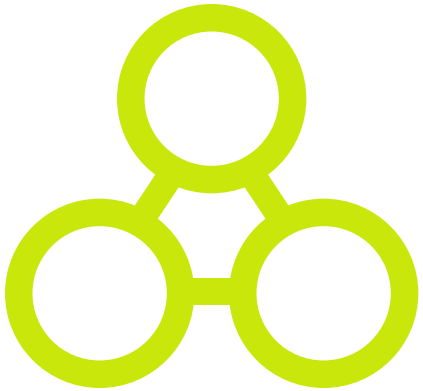


nemo

ontology & conceptual modeling research group

# Linguagens de Programação
# 2 – Amarrações

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

# Conteúdo do curso

- Introdução;
- Amarrações;
- Valores e tipos de dados;
- Variáveis e constantes;
- Expressões e comandos;
- Modularização;
- Polimorfismo;
- Exceções;
- Concorrência;
- Avaliação de linguagens.

---

- Estes slides foram baseados em:
  - Slides do prof. Flávio M. Varejão;
  - Livro “Linguagens de Programação – Conceitos e Técnicas” (Varejão);
  - Livro “Linguagens de Programação – Princípios e Paradigmas, 2a edição” (Tucker & Noonan).

# Conceito

- Amarração (*binding*) é uma associação entre entidades de programação. Ex.:
  - Uma variável e seu valor;
  - Um identificador e seu tipo.
- Enfoque na amarração de identificadores a entidades;
- Importância: a forma que uma LP faz a amarração define se a mesma é rígida ou flexível;
- Tucker & Noonan usam o termo “ligação” para se referir à amarração.

# Exemplo: variáveis

Quantas amarrações possui uma variável?

Amarração 4: a variável ocupa o endereço de memória 0x...

```
// Código em C:
int var = 100;
```

Amarração 2: a variável é do tipo `int`.

Amarração 1: a variável se chama var.

Amarração 3: a variável possui valor 100.

Qual é o tempo de cada amarração?

# Tempos de amarração (exemplos)

| Tempo de amarração | Identifi-cador | Entidade |
|---|---|---|
| Projeto da LP | * | Operação de multiplicação (C, C++, Java). |
| Projeto da LP | int | Intervalo do tipo inteiro (Java). |
| Implementação do compilador | int | Intervalo do tipo inteiro (C). |
| Compilação | Variável | Tipo da variável (C). |
| Execução | Variável | Tipo do objeto pertencente a classe polimórfica (C++, Java). |
| Ligação | Função | Código correspondente à função. |
| Carga do programa | Variável global | Posição de memória ocupada. |
| Execução | Variável local | Posição de memória ocupada. |

- Amarração pode ser estática (feita antes da execução e não muda) ou dinâmica (muda durante execução).

# Identificadores

- *Strings* (termos) definidas pelos programadores para servirem de referência a entidades de computação;
- Objetivam aumentar a legibilidade, redigibilidade e modificabilidade;
- LPs podem ser *case sensitive*:
  - C, C++, Java o são, Pascal e Basic não;
  - Afeta legibilidade, redigibilidade;
- LPs podem limitar o número máximo de caracteres:
  - Podem gerar erro ou ignorar excesso;
  - Versões iniciais de FORTRAN faziam isso;
- LPs podem restringir caracteres especiais nos nomes.

# Identificadores

- Podem ter significado especial:
  - Palavra reservada: não pode ser usada como identificador pelo programador (ex.: goto em Java);
  - Palavra-chave: tem significado pré-determinado na linguagem (ex.: goto em C, if em C e em Java);
  - Palavra pré-definida: tem significado, mas o mesmo pode ser redefinido (ex.: funções de uma API).

```
! Código válido em FORTRAN.
! São palavras-chave, mas não são palavras reservadas.
INTEGER REAL
REAL INTEGER
```

# Identificadores

- Outro exemplo, em Pascal:

```
program confuso;
const true = false;
begin

  (* ... *)

  if (a < b) = true then
     f(a)
  else
     g(b);

  (* ... *)

end.
```

# Ambientes de amarração

- A interpretação de comandos e expressões

```
/* Ex.: */          a = 5;                  g(a + 1);
```

  dependem do que denotam os identificadores utilizados nesses comandos e expressões;

- Um **ambiente** (*environment*) é um conjunto de amarrações;
- Cada amarração possui um determinado **escopo**, isto é, a região do programa onde a entidade é visível.

