

# **Algoritmos e Estruturas de Dados**

**Desenvolvimento Web e Multimédia  
Redes e Segurança Informática**

**Hugo Freitas - 2022**

# Apresentação

# Apresentação

## Docente

- Hugo Freitas
- [hfreitas@ipca.pt](mailto:hfreitas@ipca.pt)
- Licenciado em Ciências da Computação na Universidade do Minho
- 8 anos de experiência como Software Engineer
- Atualmente, Software Engineer Team Lead na empresa Checkmarx
- Docente no IPCA desde 2019

# **Apresentação**

## **Horário de atendimento**

- Sábados das 9h às 11h com marcação prévia

## **Repositório de material pedagógico**

- <https://github.com/hffreitas/Algoritmos-Estruturas-Dados>



# Apresentação

## Propósito

- Apresentar os conceitos fundamentais relativos a:
  - Algoritmia
  - Estruturas de dados
  - Programação estruturada
- Desenvolver a capacidade de compreender e analisar problemas
- Conceber e planear soluções estruturadas conducentes à sua resolução
- Utilizar a linguagem de programação C para implementar as soluções

# Apresentação

## Objetivos

- No final, os alunos deverão ser capazes de:
  - Analisar problemas
  - Propor uma solução na linguagem de programação C suportada por algoritmos representados em fluxogramas e pseudocódigo
  - Perceber o processo de codificação, compilação e execução
  - Utilizar estruturas condicionais e cíclicas, arrays, strings e estruturas

# Apresentação

## Metodologia de ensino

- Cerca de 90% de componente prática
  - Resolução de exercícios nas aulas
- Cerca de 10% dedicados à revisão de algoritmos e programação

# **Apresentação**

## **Ferramentas**

- TBD

# Apresentação

## Programa

- Algoritmos e resolução de problemas
  - Resolução de problemas
  - Aproximação descendente(*top-down approach*)
  - Noção formal de algoritmo
  - Características de um algoritmo

# Apresentação

## Programa

- Estruturas de Dados
  - Estruturas de dados primitivas
    - Boolean's
    - Numéricos
    - Alfanuméricos
    - Representação dos dados
  - Estruturas de dados não primitivas
    - Arrays
    - Matrizes

# Apresentação

## Programa

- Notação algorítmica
  - Pseudocódigo
    - Instrução de atribuição
    - Leitura e escrita de dados
    - Estrutura condicional
    - Instruções de repetição
    - Operações e expressões aritméticas
    - Operadores e operações relacionadas
    - Operadores e operações lógicas
  - Algoritmos propostos
  - Diagramas de fluxo

# Apresentação

## Programa

- Introdução à linguagem C
  - Primeiros programas
  - Instrução de atribuição
  - Leitura e escrita de dados
  - Estrutura condicional
  - Instruções de repetição
  - Operações e expressões aritméticas
  - Operadores e operações relacionadas
  - Operadores e operações lógicas
  - Modularização (Funções, procedimentos)
  - Tipos de dados complexos
  - Gestão dinâmica de memória

# Apresentação

## Avaliação

- Os resultados da aprendizagem serão avaliados através dum componente teórica (prova escrita) e de uma componente prática (trabalho de grupo)
- A nota final é a média ponderada segundo a expressão seguinte:
  - $NF = PE \times 50\% + TG \times 50\%$
  - NF - nota final, PE - prova escrita, TG - trabalho de grupo, PA - participação nas aulas
- O TG é composto por, código em C, relatório e defesa presencial

# Apresentação

## Avaliação

- Aproveitamento à UC está sujeito à obtenção de **nota mínima de 10** valores à componente prática e de 10 valores à componente teórica
- **Não serão aceites entregas** ou melhorias do trabalho prático em época de exames
- Em época de exame apenas será avaliada a componente teórica, mantendo-se, para efeitos do cálculo da nota final, o valor obtido na componente prática durante a frequência da UC
- A não obtenção da nota mínima de 10 valores na componente prática **impossibilita** a realização da prova escrita (teste ou exame)
- **PLÁGIO SERÁ SEVERAMENTE PUNIDO.**

# Apresentação

## Datas

- Teste de avaliação
  - TBD
- Entrega e Defesa do TP
  - TBD
- Exame
  - TBD

# Apresentação

## Bibliografia

### ■ Principal

- Pereira, Alexandre (2013), “C e Algoritmos”, 1a edição, edições sítabo
- Damas, Luís (1999), “Linguagem C”, 20.a edição, FCA – Editora de Informática Lda., série Tecnologias de Informação.
- Guerreiro, P. (2001), “Elementos de Programação com C”, 3.a edição, FCA – Editora de Informática Lda., série Tecnologias de Informação.
- Vasconcelos, J.B., Carvalho, J.V. (2005), “Algoritmia e Estruturas de Dados”, Centro Atlântico.

### ■ Complementar

- Loudon, Kyle (1999), “Mastering Algorithms in C”, O'Reilly.
- Kernighan and Ritchie (1988), “The C Programming Language (ANSI C)”, 2.nd edition, Prentice Hall3.

