **`plus`** has made the assignment

Valor de retorno

Tipo de retorno

```
int   soma (int  x,  int y);
{
      int      res;

      res = x + y;
      return (res);
}
```

Valor a retornar



Valor de retorno

```
int   soma (int  x,  int y);
{
       return x + y;
}
```

Exercícios com Funções

- Calcular factorial
- Calcular Fibonacci

Primitive vs. Constructed Types

- Qualquer tipo que um programa pode utilizar mas não pode definir é um *primitive type* da linguagem
- Qualquer tipo que um programa pode definir como seu é um *constructed type*
- Alguns primitive types in C: **`int`**, **`real`**, **`char`**
- A constructed type: **`int vector[]`**

Primitive Types

- The definition of a language says what the primitive types are
- Some languages define the primitive types more strictly than others:
  - Some define the primitive types exactly (Java)
  - Others leave some wiggle room—the primitive types may be different sets in different implementations of the language (C, ML)

Comparing Integral Types

C:

`char`
`unsigned char`
`short int`
`unsigned short int`
`int`
`unsigned int`
`long int`
`unsigned long int`

No standard implementation, but longer sizes must provide at least as much range as shorter sizes.

Java:

`byte` (1-byte signed)
`char` (2-byte unsigned)
`short` (2-byte signed)
`int` (4-byte signed)
`long` (8-byte signed)

Scheme:

`integer`
Integers of unbounded range

Constructed Types

- Additional types defined using the language
- Today: enumerations, tuples, arrays, strings, lists, unions, subtypes, and function types
- For each one, there is connection between how *sets* are defined mathematically, and how *types* are defined in programming languages

Making Sets by Enumeration

- Mathematically, we can construct sets by just listing all the elements:

$$S = \{a, b, c\}$$

Making Types by Enumeration

- Many languages support *enumerated types*:

```
C:       enum coin {penny, nickel, dime, quarter};
Ada:     type GENDER is (MALE, FEMALE);
Pascal:  type primaryColors = (red, green, blue);
ML:      datatype day = M | Tu | W | Th | F | Sa | Su;
```

- These define a new type (= set)
- They also define a collection of named constants of that type (= elements)

Representing Enumeration Values

- A common representation is to treat the values of an enumeration as small integers
- This may even be exposed to the programmer, as it is in C:

```
enum coin { penny = 1, nickel = 5, dime = 10, quarter = 25 };

enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
```

Making Sets by Tupling

- The Cartesian product of two or more sets defines sets of tuples:

$$S = X \times Y$$
$$= \left\{ \left(x, y \right) \mid x \in X \wedge y \in Y \right\}$$

Making Types by Tupling

- Some languages support pure tuples:

```
fun get1 (x : real * real) = #1 x;
```

- Many others support record types, which are just tuples with named fields:

C:

```
struct complex {
  double rp;
  double ip;
};
```

ML:

```
type complex = {
  rp:real,
  ip:real
};
fun getip (x : complex) = #ip x;
```

Representing Tuple Values

- A common representation is to just place the elements side-by-side in memory
- But there are lots of details:
  - in what order?
  - with “holes” to align elements (e.g. on word boundaries) in memory?
  - is any or all of this visible to the programmer?

Example: ANSI C

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member…

Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction...

The C Programming Language, 2nd ed.
Brian W. Kernighan and Dennis M. Ritchie

Operations on Tuple Values

- Selection, of course:

C: `x.ip`
ML: `#ip x`

- Other operations depending on how much of the representation is exposed:

C:

```
double y = *((double *) &x);
struct person {
  char *firstname;
  char *lastname;
} p1 = {"marcia","brady"};
```

Sets Of Vectors

- Fixed-size vectors:

$$\begin{aligned} S &= X^n \\ &= \{(x_1, \ldots, x_n) \mid \forall i \,.\, x_i \in X\} \end{aligned}$$

- Arbitrary-size vectors:

$$\begin{aligned} S &= X^* \\ &= \bigcup_i X^i \end{aligned}$$

Types Related To Vectors

- Arrays, strings and lists
- Like tuples, but with many variations
- One example: indexes
  - What are the index values?
  - Is the array size fixed at compile time?

