Linguagens de Programação

Filipe Rodrigues

frodrigues@iseg.ulisboa.pt

Gab. 302 Quelhas 2

Programa

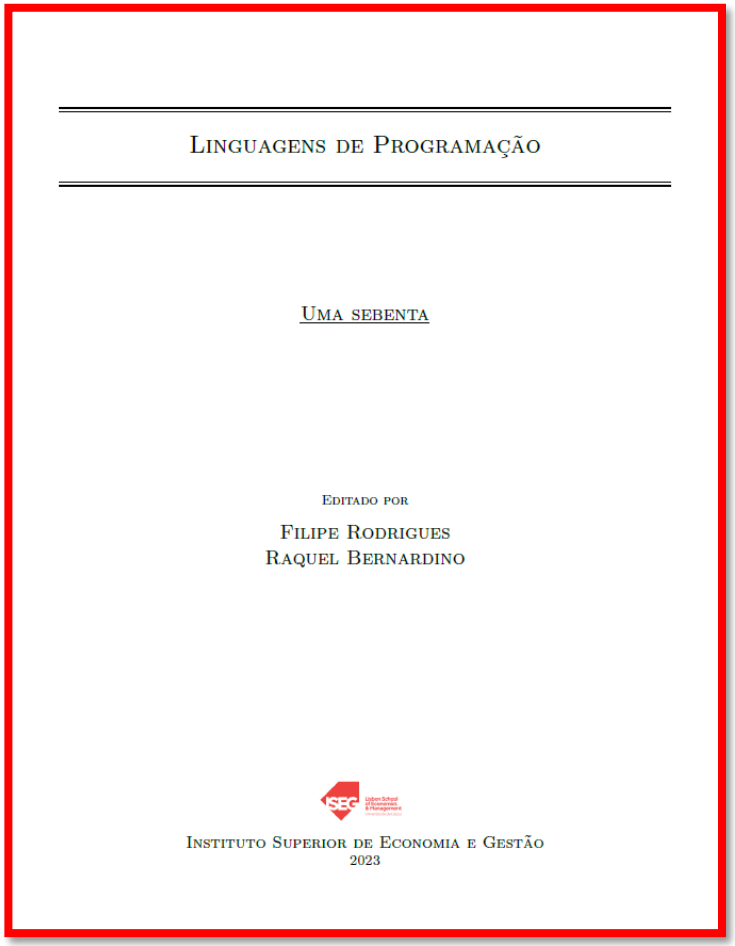
- Introdução à Programação
- Tratamento de Erros
- Leitura e Escrita de Ficheiros
- Classes e Sobrecarga de Operadores
- Herança e Polimorfismo

Bibliografia

- ✓ *Programming Principles and Practice Using C++, Bjarn Stroustrup, Second Edition, Addison-Wesley, 2014.*
- ✓ *A tour of C++, Bjarn Stroustrup, Addison-Wesley Professional, 2018.*
- ✓ https://www.learncpp.com/

Material de Apoio

- Slides
- Sebenta
- Lista de Exercícios
- Exercícios de consolidação



LINGUAGENS DE PROGRAMAÇÃO

UMA SEBENTA

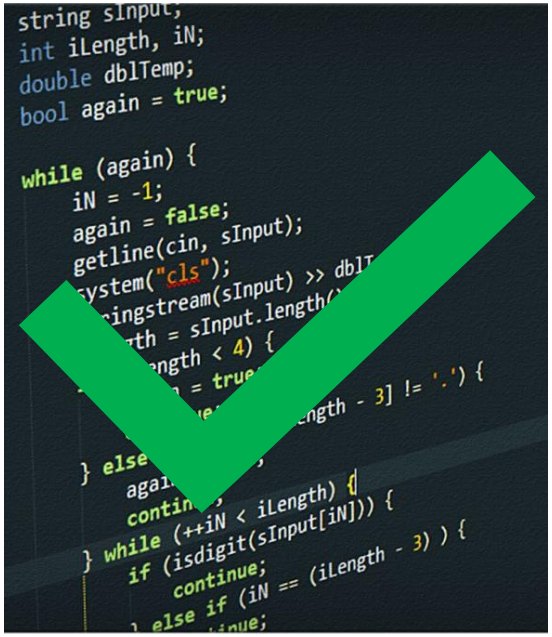
EDITADO POR

FILIPE RODRIGUES
RAQUEL BERNARDINO

INSTITUTO SUPERIOR DE ECONOMIA E GESTÃO
2023

O que é a Programação?

Programar é dizer a um computador de forma detalhada e precisa o que queremos que ele faça.



Linguagem e Software

Usaremos a linguagem de programação *C++*

e o software *QT Creator*

* Devem instalar o *QT Creator* em casa seguindo as instruções disponíveis no Fénix

Esqueleto de um programa em C++

```
//Preâmbulo

int main(){

    //Escreva aqui o seu código

    return 0;
}
```

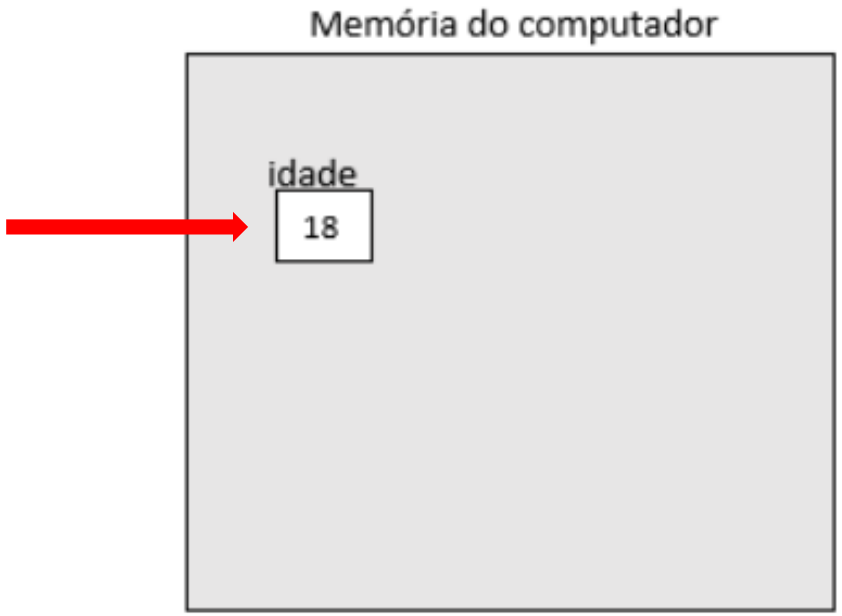
Capítulo 1 –
Variáveis e Operadores

Variáveis

- São as unidades básicas de qualquer programa uma vez que é nelas que é armazenada a informação.