# Algoritmos e resolução de problemas

# Algoritmos e resolução de problemas

## O que é um programa?

- Um programa de computador
  - Algoritmo
  - Objetivo é resolver um problema
- Um algoritmo é representado por:
  - Expressões simbólicas
  - Permitem descrever e encontrar a solução
- Um algoritmo representa:
  - Sequência finita de instruções
    - Conduzem à resolução do problema
    - Cada uma pode ser executada mecanicamente numa quantidade finita de tempo

# **Algoritmos e resolução de problemas**

## **O que são estruturas de dados?**

- Representam entidades e objetos do mundo real
- Definem a parte estática dum algoritmo
- A manipulação das estruturas de dados definem a parte dinâmica dum algoritmo
- O conjunto constitui formalmente um algoritmo

# Resolução de problemas

## O que é?

- Processo de identificar e analisar um problema
- Desenvolvendo a sua solução de modo eficiente
- 4 fases:
  - Identificação e compreensão do problema
  - Conceptualização da solução
  - Definição do algoritmo para a resolução do problema
  - Implementação da solução através de um programa

# Resolução de problemas

## Escrever um algoritmo

- Simplificado através da decomposição e análise de subproblemas

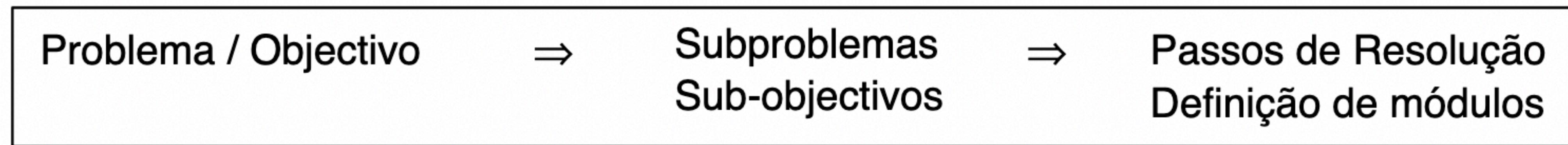


Figura 1: Abordagem para a resolução de problemas

# Resolução de problemas

## Aproximação descendente (top-down approach)

- Permite raciocinar e estruturar a solução dum problema em linguagem natural (ex: Português)
- Facilita o processo de compreensão do problema

# Resolução de problemas

## Mudança de lâmpada

Passo	Descrição
1	<b>Selecione uma nova lâmpada</b>
2	<b>Remova a lâmpada fundida</b>
3	<b>Insira uma nova lâmpada</b>

# Resolução de problemas

## Mudança de lâmpada

Passo	Descrição
1.1	<b>Selecione uma nova lâmpada da mesma potência da fundida</b>
2.1	<b>Posicione a escada em baixo do candeeiro</b>
2.2	<b>Suba a escada até que possa atingir a lâmpada</b>
2.3	<b>Rode a lâmpada no sentido contrário aos ponteiros do relógio até que se solte</b>
3.1	<b>Coloque a nova lâmpada no orifício correspondente</b>
3.2	<b>Rode a lâmpada no sentido dos ponteiros do relógio até que fique presa</b>
3.3	<b>Desça da escada</b>

# Resolução de problemas

## Mudança de lâmpada

- Definição mais precisa para o passo 1.1
  - Selecione uma lâmpada candidata à substituição
  - **Se** a lâmpada não é da mesma potência da antiga, então **repita** até encontrar uma correta:
    - Pouse a lâmpada selecionada
    - Selecione uma nova lâmpada

# Resolução de problemas

## Mudança de lâmpada

- Por exemplo, para os passos 2.2, 2.3, 3.2 e 3.3 poderiam também existir descrições mais precisas e detalhadas, do tipo: Repita... até...
- Aumento do detalhe do algoritmo pode continuar quase indefinidamente
- Grau de detalhe depende das necessidades do agente que vai executar o algoritmo.

# Resolução de problemas

## Lista Telefónica

Passo	Descrição
1	<b>Encontre a página da lista que contém o último apelido do nome</b>
2	<b>Encontre na página determinada no passo 1, o nome procurado</b>

# Resolução de problemas

## Lista Telefónica

Aumentando o detalhe, obtemos instruções elementares não ambíguas

Passo	Descrição
1.1	<b>Coloque o marcador D (dedo) ao acaso na lista</b>
1.2	<b>Abra a lista</b>
1.3	<b>Último nome está contido numa das páginas (esquerda ou direita)?</b> <b>Se sim, siga para o passo 2</b>
1.4	<b>Último nome precede a página esquerda? Se sim, coloque o marcador atrás da página esquerda; se não, coloque o marcador à frente da página direita.</b>
1.5	<b>Vá para 1.2 (retome a sequência de instruções no passo 1.2)</b>

# Resolução de problemas

## Lista Telefónica

Eliminando formulações mal definidas (ex: coloque o marcador atrás)

Passo	Descrição
1.1.1	<b>Torne A igual ao apelido do nome a selecionar (atribuição à variável A)</b>
1.1.2	<b>Escolha uma posição n ao acaso no interval [1,N] (n representa o número de páginas útil da lista)</b>
1.1.3	<b>Torne D igual a n (atribua à variável D o valor de n)</b>
1.2	<b>Abra a lista no local selecionado pelo marcador D</b>
1.3	<b>A está contido numa das páginas (esquerda ou direita)? Se sim, siga para o passo 2</b>
1.4	<b>A precede o primeiro apelido da página esquerda? Se sim, faça n igual a <math>(n+1)/2</math> (atualização do valor de n); se não, faça n igual a <math>(n+n)/2</math></b>
1.5	<b>Vá para 1.2 (retome a sequência de instruções no passo 1.2)</b>

**Um algoritmo é um processo discreto (sequência de acções indivisíveis) e determinístico (para cada passo da sequência e para cada conjunto válido de dados, correspondente uma e uma só acção) que termina quaisquer que sejam os dados iniciais (pertencentes a conjuntos pré-definidos).**