Index Values

- Java, C, C++:
  - First element of an array **`a`** is **`a[0]`**
  - Indexes are always integers starting from 0
- Pascal is more flexible:
  - Various index types are possible: integers, characters, enumerations, subranges
  - Starting index chosen by the programmer
  - Ending index too: size is fixed at compile time

Pascal Array Example

```
type
  LetterCount = array['a'..'z'] of Integer;
var
  Counts: LetterCount;

begin
  Counts['a'] = 1
  etc.
```

Types Related To Vectors

- Many variations on vector-related types:
  - What are the index values?
  - Is array size fixed at compile time (part of static type)?
  - What operations are supported?
  - Is redimensioning possible at runtime?
  - Are multiple dimensions allowed?
  - Is a higher-dimensional array the same as an array of arrays?
  - What is the order of elements in memory?
  - Is there a separate type for strings (not just array of characters)?
  - Is there a separate type for lists?

O que é um vector?

Já não é um tipo básico!

Permite processar conjuntos de dados do mesmo tipo

Exemplos:

200	450	700	300	150	650	320	...

X		O
	X	
		O

Declaração de Vectores

Declaram-se como se fosse uma variável simples

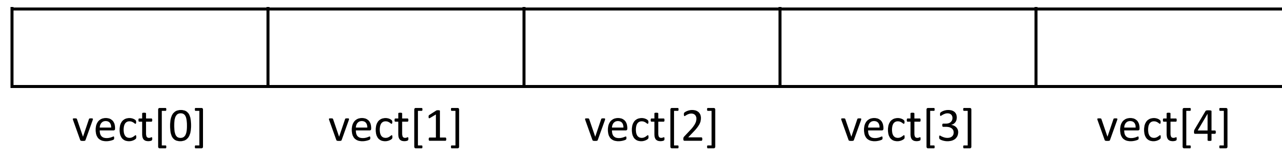
tipo nome_variavel [nº de elementos]

```
int   g [20];
```

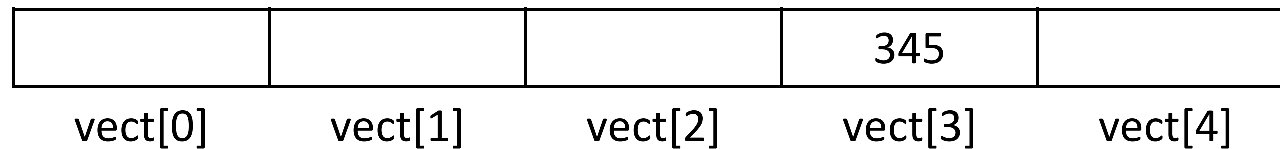
Os índices dos vectores começam em **zero** e vão até **n-1**

Declaração de Vectores

Vector com cinco posições



vect [3] = 345;



Inicialização automática

É possível inicializar automaticamente todos os elementos de um vector aquando da declaração

tipo var[n] = {$valor_1$, $valor_2$, … , $valor_n$ } ;

Exemplo:

char vogal [5] = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’} ;

Inicialização automática

Se o vector tiver **n** elementos e só forem inicializados **k**, com **k < n**, os restantes são inicializados a ZERO

```
int   v[10] = {10, 20, 30} ;
```

Inicialização automática

Se o vector tiver **n** elementos e só forem inicializados **k**, com **k < n**, os restantes são inicializados a ZERO

```
int   v[10] = {10, 20, 30} ;
```

Igual a:

```
int   v[10] = {10, 20, 30, 0, 0, 0, 0, 0, 0, 0} ;
```

Quando **não dá jeito** ter o índice ZERO no vector usa-se mais uma posição e não se usa a posição zero.

Inicialização automática

Às vezes não se coloca a dimensão do vector.

O compilador calcula-a pelo número de inicializações

tipo var [ ] = {$valor_1$, $valor_2$, … , $valor_n$ } ;

Vai criar n elementos:

```
int   v[  ];
```

Declaração incorrecta

Ex: Funções para inicializar vectores

Dois vectores para inicializar a zero

```
int v [10];
int x [20];
```

Como têm tamanhos diferentes escrevem-se duas funções

```
void inic1 (int s [10])
{
        int    i;
        for (i = 0; i < 10; i++)
           s[i] = 0;
}

void inic1 (int s [20])
{
        int    i;
        for (i = 0; i < 20; i++)
           s[i] = 0;
}
```

```
main ()
{
        int    v [10] ;
        int    x [20] ;

       inic1 (v);
       inic2 (x);
}
```

```
main ()
{
	int   v [10] ;
	int   x [20] ;

	inic1 (v);
	inic2 (x);
}
```

Na verdade em C só interessa **o tipo** para a função e não o número de índices…

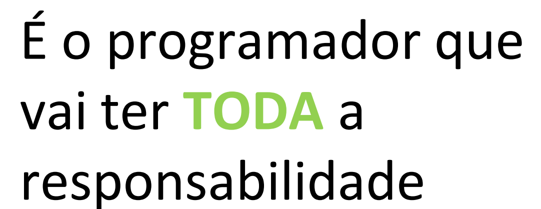
Dentro da função não é possível saber com quantos elementos o vector que foi passado no argumento foi declarado.