- Qualquer variável tem um nome e é uma localização da memória do computador onde a informação pode ser guardada de modo a ser usada por um programa.

Variável com o nome "*idade*" que guarda um valor inteiro



Nomes das Variáveis

- Só pode começar com uma letra ou com *underscore;*
- Não pode conter espaços nem caracteres que não sejam números, letras ou *underscores*;
- Deve ser único na área em que é usado;
- Não pode coincidir com palavras reservadas: *if, else, for, while, try, main, int, etc…*

Tipos de Variáveis

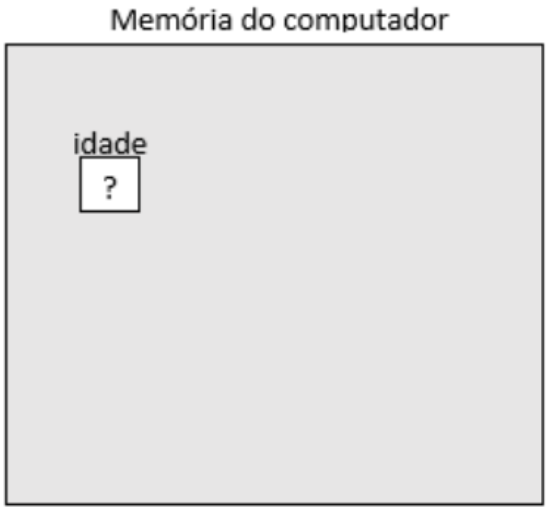
	Tipos
Números inteiros	✗ *short* *int* ✗ *long* *long long int*
Números inteiros positivos	*size_t*
Números decimais	✗ *float* *double*
Carater	*char*
Texto	*string*
Valor lógico	*bool*

Com exceção do tipo de dados *string,* todos os outros são considerados tipos de **dados primitivos**.

Declaração e Inicialização de Variáveis

- **Declaração**

```
Tipo nome;
```

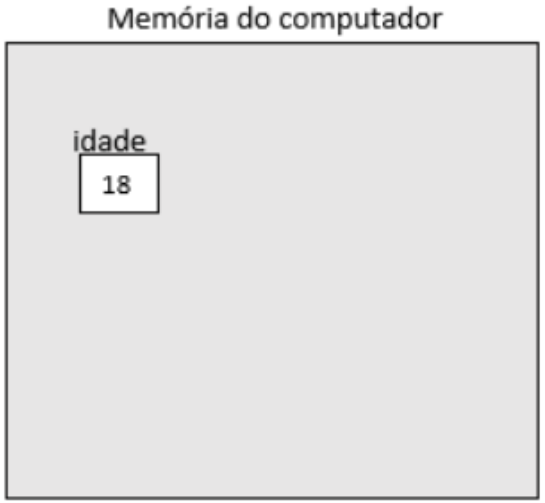


- **Declaração e Inicialização**

```
Tipo nome;
nome = valor;
```

ou

```
Tipo nome = valor;
```



Declaração e Inicialização de Variáveis

```
#include <iostream>
using namespace std;

int main(){

    int idade = 18;
    double peso = 56.8;
    string nomeP = "Pedro";
    char c = ')';
    bool logico = true;
    double altura;

    return 0;
}
```

É necessário aqui para podermos usar o tipo de dados *string*

Variáveis declaradas e inicializadas

Variável declarada e não inicializada

Alterar Valor das Variáveis

- Uma variável de um tipo primitivo ou *string* guarda apenas um valor, sendo que esse valor pode ser alterado ao longo da execução do programa.

```
int main(){
    int idade = 10;
    idade = 20;
    //...
    idade = -3;
    return 0;
}
```

Declaração e inicialização da variável *idade* com o valor 10

Alteração do valor da variável *idade* para 20

Alteração do valor da variável *idade* para -3. É possível?

Escrita de Variáveis

- Para escrever no ecrã o valor guardado numa variável (ou qualquer outra informação) usamos a instrução ***cout*** e o operador <<.

```
cout << nome_da_variável;

cout << "qualquer texto";
```

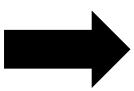
- A utilização destes elementos requer a inclusão da seguinte instrução no preâmbulo:

```
using namespace std;
```

Escrita de Variáveis - Exemplo

```
int idade = 15;
string nomeP = "Pedro";
```

```
cout << "Output: \n";
cout << nomeP;
cout << endl;
cout << idade;
cout << "\n";
```



```
Output:
Pedro
15
```

Mudança de linha

ou

```
cout << "Output: \n" << nomeP << endl << idade << "\n";
```

Leitura de Variáveis

- Para ler uma variável, isto é, pedir o seu valor ao utilizador usamos a instrução ***cin*** e o operador >>.

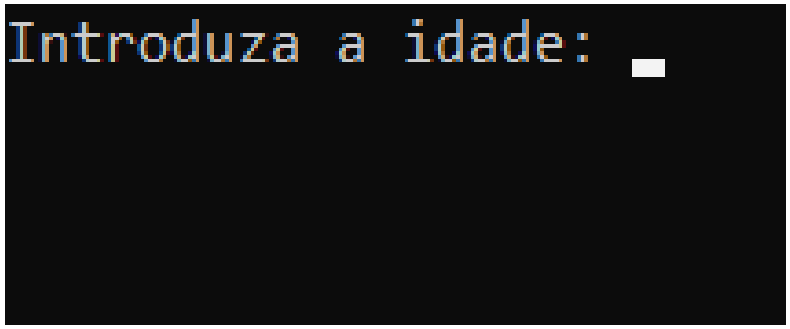
```
cout << "Introduza um valor: ";
cin >> nome_da_variável;
```

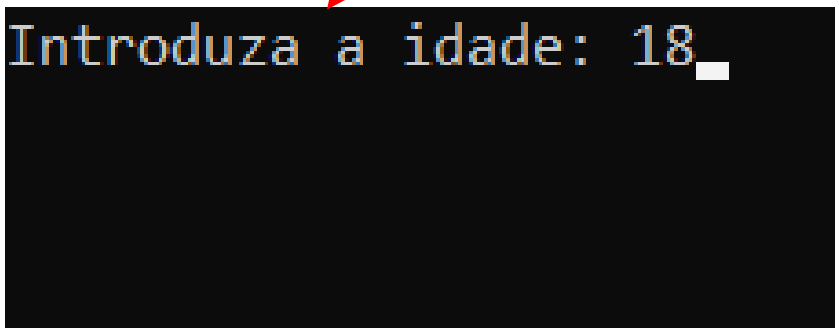
- A utilização destes elementos requer a inclusão da seguinte instrução no preâmbulo:

```
using namespace std;
```

Leitura de Variáveis - Exemplo

```
int idade;
cout << "Introduza a idade: ";
cin >> idade;
```