**Um algoritmo é constituído por um conjunto de expressões simbólicas que representam acções (escolher, atribuir, etc.), testes de condições (estruturas condicionais) e estruturas de controlo (ciclos e saltos na estrutura sequencial do algoritmo) de modo a especificar o problema e respetiva solução.**

## **Noção formal de algoritmo**

# Resolução de problemas

## Características de um algoritmo

### Entradas

Quantidades inicialmente especificadas (por exemplo, através de instruções de leitura)

### Saídas

Uma ou mais saídas (habitualmente por instruções de escrita)

### Finitude

A execução deve terminar sempre num número finito de passos

### Precisão

Todos os passos do algoritmo devem ter um significado preciso não ambíguo, especificando exactamente o que dever ser feito. Para evitar a ambiguidade das linguagens humanas, linguagens especiais foram criadas para exprimir algoritmos

### Eficácia

Os passos devem conduzir à resolução do problema proposto. Devem ainda ser executáveis numa quantidade finita de tempo e com uma quantidade finita de esforço

### Eficiência

Em muitos casos colocam-se questões de eficiência a um algoritmo

# Resolução de problemas

**Quantos alunos estão na sala de aula neste momento?**

1. Professor contar os alunos
2. Contar os lugares vazios
3. Estimar baseado no tamanho total do local e multiplicar pelo número de pessoas por m<sup>2</sup>
4. Usar um torniquete
5. Contar o número de filas e, dado que todas tenham o mesmo número de alunos, então bastaria uma simples multiplicação
6. Cada aluno sentado no início de cada fila conta o número de alunos da sua fila, não esquecendo de se conta; Depois soma-se todas as contagens de todos os primeiros da fila
7. Todos os alunos levantam-se e atribuem o número 1; Aos pares, somam o número de cada um, um deles guarda a soma e o outro senta-se; repetir até ficar apenas 1 aluno

# **Quantos alunos estão na sala de aula neste momento?**

## **Professor contar os alunos**

Professor contar os alunos

Ter cuidado para não contar o mesmo aluno 2 vezes

Não esquecer de contar ninguém

### **Vantagens**

**Simples, fácil de executar**

**Solução perfeita para salas de 20 a 30 alunos**

**Não é necessário conhecimentos prévios**

**Não exige equipamentos adicionais**

### **Desvantagens**

**Se o número de alunos for grande, poderá demorar demasiado tempo**

**Quanto maior o número de alunos, maior a possibilidade de haver erros de contagem**

# Quantos alunos estão na sala de aula neste momento?

## Contar os lugares vazios

Contar os lugares vazios

Subtrair o número de lugares com o número  
de lugares vazios

### Vantagens

**Simples, fácil de executar**

**Boa solução para salas com ocupação quase total**

**Não exige equipamentos adicionais**

### Desvantagens

**Necessidade de conhecer antecipadamente o número total de lugares na sala**

**Se a sala tiver com pouca ocupação o método anterior é mais eficaz**

**Não funciona para locais sem numero de lugares definido**

# **Quantos alunos estão na sala de aula neste momento?**

## **Estimar baseado no espaço**

Estimar baseado no tamanho total do local e multiplicar pelo número de pessoas por metro quadrado

### **Vantagens**

**Solução elegante e eficaz**

**Boa solução para locais públicos sem lugares pré-definidos**

### **Desvantagens**

**Necessidade de conhecer, antecipadamente, tamanho do local**

**Não é um método exato**

# **Quantos alunos estão na sala de aula neste momento?**

## **Usar um torniquete**

### **Vantagens**

**Eficaz e exata**

**Boa solução para locais completamente fechados**

### **Desvantagens**

**Necessidade de ter torniquetes em todas as entradas**

**Necessidade de bloquear todos os acessos**

# **Quantos alunos estão na sala de aula neste momento?**

## **Contar o número de filas**

Contar o número de filas e, dado que todas tenham o mesmo número de alunos, então bastaria uma simples multiplicação

### **Vantagens**

**Eficaz e exata**

**Boa solução para paradas de exército**

### **Desvantagens**

**Necessita ter exatamente o mesmo número de pessoas por fila**

# **Quantos alunos estão na sala de aula neste momento?**

## **Cada aluno situado no início duma fila, conta a sua fila**

Cada aluno sentado no início de cada fila conta o número de alunos da sua fila, não esquecendo de se contar a si mesmo

Depois soma-se todas as contagens de todos os primeiros na fila

### **Vantagens**

**Simples**

**Solução escalável**

**Não é necessário ter conhecimentos prévios**

**Não exige equipamentos adicionais**

### **Desvantagens**

**Se o número de alunos for grande, poderá demorar demasiado tempo**

**Quanto maior for o número de alunos, maior a possibilidade de ocorrerem erros**

# Quantos alunos estão na sala de aula neste momento?

## Contagem em pares

Todos os alunos levantam-se e atribuem o número 1.

Em seguida, organizam-se em pares. Em cada par, primeiro é somado o número de cada um dos dois, um deles guarda o número e permanece de pé, o outro senta-se.