Isto é muito bom pois dá uma grande liberdade

… Mas é extremamente PERIGOSO

```
#include <stdio.h>
void inic (int s [  ], int n)
{
        int    i;
        for (i = 0; i < n; i++)
          s[i] = 0;
}

main ()
{         int    v [10] ;
          int    x [20] ;

          inic (v, 10);
          inic (x, 20);
}
```

```c
#include <stdio.h>
void inic (int s [  ], int n)
{
          int    i;
          for (i = 0; i < n; i++)
            s[i] = 0;
}

main ()
{          int    v [10] ;
           int    x [20] ;

           inic (v, 10);
           inic (x, 20);
}
```



É o programador que vai ter **TODA** a responsabilidade

Vectores multi-dimensionais

Não existe limite para o número de dimensões que um vector pode ter

Declaração de um vector com n dimensões

tipo vector [dim_1] [dim_2] [...] [dim_n]

Exemplo:

Jogo do Galo

Inicialização

```
int  v [2][4] = {{ 1, 2, 3, 4}, {11, 12, 13, 14}} ;
```

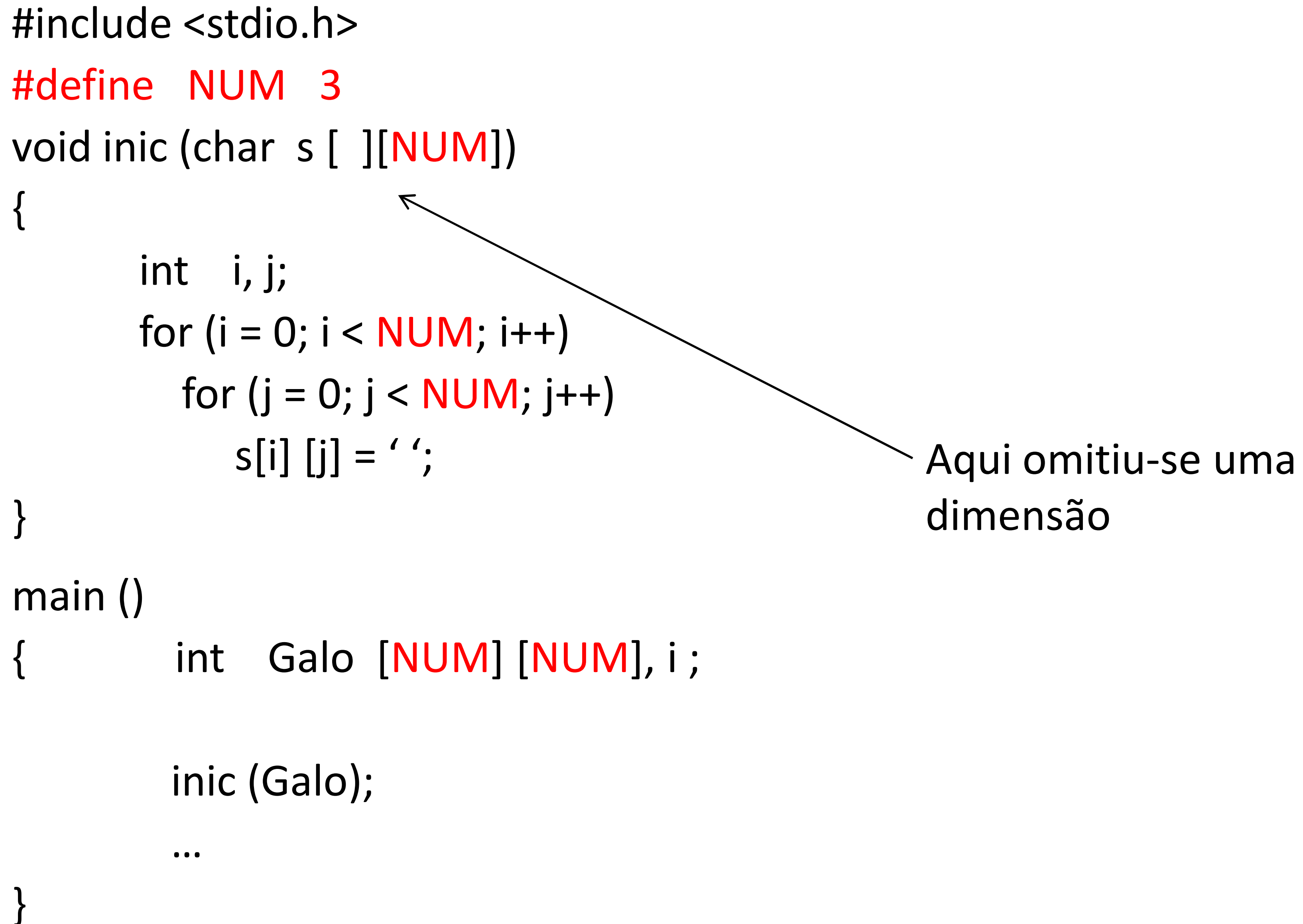
ou

```
int  v [ ][ ] = {{ 1, 2, 3, 4}, {11, 12, 13, 14}} ;     /*  ERRO  */
```

```
int  v [ ][4] = {{ 1, 2, 3, 4}, {11, 12, 13, 14}} ;
```