# Ambientes de amarração

- Um identificador pode estar amarrado a duas entidades distintas em um mesmo ambiente:
- Exemplo em C:

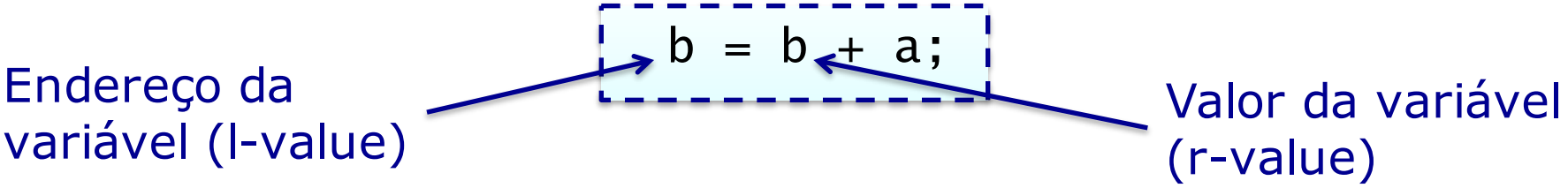
```
int a = 13;
void f() {
  int b = a;
  int a = 2;
  b = b + a;
}

// Qual o valor de b ao final de f()?
```

b = 15

- Em outra LP, resultado poderia depender do tempo de amarração. Se variáveis locais são amarradas no início do bloco, a poderia valer 0 (default) inicialmente.

# Ambientes de amarração

- Esse mesmo exemplo esconde uma duplicidade de entidades em um mesmo identificador:




- Algumas LPs exigem que isso seja feito explicitamente;
- Exemplo em ML:

```
b := !b + !a;
```

- Em C, C++, isso acontece com ponteiros somente.

# Escopo

- Segundo Tucker & Noonan:

> O escopo de um nome é a coleção de comandos que podem acessar essa ligação de nome.

- Exemplo em C:

```
void g() {
  int x = 0;                //
  int a = x + 2;            // Escopo de x.
  printf("%d\n", x);        //
}

// x inacessível fora de g()
```

# Escopo

- Estático (ou léxico):
  - Definição do subprograma;
  - Tempo de compilação;
  - Texto do programa.
- Dinâmico:
  - Chamada do subprograma;
  - Tempo de execução;
  - Fluxo de controle do programa.

# Escopo dinâmico

- Exemplo em pseudocódigo:

```
procedimento sub() {
  inteiro x = 1;
  procedimento sub1() {
     escreva(x);
  }
  procedimento sub2() {
     inteiro x = 3;
     sub1();
  }
  sub2();
  sub1();
}

// O que é escrito ao se chamar sub()?
```