Os que ficaram de pé repetem o processo até que fique apenas um aluno de pé

### Vantagens

**Resultado exato**

**Solução escalável**

**Para um grupo de 1000 pessoas, termina em 10 passos. 1.000.000 pessoas, termina em 20 passos**

**Função logarítmica**

### Desvantagens

**Exige organização**

# **Mudar um pneu dum carro**

## **Processo simples**

1. Levantar o carro com o macaco
2. Tirar os parafusos da roda com o pneu furado
3. Tirar a roda do eixo
4. Colocar a roda com o pneu novo no eixo
5. Apertar os parafusos
6. Baixar o carro

# Mudar os 4 pneus dum carro em menos de 8 segundos

## Nelson Piquet

Nos anos 80, na fórmula 1, Nelson Piquet (tricampeão) imaginou que poderia ser campeão se usasse um composto de pneu mais macio e com isso ganhar preciosos segundos aos seus rivais.

**Problema:** necessidade de trocar os 4 pneus porque tinham um desgaste maior.

Nelson Piquet, após alguns cálculos, concluiu que se levasse menos de 8 segundos a trocar os 4 pneus, valeria a pena aplicar este método

# Mudar os 4 pneus dum carro em menos de 8 segundos

## Solução caseira

- Demora cerca de 20 minutos por pneu.
- Com prática poderia ser reduzido para 2/3 minutos por pneu.
- Bastante longe dos 8 segundos desejados.

Problema	Solução
<b>Ter que trocar o macaco para cada roda demora muito tempo</b>	<b>Usar macaco hidráulico e levantar o carro todo duma só vez</b>
<b>Cada roda tem 4 parafusos que terão de ser desenroscados e enroscados</b>	<b>Usar uma aparafuladora elétrica</b>

# Mudar os 4 pneus dum carro em menos de 8 segundos

Problema	Solução
<b>Mesmo com aparafusadora elétrica demora muito tempo aparafusar os 4 parafusos</b>	<b>Usar roda com 1 parafuso</b>
<b>Continua a demorar alguns minutos, pois uma pessoa tem de percorrer as 4 rodas</b>	<b>Com 4 pessoas, cada uma troca uma roda, divide-se o tempo por 4 (menos de 1 minutos, mas superior a 8 seg)</b>
<b>Cada pessoa tem de desapertar, retirar a roda, colocar a nova roda e apertar</b>	<b>Com 3 pessoas por roda, 1 desaperta, 1 retira o pneu e outra coloca o novo pneu ( a que desapertou, pode</b>

**Solução final:** Com 3 pessoas por roda mais 2 para levantar o carro todo duma vez, conseguimos baixar o tempo para menos de 8 seg ( hoje em dia muda-se os 4 pneus em menos de 3 seg)