Só se pode omitir um valor (o mais à esquerda)

```
#include <stdio.h>
#define   NUM   3
void inic (char  s [  ][NUM])
{
          int    i, j;
          for (i = 0; i < NUM; i++)
             for (j = 0; j < NUM; j++)
                 s[i] [j] = ' ';
}
main ()
{            int    Galo  [NUM] [NUM], i ;

             inic (Galo);
             …
}
```



Introdução

Existem os tipos básicos: char, int, float, double

Aprendemos a agrupá-los em vectores (sempre do mesmo tipo)

As estruturas permitem agrupar elementos de **tipos diferentes**

Os elementos podem ser os básicos, mas também vectores, *strings*, apontadores, e mesmo estruturas

Introdução

Exemplo:

Dados sobre um indivíduo

```
int idade;
char sexo, estad_civil;
char Nome [50];
float salario;
int dia;
char mês [12];
int ano;
```

Estas variáveis não estão relacionadas entre si

Declaração de Estruturas

struct [nome_da_estrutura]

{

$tipo_1$ $campo_1$, $campo_2$;

…

$tipo_n$ campo;

};

```
struct  Data
 {   int   dia, ano;
     char mês [12];
 } ;
```

Para o compilador existe agora um novo **tipo**

Declaração de Estruturas

struct [nome_da_estrutura]

{

$tipo_1$ $campo_1$, $campo_2$;

…

$tipo_n$ campo;

} **v_1, v_2, v_n;**

```
struct  Data
 {   int   dia, ano;
     char mês [12];
 } ;
```

Declaração de Estruturas

struct [nome_da_estrutura]

{

$tipo_1$ $campo_1$, $campo_2$;

…

$tipo_n$ campo;

} **v_1, v_2, v_n;**

```
struct  Data
 {   int  dia, ano;
     char mês [12];
 }    d,  datas [100],   *ptr_data;
```

Declaração de Estruturas

struct [nome_da_estrutura]
{
$tipo_1$ $campo_1$, $campo_2$;
…
$tipo_n$ campo;
} ;

```
struct  Data
 {   int   dia, ano;
     char mês [12];
 } ;
 struct  Data     d, AnosTeresa, AnosPedro;
```

Acesso aos membros

Para se aceder aos membros usa-se o ponto (.)

Acesso aos membros

Para se aceder aos membros usa-se o ponto (.)

```
struct  Data
{   int   dia, ano;
    char mês [12];
} ;
struct  Data     AnosTeresa, AnosPedro;
```

Acesso aos membros

Para se aceder aos membros usa-se o ponto (.)

```
struct  Data
{   int   dia, ano;
    char mês [12];
} ;

struct  Data     AnosTeresa, AnosPedro;

AnosTeresa.dia = 20;
AnosTeresa.ano = 1991;
strcpy (AnosTeresa.mês, “Março”);
```

Inicialização automática

Pode também ser inicializada na declaração

struct nome var = { $valor_1$, $valor_2$, …, $valor_n$ }

Inicialização automática

Pode também ser inicializada na declaração

struct nome var = { valor$_1$, valor$_2$, …, valor$_n$ }

```
struct  Data
{   int  dia, ano;
    char mês [12];
}   AnosTeresa  =  { 20, 1991, “Março”};
```