Constantes

- São variáveis cujo valor inicial não pode ser alterado.
- A sua declaração e inicialização é feita da seguinte forma:

```
const Tipo nome = valor;
```

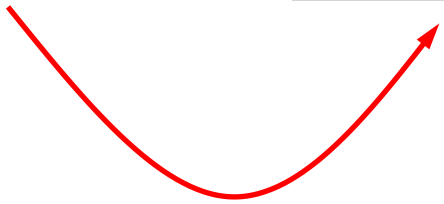
Exemplo:

```
int main(){
    const double pi = 3.141592;
    pi = 3.14;    //ERRO!
    return 0;
}
```

Operadores

- Operadores aritméticos simples

Operador	Nome	Exemplo	Resultado
+	Soma	$a+b$	18
-	Subtração	$a-b$	8
*	Multiplicação	$a*b$	65
/	Divisão inteira ou decimal	a/b	2 ou 2.6
%	Resto da divisão inteira	$a\%b$	3



Se a=13 e b=5

Operadores

Os operador + pode também ser usado para variáveis do tipo *string* como *operador de concatenação (*junção)

```
string a = "Eu tenho ";
int c = 18;
string b = " anos.";

string frase = a + to_string(c) + b + " Sou Jovem!";
```

Converte o valor numérico 18 para a *string* "18"

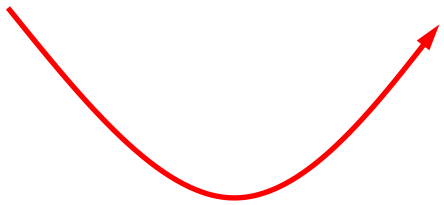
A nova *string* criada é:

"Eu tenho 18 anos. Sou Jovem!"

Operadores

- Operadores aritméticos compostos

Operador	Nome	Exemplo	Significado	Valor de a
+=	Soma/atribuição	a+=b	a=a+b	8
-=	Subtração/atribuição	a-=b	a=a-b	4
*=	Multiplicação/atribuição	a*=b	a=a*b	12
/=	Divisão/atribuição	a/=b	a=a/b	3
++	Incremento	a++	a=a+1	7
--	Decremento	a--	a=a-1	5



Se a=6 e b=2

Operadores

O operador += pode também ser usado para variáveis do tipo *string* como *operador de concatenação + atribuição.*

```
string a = "AA";
string b = "BB";
a += b
```

Neste caso teríamos:

a = "AABB"

b = "BB"

Operadores

- Operadores relacionais e operadores lógicos

Operador	Significado
<	menor
>	maior
<=	menor ou igual
>=	maior ou igual
==	igual
!=	diferente

Operador	Significado
&& ou *and*	conjunção (e)
|| ou *or*	disjunção (ou)
!	negação

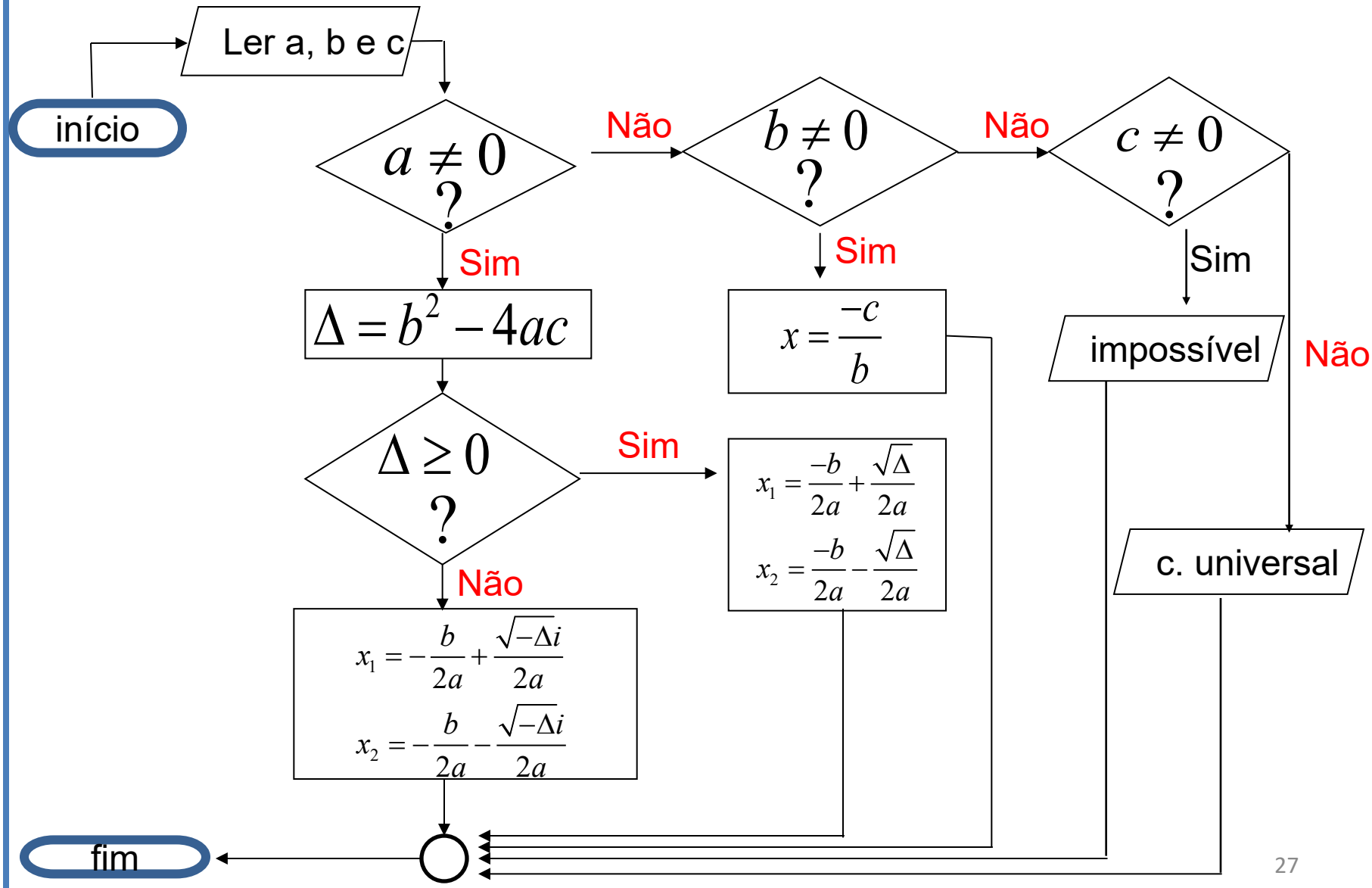
Capítulo 2 – Estruturas de Controlo Condicionais e Cíclicas

Estruturas Condicionais

- Permitem que o programa siga caminhos distintos em função da verificação ou não de determinadas condições lógicas.