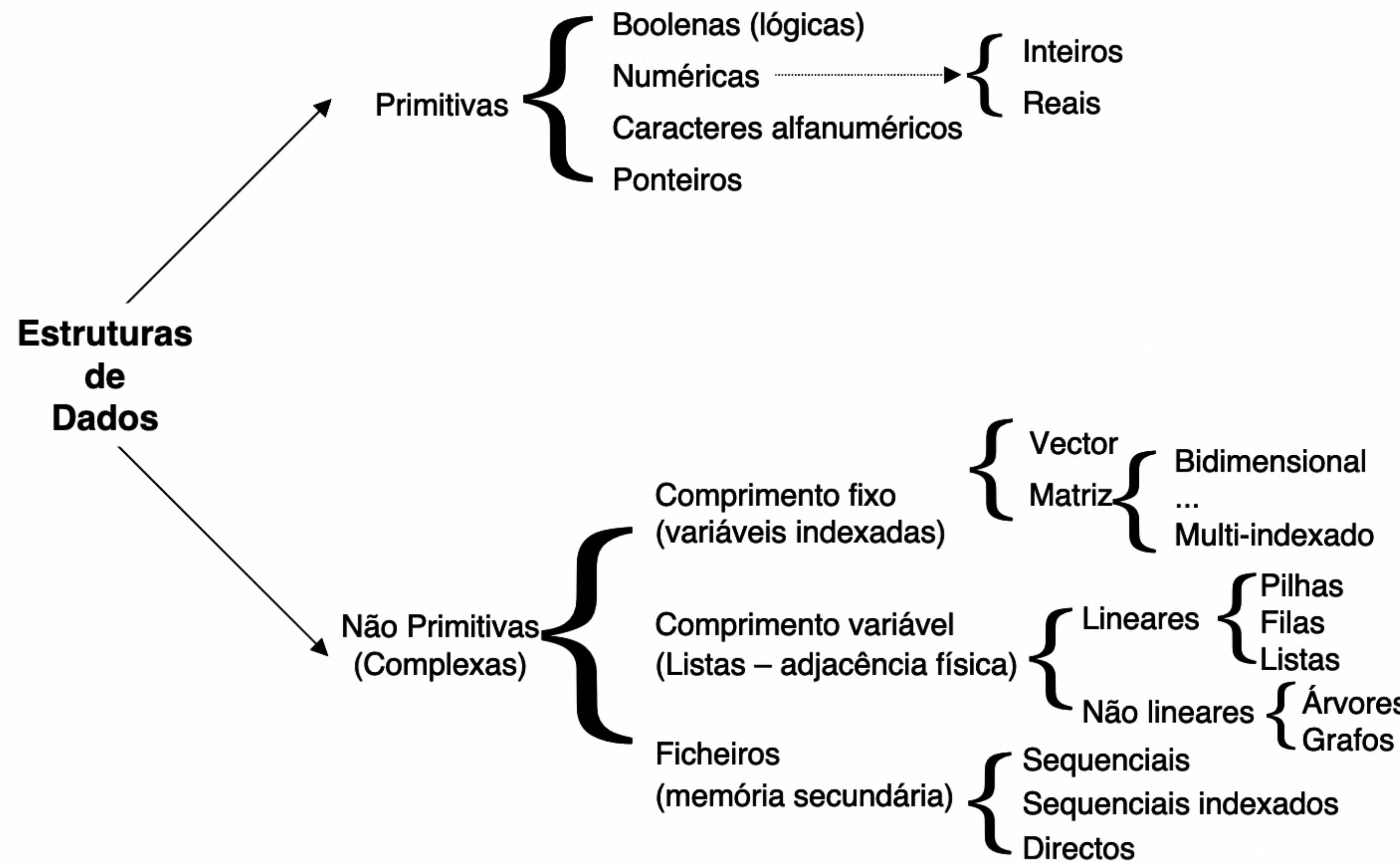
# Estruturas de Dados

# Estruturas de dados

## O que são?

- A resolução de problemas através de algoritmos requer a representação de:
  - entidades e;
  - objetos reais
- As diferentes formas nos quais os itens de dados são logicamente relacionados definem diferentes estruturas de dados

# Estruturas de dados



# Estruturas de dados

## Definição de estruturas

- A definição de estruturas de dados tem subjacente as operações a executar nos dados representados
- Diferentes operações poderão ser executadas:
  - Criar e eliminar estruturas de dados
  - Operações para inserir, alterar e eliminar elementos da estrutura de dados
  - Operações para aceder aos elementos da estrutura de dados

# Estruturas de dados

## Síntese

- Uma eficiente manipulação de dados envolve uma análise das seguintes questões
  - A. Compreender a relação entre dados
  - B. Decidir operações a executar nos dados logicamente relacionados
  - C. Representar os elementos dos dados de modo a:
    1. Manter as relações lógicas entre os dados
    2. Executar de forma eficiente as operações nos dados
  - D. Construir o algoritmo e escolher a linguagem de programação mais apropriada que permita de modo ‘natural’ e ‘expressivo’ representar as operações executadas nos dados.

# Estruturas de dados primitivas

## Tipo de dados boleado

- Permite representar dois e apenas dois estados: *verdadeiro* e *falso*
- Pode ser representado pelos dois estados existentes na codificação binária: *1-verdadeiro* e *0-falso*
- É aplicado em situações reais que unicamente denotam dois estados possíveis.

# Estruturas de dados primitivas

## Tipo de dados numérico

- Representativo dos valores numéricos no domínio dos números inteiros e reais
- 37 é um tipo numérico inteiro
- $\sqrt{2}$  é um tipo numérico real

# Estruturas de dados primitivas

## Tipo de dados alfanumérico

- Sistema de codificação designado ASCII (American Standard Code for Information Interchange)
- O conjunto de caracteres ASCII permite representar:
  - Alfabeto (minúsculas e maiúsculas)
  - Caracteres numéricos decimais {0,1,2,3,4,5,6,7,8,9}
  - Operadores e caracteres especiais
  - Caracteres de controlo - DEL- delete, CR - carriage return, HT - horizontal tab, etc

# Estruturas de dados

## Representação

- Os tipos de dados referidos são representados na memória principal de diferentes formas:
  - Dados simbólicos
  - Dados numéricos
  - Informação contextual

# Estruturas de dados

## Representação simbólica

- Letras - {a..z, A..Z}
- Dígitos decimais - {0..9}
- Sinais de operação - {+ - \* /}
- Sinais de pontuação - {., ; : ? !}
- Símbolos especiais - {" \$ # % ' |}
- Os símbolos são codificados em binário num código de comprimento n. Os códigos usuais têm 8 bits (n=8, 256 símbolos representáveis)

# Estruturas de dados

## Representação numérica

- Os dados representam quantidades numéricas no sistema de numeração binário
- $0101\ 1011 =$   
 $0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 64 +$   
 $16 + 8 + 2 + 1 = 91$

# Estruturas de dados

## Informação contextual

- Conteúdo informativo (significado) de um dado, depende do contexto.
- Por exemplo 0100 0001 pode representar:
  - 65 - operação binária
  - 41 - operação utilizando BCD (binary code decimal)
  - A - símbolo ASCII
  - MOV A,B - instrução do microprocessador INTEL 8080

# Estruturas de dados

## Tipos de dados vs Variáveis vs Constantes

- $X \leftarrow 47$ 
  - Significa que à variável  $X$  foi atribuído o valor 47. Esta operação de atribuição tem por pressuposto o facto de  $X$  ser uma variável do *tipo inteiro*
- $CAR \leftarrow 'G'$ 
  - Significa que à variável  $CAR$  foi atribuído o valor ‘G’. Esta operação de atribuição tem por pressuposto o facto que  $CAR$  é uma variável do *tipo alfanumérico*. Por convenção, os valores alfanuméricos são representados entre plicas.

# Estruturas de dados

## Nomes de variáveis

- Uma variável representa um determinado valor e o seu nome é escolhido de forma a reflectir o valor que representa
  - A variável MAX pode representar o valor máximo de um conjunto de valores numéricos.
- Por convenção, o nome duma variável **não deve** conter espaços, **não deve** iniciar por números e **não deve** conter determinados caracteres especiais.
- O nome duma variável **deve** iniciar-se sempre por uma letra seguido dum conjunto de caracteres incluindo letras, números e alguns caracteres especiais como, por exemplo, o ‘\_’.