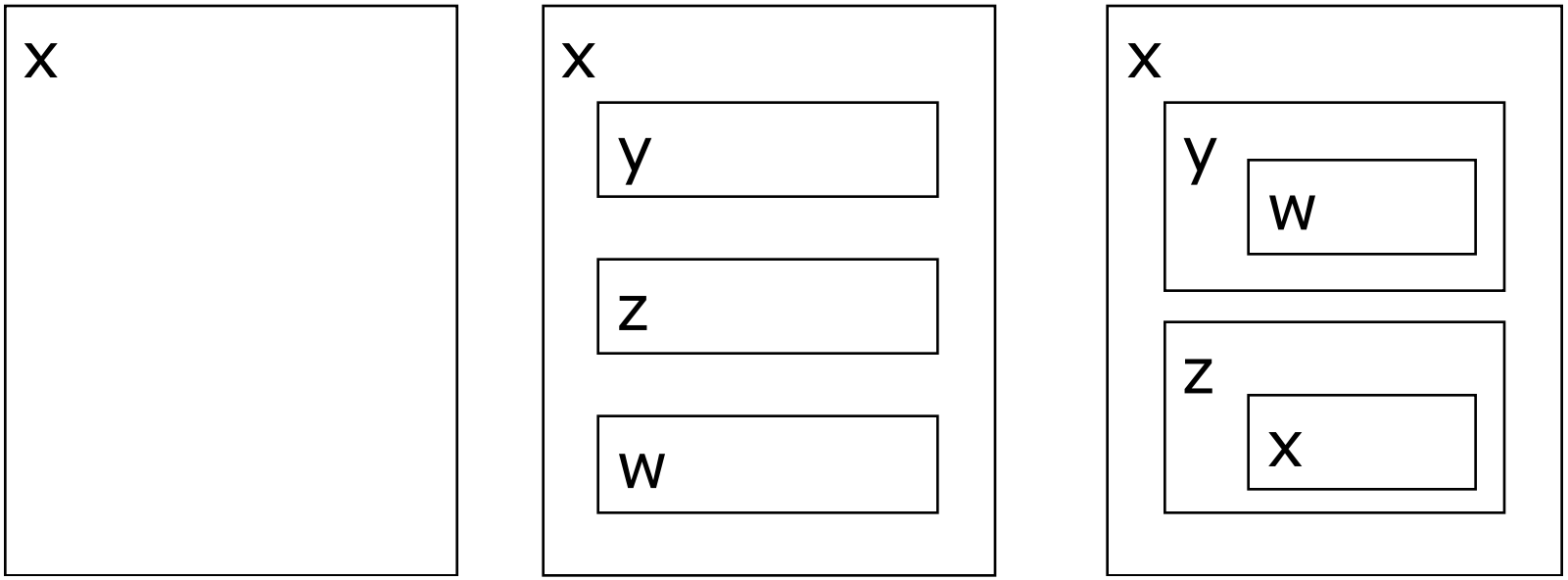
3
1

# Escopo dinâmico

- Problemas:
  - Eficiência: checagem de tipos durante execução, acesso deve seguir sequência de chamadas;
  - Legibilidade: deve-se seguir a sequência de chamadas para entender a amarração;
  - Confiabilidade: subprograma pode acessar variáveis locais do bloco que o chama;
  - Propenso a erros do programador;
- Pouquíssimo usado por LPs:
  - APL, Snobol4 e versões iniciais de Lisp e Perl;
  - Common Lisp e Perl suportam os dois tipos.
  - Facilmente substituída por passagem de parâmetros.

# Escopo estático

- Bloco monolítico: versões antigas de Basic e Cobol;
- Blocos não aninhados: Fortran;
- Blocos aninhados: Algol e linguagens Algol-like.




Bloco Monolítico | Blocos não aninhados | Blocos Aninhados

Escopos não aninhados são ditos "disjuntos" e não compartilham amarrações.

# Escopo estático

| | Algol | C | Java | Ada |
|---|---|---|---|---|
| **Pacote** | n/a | n/a | Sim | Sim |
| **Classe** | n/a | n/a | Aninhado | Sim |
| **Função** | Aninhado | Sim | Sim | Aninhado |
| **Bloco** | Aninhado | Aninhado | Aninhado | Aninhado |
| **Laço for** | Não | Não | Sim | Aninhado |

- Legenda:
  - n/a: conceito não está presente na linguagem;
  - Sim: conceito define escopo estático não-aninhado;
  - Aninhado: conceito define escopo estático aninhado.

# Escopo estático

- Ocultamento/visibilidade de Entidade em Blocos Aninhados – exemplo em C:

```c
// Obs.: alguns compiladores podem não compilar isso...
int main() {
   int i = 0, x = 10;
   while (i++ < 100) {
     float x = 3.231;
     printf("x = %f\n", x*i);
   }
}

/* O que o programa imprime na primeira e última
 * iteração do while? */
```

x = 3.231000
x = 323.100006

- Para evitar confusão, Java só permite ocultamento entre atributo e variável local.