- Estão inevitavelmente dependentes dos operadores relacionais e lógicos.

Exemplo: equação de 2ºgrau $ax^2 + bx + c = 0$



If e If…Else

```
if ( condição ) {
    Instruções
}
```

```
if ( condição ) {
        Instruções_1
}else{
        Instruções_2
}
```

Encadeadas tipo 1

```
if ( condição1 ) {
      instruções_1
}else{
      if ( condição_2 ) {
          instruções_2
      }else{
          if ( condição_3 ) {
                instruções_3
          }else{
                instruções_4
          }
      }
}
```

```
int qt, preco;
cout << "Quantidade: ";
cin >> qt;

if ( qt < 50 ) {
    preco = 5 * qt;
}else{
     if ( qt < 99 ) {
        preco = 4 * qt;
     }else{
          if ( qt < 150 ) {
             preco = 3.5 * qt;
          }else{
             preco = 3.3 * qt;
          }
     }
}
```

Encadeadas tipo 2

```
if ( condição_1 ) {
      instruções_1
}else if ( condição_2 ) {
      instruções_2
}else if ( condição_3 ) {
      instruções_3
}else{
      instruções_4
}
```

```
int qt, preco;
cout << "Quantidade: ";
cin >> qt;

if ( qt < 50 ) {
    preco = 5 * qt;
}else if( qt < 99 ) {
    preco = 4 * qt;
}else if ( qt < 150 )
    preco = 3.5 * qt;
}else{
    preco = 3.3 * qt;
}
```

Estruturas Cíclicas

- Permitem executar instruções/processos de forma repetitiva.

- Contêm uma condição de paragem, que determina o final do processo iterativo.

Exemplo: Algoritmo de “Euclides”

Dados os números inteiros m=38 e n=10, calcular o seu máximo divisor comum mdc(38,10).

Como 38=3*10+8

os divisores comuns a 38 e 10 são os comuns a 10 e 8;

Como 10=1*8+2

os divisores comuns a 10 e 8 são os comuns a 8 e 2;

Como 8=4*2+0

o máximo divisor comum entre 8 e 2 é 2 !

=> mdc(38,10)=2

Algoritmo de “Euclides” (versão I)

1. **Ler** **m** e **n** (inteiros diferentes de 0);
2. **Se** **m<n** **Então**

   maior=**n**, menor=**m**

   Senão

   maior=**m**, menor=**n**
3. Calcula resto=mod(maior , menor);
4. **Se** resto==0

   Escrever menor; **FIM**
5. **Enquanto** (resto!=0)

   resto=mod(maior,menor), maior=menor, menor=resto
6. **Escrever** maior; **FIM**

Algoritmo de “Euclides” (versão II)

1. **Ler** **m** e **n** (inteiros diferentes de 0);
2. **Enquanto** (**n**!=0)

   resto=mod(**m,n**), **m**=**n**, **n**=resto;
3. **Escrever** **m**; **FIM**

While e Do...While

do{

instruções

}while (condição);

while (condição) **{**

instruções

}

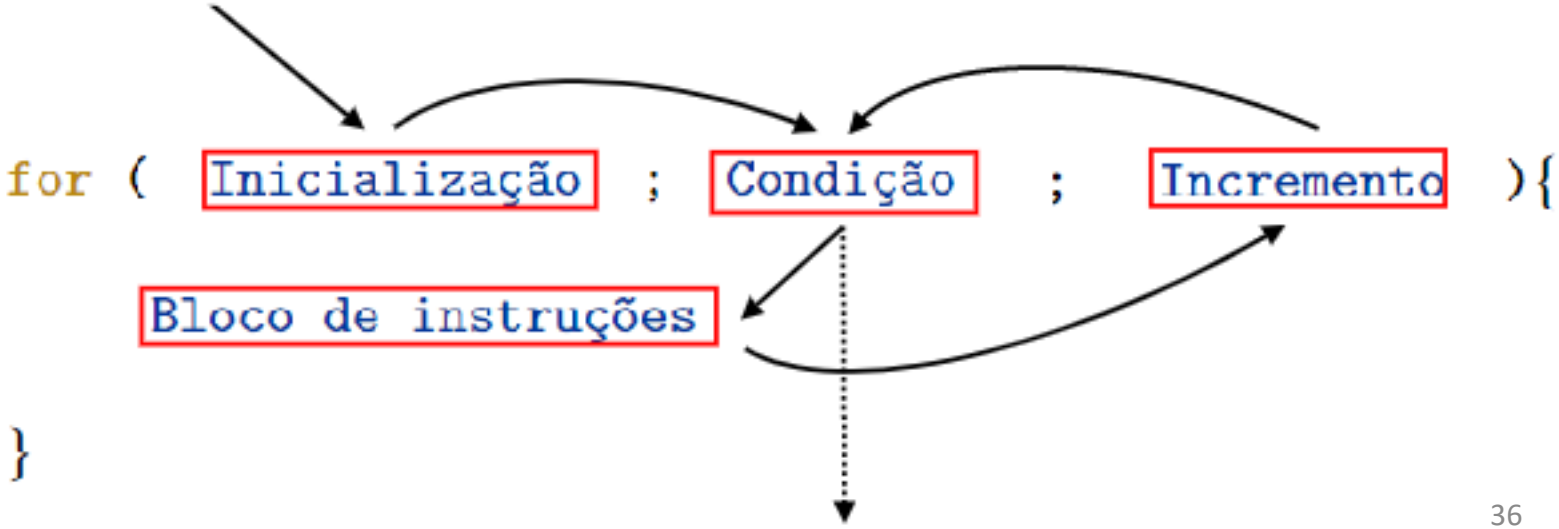
```
int n;
int conta = 0;

do{
    cout << "Valor: ";
    cin >> n;
    if ( n > 0)
        conta++;
}while( n > 0 );
```

```
int n = 100;
int conta = 0;

 while( n > 0 ){
    cout << "Valor: ";
    cin >> n;
    if ( n > 0)
        conta++;
 }
```

Ciclo For

```
for ( Inicialização ; Condição ; Incremento ) {
        Bloco de Instruções
}
```

Sequenciamento de instruções:



Ciclo while *vs* Ciclo for

Ciclo while

```
int n = 4;

int i = 0;
while( i < n ){
    cout << i << " ";
    ++i;
}
```

Ciclo for

```
int n = 4;

for(int i = 0; i < n; ++i){
    cout << i << " ";
}
```

Ciclos Encadeados

- Ciclos encadeados são ciclos que contêm outros ciclos dentro deles.

- O ciclo (ou ciclos) interior são executados em cada iteração do ciclo exterior.