Inicialização automática

Pode também ser inicializada na declaração

struct nome var = { $valor_1$, $valor_2$, …, $valor_n$ }

```
struct  Data
{    int   dia, ano;
     char mês [12];
} ;

struct Data AnosTeresa  =  { 20, 1991, “Março”};
```

Inicialização automática

Se a estrutura for um vector

```
struct  Data
{   int   dia, ano;
    char mês [12];
} ;
```

```
struct Data AnosAmigos []  =  {{20, 1991, “Março”}, {24, 1989,
 “Março”}, {13, 1994, “Janeiro”}};
```

Aqui o índice ficaria 3

Definições de tipos

A declaração da variável tem de ter sempre a palavra **struct**

Era bom usar apenas **uma palavra** para representar a estrutura…

typdef tipo_existente sinónimo

Não é criado um novo tipo. Apenas um novo nome para identificar o tipo.

Pode ser usado com qualquer tipo da linguagem

Definições de tipos

```
typedef   int   inteiro;

inteiro  a, b, c;

          struct  Data
{   int  dia, ano;
    char mês [12];
} ;
```

Definições de tipos

```
  typedef   int   inteiro;

  inteiro   a, b, c;

typedef   struct  Data
{   int   dia, ano;
    char mês [12];
}   DATA_NASC ;
```

Definições de tipos

```
  typedef   int   inteiro;

  inteiro  a, b, c;

typedef  struct  Data
{   int  dia, ano;
    char mês [12];
}  DATA_NASC ;

DATA_NASC   AnosTeresa, AnosPedro;
```

Definições de tipos

```
  typedef   int   inteiro;

   inteiro   a, b, c;

typedef   struct
{    int   dia, ano;
     char mês [12];
}   DATA_NASC ;

DATA_NASC    AnosTeresa, AnosPedro;
```

Definições de tipos

```
typedef   int   inteiro;

inteiro   a, b, c;

typedef   struct
{    int   dia, ano;
     char mês [12];
}   DATA_NASC ;

DATA_NASC    AnosTeresa, AnosPedro;
```

Atenção que isto não é uma variável...

Onde definir estruturas e typedef

Se a definição estiver dentro de uma função, apenas essa função conhece a definição

Para serem conhecidas no programa todo devem

ser definidas logo no início (depois dos **#include**)
ou num ficheiro (.h) de que se faça o **#include**

Estruturas dentro de estruturas

Quando uma estrutura contiver outra estrutura, esta última tem de já estar definida.

```
typedef   struct
{    int   dia, ano;
     char mês [12];
}   DATA_NASC;

typedef   struct
{    char             nome [100];
     char             sexo;
     DATA_NASC dt_nasc;
}   Pessoa;
```

Estruturas dentro de estruturas

```
typedef   struct
{    int   dia, ano;
     char mês [12];
}     DATA_NASC;

typedef   struct
{    char            nome [100];
     char            sexo;
     DATA_NASC   dt_nasc;
}   Pessoa;

Pessoa     João, Manuel, Teresa;

Teresa.dt_nasc.dia = 20;
```

Passagem para funções

Quando se chama, coloca-se a variável

Na função usa-se

struct nome   ou   nome (no caso do **typedef**)

```
#include <stdio.h>
#include <defsdata.h>   /* onde estão as definições de Pessoa */

void Mostrar (struct pessoa x)
{
     printf (“Nome            : %s\n”, x.nome);
     printf (“Género          : %c\n”, x.sexo);
     printf (“Dt. Nasc        : %d/%s/%d\n”, x.dt_nasc.dia,
                                   x.dt_nasc.mes, x.dt_nasc.ano);
}

main ()
{     struct pessoa Teresa = {“Teresa”, “F”, {20, 1991, “Março”}};

      Mostra (Teresa);
}
```

```
#include <stdio.h>
#include <defsdata.h>   /* onde estão as definições de Pessoa */

void Mostrar (Pessoa x)
{
    printf (“Nome           : %s\n”, x.nome);
    printf (“Género         : %c\n”, x.sexo);
    printf (“Dt. Nasc       : %d/%s/%d\n”, x.dt_nasc.dia,
                              x.dt_nasc.mes, x.dt_nasc.ano);
}

main ()
{     Pessoa Teresa = {“Teresa”, “F”, {20, 1991, “Março”}};

      Mostra (Teresa);
}
```

Operações sobre estruturas

Se x e y forem duas variáveis com a mesma estrutura

Pode-se fazer

```
x = y;
```