# Estruturas de dados

## Variáveis válidas

- NUM
- N\_ALUNO
- NOMEALUNO
- NOME\_ALUNO
- X1
- Y2

# Estruturas de dados

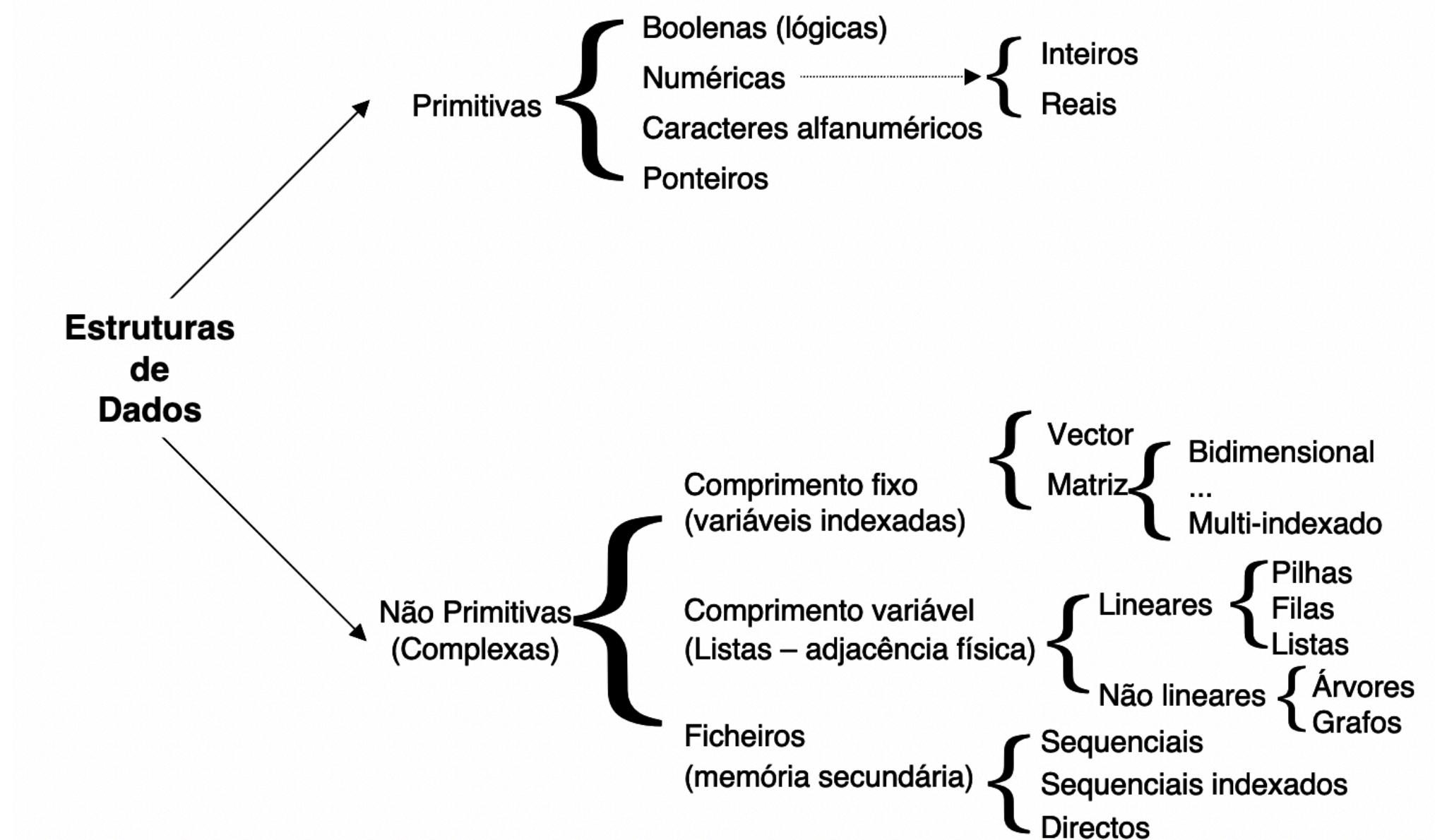
## Variáveis inválidas

- 7NUM
- Y-Z
- T/4
- U\*VAR
- DUAS-PAL
- DUAS PALAVRAS

# Estruturas de dados não primitivas

## O que são

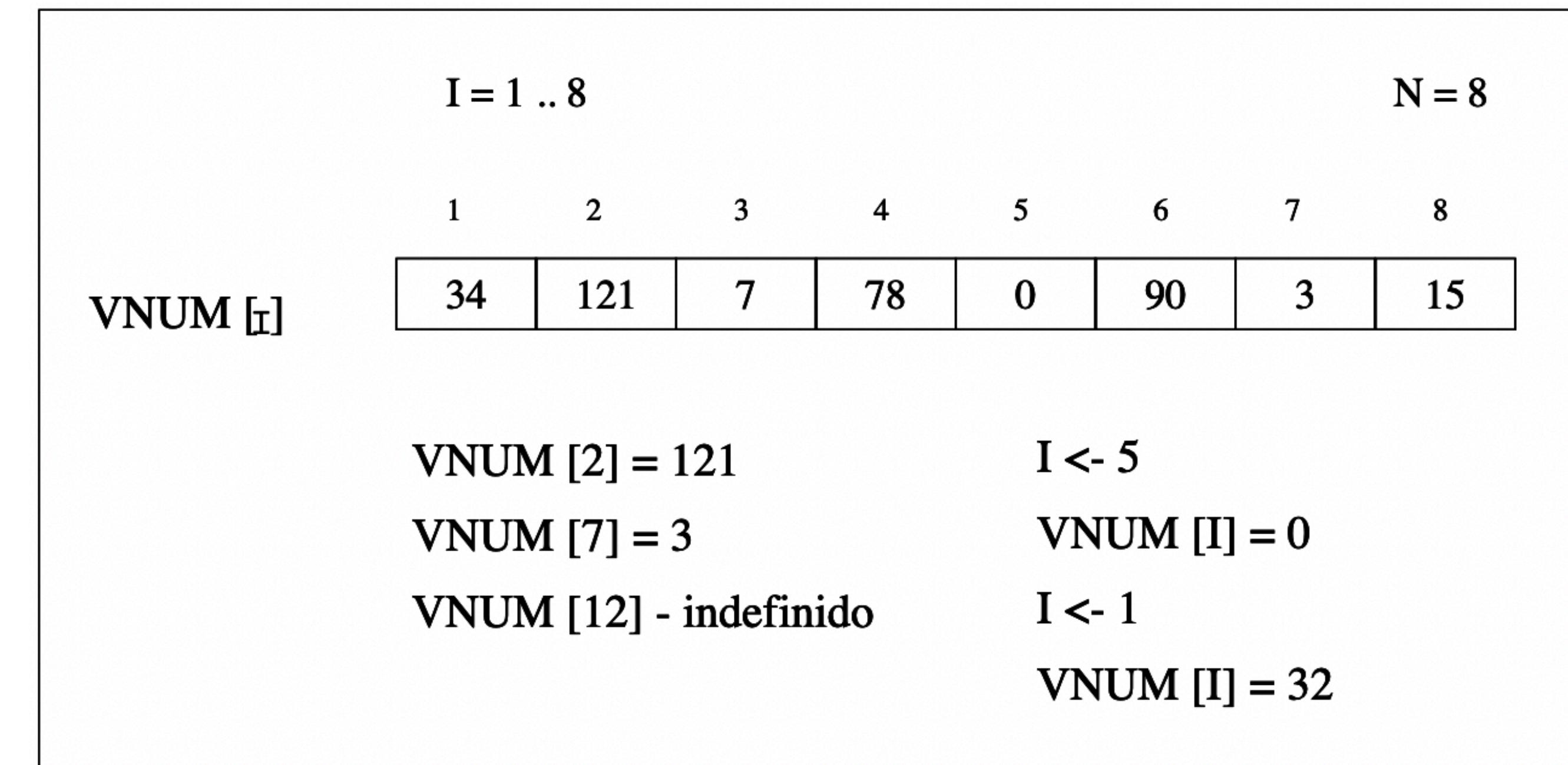
- São definidas através de conjuntos de estruturas de dados primitivas.
- No contexto desta disciplina, serão estudadas as estruturas de dados de comprimento fixo e algumas exemplos de ficheiros de dados e suas aplicações



# Estruturas de dados não primitivas

# Vetores

- Elementos dum vetor (array) são representados através de conjuntos de variáveis de um determinado tipo de dados.



# Estruturas de dados não primitivas

## Matrizes

- Uma matriz pode ser interpretada como um vetor bidimensional

MAT [ I , J ]						
Índice representativo das linhas da matriz N = 7	Índice representativo das colunas da matriz J					
	1	2	3	4	5	6
	1	55	12	72	8	15
	2	121	67	17	78	12
	3	34	4	71	7	54
	4	56	12	12	7	56
	5	34	21	15	8	0
	6	76	32	78	78	56
	7	43	221	321	77	45

MAT [2,3] = 17                                    I <- 3  
MAT [7,1] = 43                                    J <- 5  
MAT [4,12] - indefinido                        MAT [I,J] = 54

# Notação algorítmica

# Notação algorítmica

## Prefácio

- Na fase de desenvolvimento dum algoritmo podemos utilizar:

- Linguagem auxiliar (pseudo-código)