# Escopo estático

- Tradutores usam tabelas de símbolos para manter o registro dos identificadores e suas amarrações:

```
01 int main() {
02    int i = 0, x = 10;
03    while (i++ < 100) {
04      float x = 3.231;
05      printf("x = %f\n", x*i);
06    }
07 }
```

<x, 4>
<i, 2> <x, 2>
<main, 1>

Pilha de dicionários

# Escopo estático

- Exemplo de referência seletiva em ADA:

```
procedure A is
  x : INTEGER;
  procedure B is
     y : INTEGER;
     procedure C is
        x : INTEGER;
     begin
        x := A.x;
     end C;
  begin
     null;
  end B;
begin
  null;
end A;
```

# Escopo estático

- Outro exemplo de referência seletiva em Java:

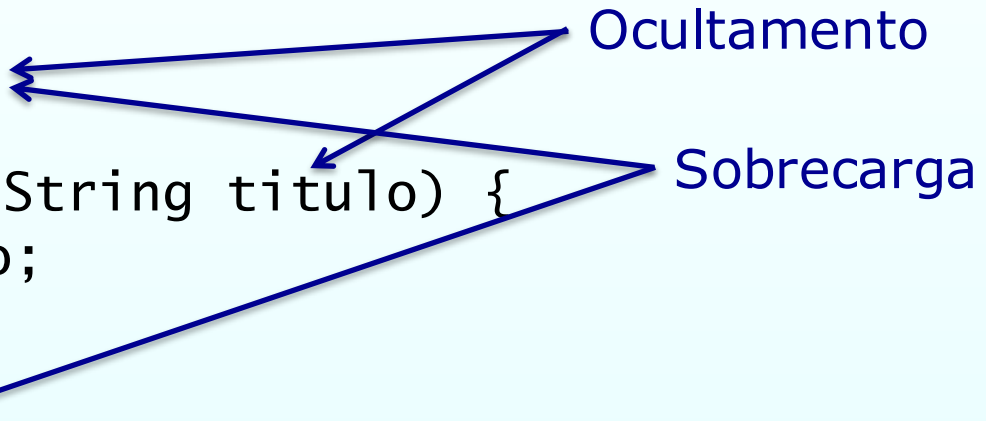
```
class A {
  public String nome = "a";
  public class B {
     public String nome = "b";
     public class C {
        public String nome = "c";
        public void imprime(String nome) {
           System.out.println(nome);
           System.out.println(this.nome);
           System.out.println(C.this.nome);
           System.out.println(B.this.nome);
           System.out.println(A.this.nome);
        }
     }
  }
}      // O que imprime o método imprime("d")?
```

d
c
c
b
a

# Escopo estático

- Ocultamento vs. sobrecarga em Java:

```
class Livro {
  private String titulo;

  public void setTitulo(String titulo) {
     this.titulo = titulo;
  }

  public String titulo() { return titulo; }
}

// O que é ocultamento e o que é sobrecarga?
```