Exemplo:

```
for (Inicialização_1 ; Condição_1  ; Incremento_1){
        for (Inicialização_2 ; Condição_2  ; Incremento_2){
                Bloco de Instruções
        }
}
```

Ciclos Encadeados - Exemplo

```
int n = 5;

for(int i = 1; i < n; i++) {
    for(int j = i + 1; j < n; j++) {
        cout << "(" << i << "," << j << ")   ";
    }
}
```

(1,2) (1,3) (1,4) (2,3) (2,4) (3,4)

Instrução break

A instruções ***break*** permitem interromper um ciclo. No caso de ciclos encadeados, apenas um dos ciclos é interrompido.

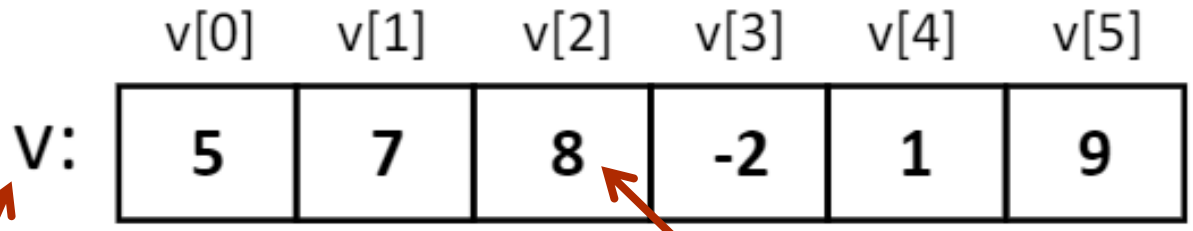
```
int n = 5;

for(int i = 1; i < n; i++) {
   for(int j = i + 1; j < n; j++) {
       cout<<"("<<i<<","<<j<<") ";
           if ( j % i == 0 )
               break;
    }
}
```

(1,2) (2,3) (2,4) (3,4)

Capítulo 3 – Variáveis Indexadas - Vetores

Vetores

- Permitem armazenar numa só variável vários valores, desde que sejam todos do mesmo tipo.



Vetor de inteiros *v* com 6 posições (desde 0 até 5)

Elemento que está na posição 2 do vetor. (v[2] = 8)

* Necessitamos do pacote *vetor*: `#include<vector>`

Vetores - declaração

- **Opção 1: Declaração e Inicialização a partir de uma lista.**

```
vector<Tipo> nome = { … };
```

Este processo é usado quando conhecemos os elementos do vetor no momento da sua criação

Exemplo:

```
vector<int> v = {5, 7, 8, -2, 1, 9};
```

Vetores - declaração

- **Opção 2: Declaração de vetor com dimensão**

```
vector<Tipo> nome(n);
```

Este processo é usado quando apenas conhecemos a dimensão do vetor no momento da sua criação

Exemplo:

```
vector<int> v(6);
v[0]=5; //ou v.at(0)=5;
   ...
v[5]=9;
```

Preenche cada uma das posições já criadas

Vetores - declaração

- **Opção 3: Declaração de vetor sem dimensão**

```
vector<Tipo> nome;
```

Quando nem a dimensão do vetor nem os seus elementos são conhecidos no momento da criação.

Exemplo:

```
vector<int> v;
v.resize(6);
v[0]=5;
   ...
```

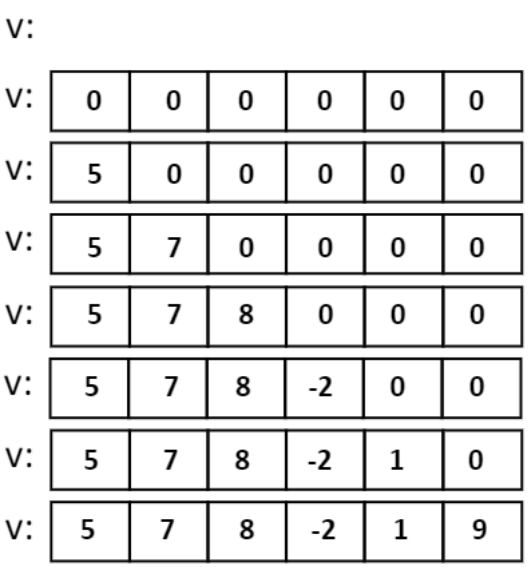
Redimensiona o vetor para criar de uma só vez todas as posições

```
vector<int> v;

v.push_back(5);
   ...
```

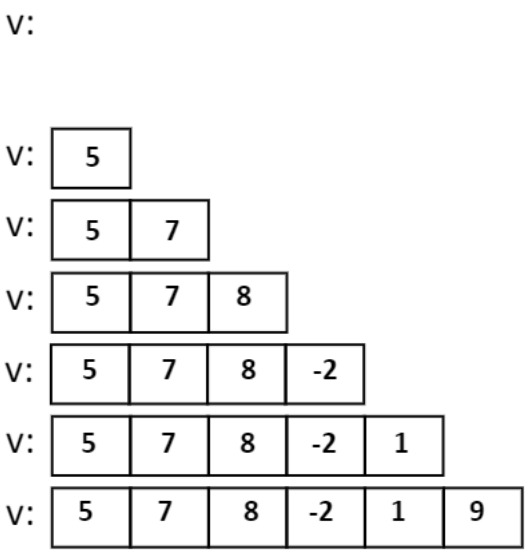
Cria uma nova posição no vetor e preenche-a com o valor 5

Resize *vs* Push_back

```
vector<int> v;
v.resize(6);
v[0]=5;
v[1]=7;
   ...
```




```
vector<int> v;

v.push_back(5);
v.push_back(7);
   ...
```



Vetores - declaração

- O operador [ ] e o método *.at()* apenas podem ser usados para aceder a posições do vetor que já existam.

- O método *.at()* verifica se a posição do vetor existe antes de tentar aceder a ela, enquanto que o operador [ ] não o faz.

- O método *.push_back()* adiciona sempre uma nova posição ao vetor e preenche-a com o valor que recebe.

Vetores - Exemplo

Uma empresa tem 100 produtos, cada um com o seu preço e pretende atualizar o IVA que era de 6% e passou a ser de 23%.

100 variáveis

Declaração

```
double preco0;
double preco1;
       …
double preco99;
```

Atualização

```
preco0=0.23/0.06*preco0
preco1=0.23/0.06*preco1
              …
preco99=0.23/0.06*preco99
```

1 variável indexada com 100 elementos

Declaração

```
vector<double> preco(100);
```

Atualização

```
for(int i=0; i<=99; ++i)
  preco[i]=0.23/0.06*preco[i];
```

Vetores - Manipulação

- A manipulação de vetores é feita posição a posição. Por exemplo, para imprimir/preencher um vetor é necessário imprimir/preencher cada uma das suas entradas.