Linguagem de representação de alto nível que pode ser traduzida em qualquer linguagem de programação

- Diagrama de Fluxo (fluxograma)

Notação gráfica

- Linguagens de programação (ex: C, C++, Java, C#, etc.)

# Notação algorítmica

## Pseudo-código

- É uma notação algorítmica muito próxima da linguagem natural.
- Um algoritmo é identificado por um nome que deverá ser elucidativo do problema em causa.
- Por convenção, o nome do algoritmo é seguido duma breve descrição das tarefas executadas pelo algoritmo.
- Um algoritmo é construído através de uma sequência de passos numerados.
  - Cada passo deverá conter um comentário explicativo da tarefa a executar no contexto global do algoritmo

# Pseudo-código

## Regras

1. Nome do algoritmo em letras maiúsculas
2. Breve comentário das tarefas a desenvolver pelo algoritmo
3. Conjunto de passos numerados e comentados
4. Estruturas de dados, funções e sub-rotinas em letra maiúscula
5. Estruturas de controlo: primeira letra em maiúscula.
6. O algoritmo termina com a instrução *Exit* (fim lógico). O símbolo representa o fim físico do algoritmo.

# Pseudo-código

## Factorial dum número inteiro

Algoritmo FACTORIAL. Este algoritmo calcula o factorial de um número inteiro.

1. [Leitura do número]

Read(N)

2. [Caso particular de N = 0 ou N = 1]

If N=0 Or N=1

Then FACT ← 1

3. [Outros números]

Else FACT← N

NUM ← N -1

Do For I = NUM To 1 Step -1

    FACT ← FACT \* I

4. [Impressão do resultado (N!)]