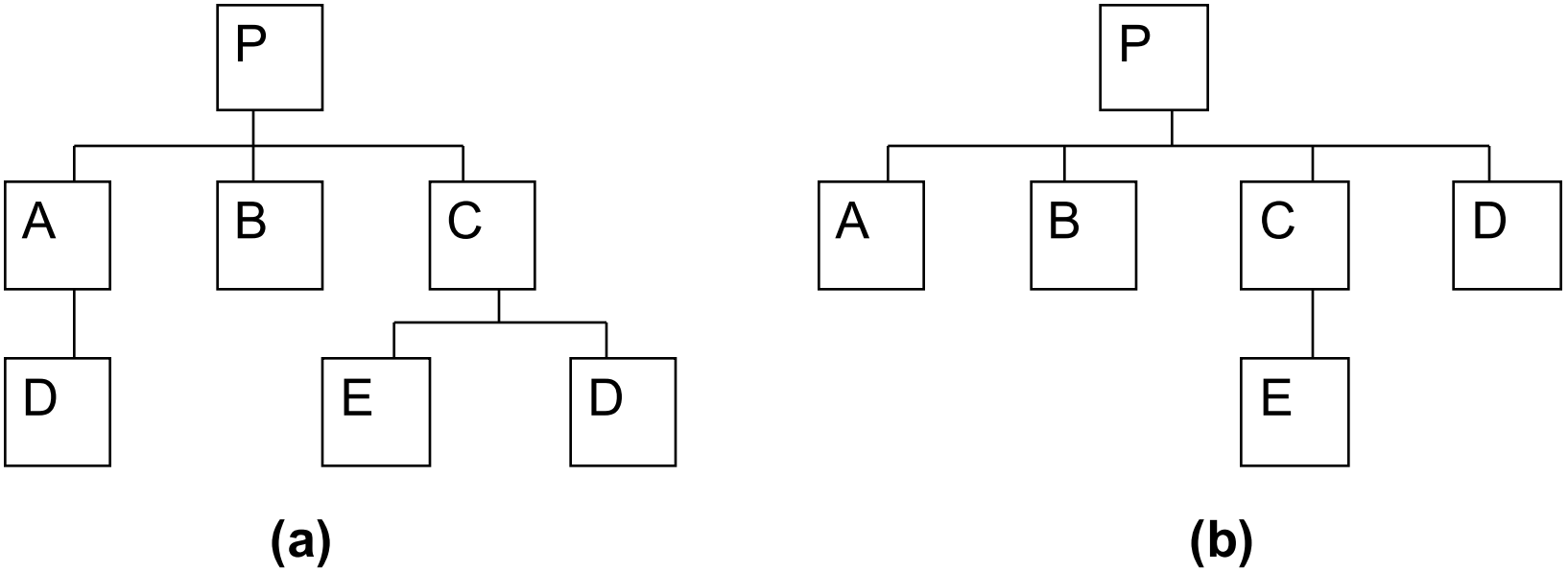

  - Retomaremos o assunto sobrecarga no cap. 5.

# Sobrecarga

- Capacidade de usar o mesmo nome em amarrações diferentes;
- Deve ser possível distinguir pelo contexto de uso do identificador:
  - Em Java, `System.out.print()` aceita qualquer tipo, distingue pelo parâmetro;
  - Em Modula não há sobrecarga nos writes: `WriteInt()`, `WriteReal()`, etc.;
- Ada e C++ permitem sobrecarga inclusive de operadores (+, -, *, etc.). Java só de métodos;
- Praticamente toda linguagem tem algum tipo de sobrecarga: + para inteiros e para reais.

# Escopo estático

- Problemas com a estrutura aninhada:




- D precisa ser usado dentro de A e C:
  - Em (a) há repetição;
  - Em (b), D fica visível para P e B também.

# Escopo estático

- Estruturas de blocos em C:

```
int x = 10;
int y = 15;

void f() {
  if (y - x) {
     int z = x + y;
  }
}

void g() {
  int w;
  w = x;
}
```

```
int main() {
  f();
  x = x + 3;
  g();
}
```

# Escopo vs. tempo de vida

- Em Algol-like, escopo = tempo de vida;
- Pode causar problemas – exemplos em C:

```
void conta() {
  int contador = 0;  // Deseja-se manter o valor.
  // ...
}
```

- Solução:

```
void conta() {
  static int contador = 0;  // Agora sim!
  // ...
}
```

Note: `static` se refere à forma de alocação da variável. Java herda o termo, que parece inadequado quando é associado a "escopo de classe e não de objeto". Porém essa é uma consequência da alocação estática!

# Escopo vs. tempo de vida

- Outro exemplo: *closures*.

```
// Exemplo em Groovy.
def obterClosure() {
  def valor = 10;
  return { println valor * 2 };
}

def closure = obterClosure();
closure();
```

Pense em *closures* como ponteiros para função. O que a chamada closure() imprime na tela?

20

# Definições e declarações (D&D)

- Definição:
  - Produz amarrações entre identificadores e entidades criadas na própria definição;
  - Ex.: `typedef struct TCoord { double x, y; } *Coord;`
- Declaração:
  - Produz amarrações entre identificadores e entidades já criadas ou que ainda o serão;
  - Ex.: `typedef struct TCoord *Coord;`

# Definições e declarações (D&D)

- Linguagens podem restringir momentos para definições e declarações.
  - Pascal tem um espaço reservado antes do bloco para variáveis;
  - Versões iniciais de C: permite dentro do bloco, mas como primeira instrução;
  - Java/C++: onde quiser, inclusive no for.