- A primeira posição do vetor é a posição 0. A dimensão do vetor é dada pelo método *.size()* e por isso a última posição do vetor será *.size() – 1.*

- Usamos, normalmente, ciclos *for* para percorrer vetores

```
for(int i = 0; i < v.size(); i++)
```

Vetores – Leitura

- Se soubermos quantos elementos terá o vetor, podemos usar um ciclo *for.*

```
vector<int> v(3);

for(int i = 0; i < v.size(); ++i){
   cin >> v[i]; //ou cin>>v.at(i);
}
```

- Caso contrário, devemos usar um ciclo *while*

```
vector<double> v;

double x;
while(cin >> x){ //Enquanto forem lidos valores numéricos
   v.push_back(x);
}
```

Ordenar vetor

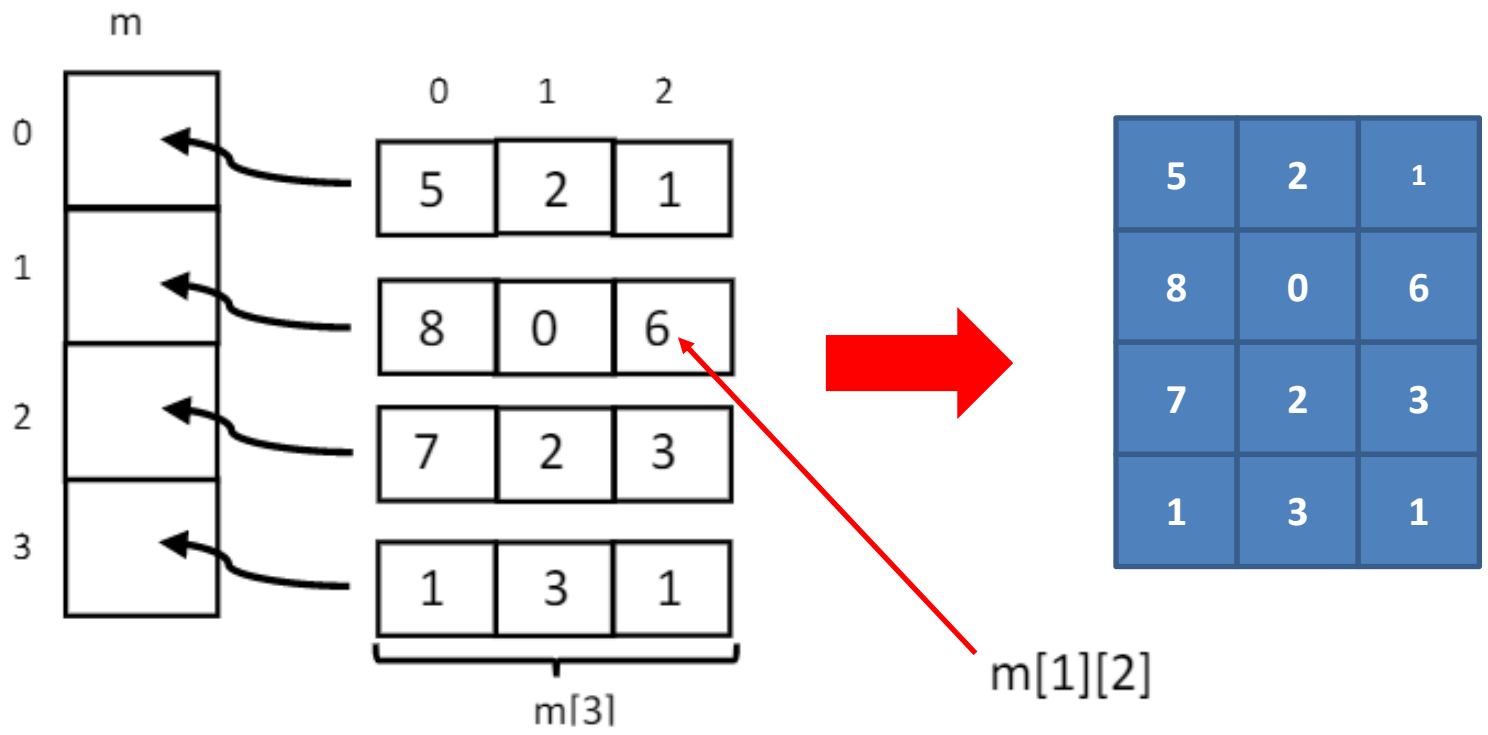
Existem vários algoritmos que podemos implementar para ordenar um vetor. O método *sort, p*ermite ordenar um vetor automaticamente.

```
vector<int> v = {5, 7, 8, -2, 1, 7};

//Ordenar vetor v por ordem crescente
   sort( v.begin(), v.end() );

//Ordenar vetor v por ordem decrescente
   sort( v.begin(), v.end(), greater <>() );
```

* Necessitamos do pacote *algorithm*

Matrizes

Uma matriz em C++ é um vetor de vetores.




```
vector<vector<int>> m;
```

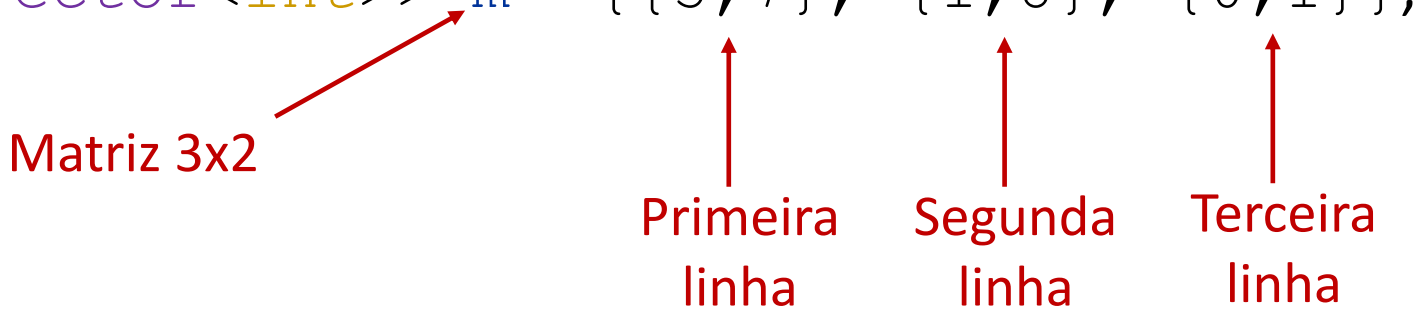
Matriz – declaração e definição

- **Opção 1: Declaração e Inicialização a partir de uma lista.**

```
vector<vector<Tipo>> nome = {{…},…,{…}};
```

Exemplo:

```
vector<vector<int>> m = {{5,7}, {1,8}, {0,1}};
```



Matriz – declaração e definição

- **Opção 2: Definir número de linhas, número de colunas e os elementos**

Definir número de linhas (n)

```
vector<vector<Tipo>> nome(n);
        ou
vector<vector<Tipo>> nome;
nome.resize(n)
```