# Declaração de constantes

- Em C:

```
const float pi = 3.14;
#define pi 3.14
```

  - Possibilidade de alterar *consts* criou cultura de uso de #define, mas este não tem escopo nem tipo.

- Em Java:

```
// Dentro de uma classe qualquer:
final int const1 = 9;
static final int const2 = 39;
final int const3 = (int)(Math.random()*20);
static final const4 = (int)(Math.random()*20);
final int j;
Construtor () {
    j = 1;
}
```

# Exemplos de D&D: tipos

- Definições de tipos em C:

```
struct data {
  int d, m, a;
};
```

```
union angulo {
  int graus;
  float rad;
};
```

```
enum dia_util {
  seg, ter, qua,
  qui, sex
};
```

- Declarações de tipos em C:

```
struct data;
typedef union angulo curvatura;
typedef struct data aniversario;
```

# Exemplos de D&D: variáveis

- Definições de variáveis em C:

```
int k;
union angulo ang;
struct data d;
int *p, i, j, k, v[10];
```

- Definições com inicialização:

```
int i = 0;
char virgula = ',';
float f, g = 3.59;
int j, k, l = 0, m=23;
```

# Exemplos de D&D: variáveis

- Definições com inicialização dinâmica:

```
void f(int x) {
   int i;
   int j = 3;
   i = x + 2;
   int k = i * j * x;
}
```

- Definições com inicialização em variáveis compostas:

```
int v[3] = { 1, 2, 3 };
```

# Exemplos de D&D: variáveis

- Declaração de variáveis em C:

```
extern int a;
```

- Declaração de variáveis em C++:

```
int r = 10;
int &j = r;
j++;
```

# Exemplos de D&D: subprogramas

- Definições de subprogramas em C:

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

- Declaração de subprogramas em C:

```
int incr (int);

void f(void) {
   int k = incr(10);
}

int incr (int x) {
   x++;
   return x;
}
```

Um compilador que fizesse um pré-processamento poderia dispensar a declaração de incr. C não o faz, por isso a "*forward reference*". Em Java ela é desnecessária.

# Definições sequenciais

- Definições sequenciais em C:

```
struct funcionario {
  char nome [30];
  int matricula;
  float salario;
};

struct empresa {
  struct funcionario listafunc[1000];
  int numfunc;
  float faturamento;
};

void f() {
  int m = 3;
  int n = m;
}
```

# Definições sequenciais

- Definições sequenciais em ML:

```
val par = fn (n: int) => (n mod 2 = 0)
val negacao = fn (t: bool) => if t then false else true
val impar = negacao o par
val jogo = if x < y then par else impar
```

# Definições recursivas

- Definição recursiva de função em C:

```
float potencia (float x, int n) {
  if (n == 0) {
     return 1.0;
  } else if (n < 0) {
     return 1.0 / potencia(x, -n);
  } else {
     return x * potencia(x, n - 1);
  }
}
```

- Tipo recursivo em C:

```
struct lista {
   int elemento;
   struct lista * proxima;
};
```

# Definições recursivas

- Definições mutuamente recursivas em C:

```
void segunda (int);

void primeira (int n) {
   if (n < 0) return;
   segunda (n - 1);
}

void segunda (int n) {
   if (n < 0) return;
   primeira (n - 1);
}
```

# Definições recursivas

- Erro em definição de função strcmp em C:

```
int strcmp (char *p, char *q) {
  return !strcmp (p, q);
}
```

  - A ideia era chamar o `strcmp` original, mas acaba chamando ele mesmo recursivamente;
  - Java possui `super.metodo()` para explicitar o não-uso da recursividade;

- Explicitação de recursividade em função ML:

```
val rec mdc = fn (m: int, n: int) =>
  if m > n then mdc (m - n, n)
  else if m < n then mdc (m, n - m)
  else m
```

# Conclusões

- Foi apresentado o conceito de amarração, tempos de amarração, ambientes de amarração, escopo, etc.
- Falamos de definições e declarações de constantes, tipos, variáveis e subprogramas;
- Nas próximas aulas retomaremos de forma mais aprofundada:
  - Tipos de dados;
  - Variáveis;
  - Constantes
  - Subprogramas.


**http://nemo.inf.ufes.br/**