Definir número de colunas (m)

```
for(int i=0; i<n; i++)
    nome[i].resize(m);
```

Preencher Matriz

```
m[0][0]= 5
    …
m[n-1][m-1]= 7
```

Capítulo 4 – Funções

Declaração, Definição e Chamada

$$f: \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{R}$$

```
//declaração
    double f(int, int);
```

$$f: \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{R}$$

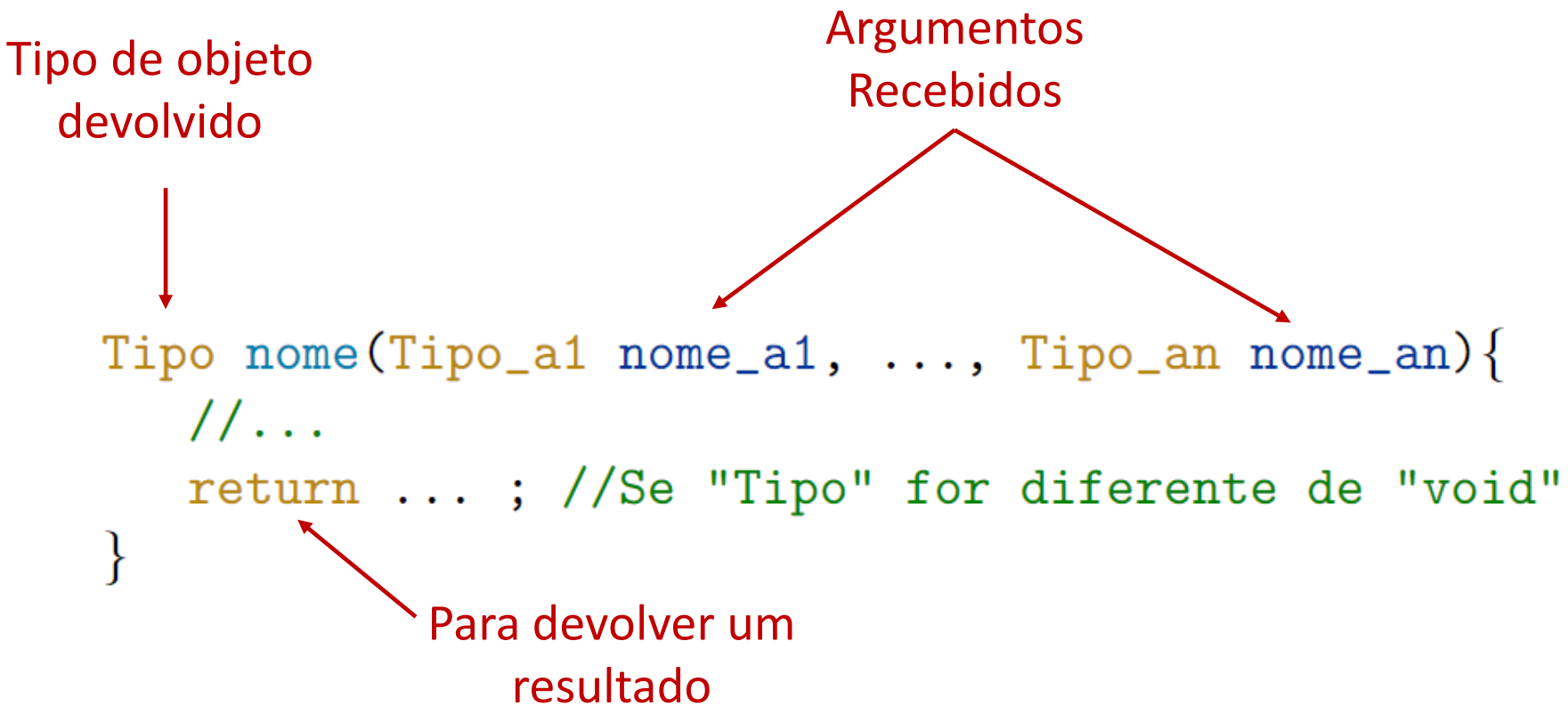
$$(x, y) \longrightarrow f(x, y) = \frac{x}{y^2 + 1}$$

```
//declaração e definição
    double f(int x, int y){
      return x / (y * y + 1);
    }
```

$f(2, 3),\ f(5^2, -8)\ \ldots$

```
//chamadas
    cout << f(2,3);              //chamada 1

    double z1 = 5 * f(1,7);      //chamada 2

    int a = 6, b = 1;
    double z2 = f(a,b);          //chamada 3
```

Estrutura geral de uma função



Uma função do tipo *void* (vazia) não devolve qualquer resultado, pelo que não contém a instrução *return.*

Função void *vs* não void

```
void ordem(int n1, int n2){
    if( n1 < n2 )
       cout << n1 << " <= " << n2;
    else
       cout << n2 << " <= " << n1;
}
```

Apenas escreve algo no ecrã, não devolve qualquer resultado

```
ordem(7,5)
```

```
int maximo(int n1, int n2){
    int max = n1;
    if( max < n2 )
        max = n2;
    return max;
}
```

Devolve um resultado do tipo int que pode ser usado no programa

```
int y = maximo(a,b) - 6;
```

Funções Recursivas

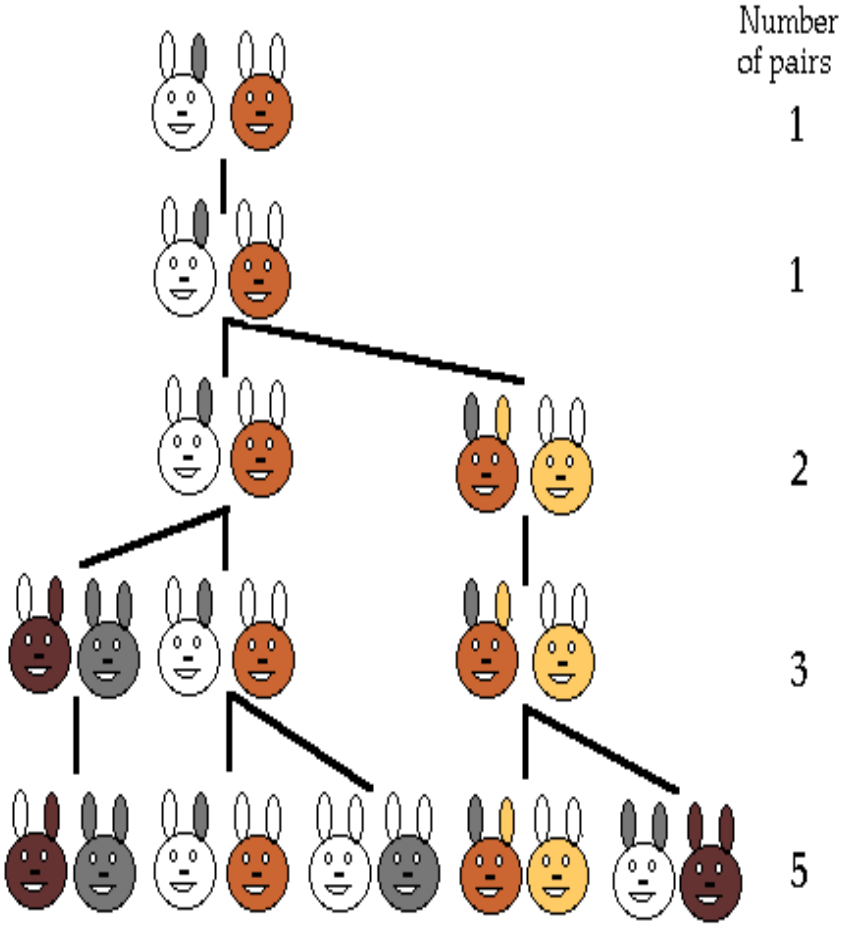
Uma função recursiva é uma função que se chama a si própria. Neste tipo de função, é extremamente importante definir cuidadosamente o critério de paragem.

Exemplo:

fatorial(6)
6 * fatorial(5)
6 * 5 * fatorial(4)
6 * 5 * 4 * fatorial(3)
6 * 5 * 4 * 3 * fatorial(2)
6 * 5 * 4 * 3 * 2 * fatorial(1)
6 * 5 * 4 * 3 * 2 * 1

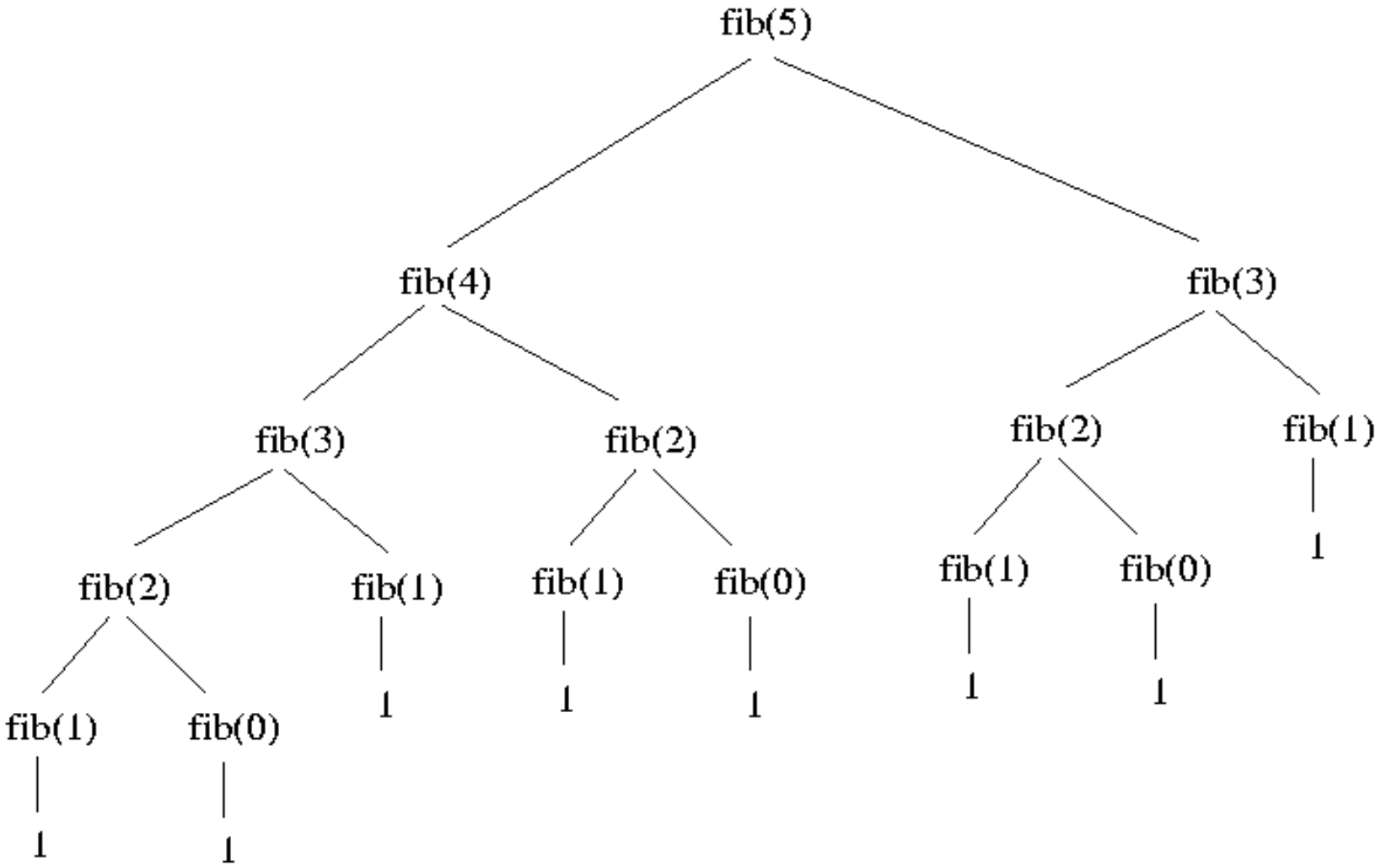
```
int fatorial(int n){
   if ( n == 1 )
      return 1;
   else
      return n*fatorial(n-1);
}
```

Exemplo: Fibbonacci



fib(0) = 1
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1

Exemplo: Fibbonacci



Passagem de argumentos

Passagem *por valor*. O que é passado como argumento à função é uma cópia do valor da variável e não a própria variável.

```
Tipo nome(Tipo_argumento argumento){...}
```

Passagem *por referência*. O que é passado como argumento à função é o local da memória onde se encontra a variável.

```
Tipo nome(Tipo_argumento& argumento){...}
```

Passagem *por referência constante*. Difere do anterior na medida em que a função não pode alterar a variável.

```
Tipo nome(const Tipo_argumento& argumento){...}
```

Passagem de argumentos: Exemplo

```
void f(int a, int& b, const int& c){
    a += 10 + c;
    b += 10 + c;
    //c += 10;     ERRO!
    cout << a << " " << b << " " << c;  //a=12, b=12, c=1
}
```

```
int main(){
    int x = 1;
    int y = 1;
    int z = 1;
    f(x, y, z);
    cout << x << " " << y << " " << z;  //x=1, y=12, z=1

    return 0;
}
```

Não é alterado na função

É alterado

Passagem de argumentos

Não é regra, mas é uma boa prática...