Print('O Factorial de ',N,', é igual a ', FACT)

5. [Termina]

Exit

# Pseudo-código

## Instrução de atribuição

- É representada através do símbolo  $\leftarrow$ 
  - Que é colocado à direita da variável que recebe o valor da atribuição (ex: FACT  $\leftarrow$  1).
  - Ter em atenção a diferença entre o sinal de atribuição  $\leftarrow$  e o sinal  $=$  que é utilizado como operador relacional.
- Exemplos:

VAR  $\leftarrow$  12/7

FACT  $\leftarrow$  FACT \* NUM

N  $\leftarrow$  MOD( NUM1, NUM2)

# Pseudo-código

## Leitura e escrita de dados

- É possível obter (ou ler) valores de variáveis, assim como escrever (ou imprimir) os valores dessas variáveis através de:
  - instruções de leitura (entrada de dados) e;
  - instruções de escrita (saída de dados).

### • Leitura de dados

Sintaxe:

Read(<nome da variável>) ex: Read (N\_ALUNO)

### • Saída de dados

Sintaxe:

Write (<nome variável>) ex: Write (N\_ALUNO) ou

Write (<'texto'>) ex: Write('so texto...') ou

Write (<nome da variável, 'texto'>) ex: Write ('O número de alunos é igual a ', N\_ALUNO)

Write ('texto', <nome da variável 1>, 'texto', <nome da variável 2, 'texto,...>) ex: Write ('O número ',N\_ALUNO,' refere-se ao n.º de alunos da disciplina de programação')

# Pseudo-código

## Estrutura condicional

- **If Statement - If ... Then ... Else**
- Esta declaração representa um teste de condição lógica (se ... então ... senão).
- É executado um conjunto de instruções consoante a condição especificada for verdadeira ou falsa.
- Pode ter uma das seguintes formas:

If Condition  
Then \_\_\_\_\_  
\_\_\_\_\_

If Condition  
Then \_\_\_\_\_  
Else \_\_\_\_\_  
\_\_\_\_\_

# Pseudo-código

## Estrutura condicional

- A seguir à clausula **Then** um conjunto de instruções é executado no caso da condição ser verdadeira.
- Caso contrário (**Else**) será executado o conjunto de instruções a seguir à clausula *Else*

If N > 10 Then

X ← (X + 15) /2

Print(X)

If N > 10 Then

X ← (X + 15) /2

Print(X)

Else

X ← X \* 5

Print(X)

# Pseudo-código

## Instruções de repetição

- Existem 3 tipos de instruções que permitem controlar iterações ou ciclos de processamento.
  - Do while <logical condition>
  - Repeat until <logical condition>
  - Do for INDEX = <numerical sequence>

# Pseudo-código

## Do...While

- É utilizada quando é necessário repetir um conjunto de passos em função de uma determinada expressão lógica.
- Estes passos são repetidos enquanto a expressão lógica for verdadeira.

**Do While <logical condition>**

\_\_\_\_\_

\_\_\_\_\_

...

Sintaxe

Read (Z)

X ← 15

Do While Z >= 135

X ← X-5

Z ← (X<sup>4</sup>) /3

Print(X,Z)

Exemplo

# Pseudo-código

## Repeat Until

- É utilizada quando é necessário repetir um conjunto de passos em função duma determinada expressão lógica.
- Os referidos passos são repetidos até que a expressão lógica se torne verdadeira.

**Repeat until <logical condition>**

\_\_\_\_\_

\_\_\_\_\_

...

Sintaxe

$X \leftarrow 1$

Repeat until  $K > 75$

    Write( NOME\_ALUNO[K] )

$K \leftarrow K + 1$

Exemplo

# Pseudo-código

## Do For

- É utilizada quando é necessário repetir um conjunto de passos um determinado número de vezes.

**Do For INDEX = N1 to N2 Step P**

---

---

...

Sintaxe

X← 10  
Do For I = 1 to 70  
    X ← X + 5  
Print (X)

Exemplo

X← 0  
Do For J = 10 to 100 Step 10  
    X ← X + 25  
Print (X)

Exemplo

- INDEX indica o índice que é incrementado em cada ciclo de processamento.
- N1 e N2 referem dois números inteiros representativos do intervalo inferior e superior da sequência de números a executar.
- A cláusula **Step** determina o incremento na sequência numérica. Por defeito o **Step tem valor 1**.

# Pseudo-código

## Operações e expressões aritméticas

- A notação algorítmica inclui operações e funções matemáticas que permitem efetuar variados cálculos aritméticos.
- Dois tipos de valores numéricos: Inteiros e Reais
- Regras de precedência:
  1. Parêntesis - ()
  2. Exponenciação - ( $\uparrow$ )
  3. Multiplicação - (\*)
  4. Divisão - (/)
  5. Adição - (+)
  6. Subtração - (-)

# Pseudo-código

## Operações e expressões aritméticas

- Determinadas funções matemáticas podem ser utilizadas na definição de expressões computacionais:
  - mod(M,N) - função que retorna o resto da divisão de M por N
  - int(NUM) - função que retorna a parte inteira de um número real.
  - sqr(NUM) - função que retorna a raiz quadrada de um número inteiro ou real.

X  $\leftarrow$  (A+B)  $\uparrow$ 3 - (A-A/3) \* B

N  $\leftarrow$  mod(20,6)

RNUM  $\leftarrow$  sqr(NUM)

# Pseudo-código

## Operadores e operações relacionais

- Os operadores matemáticos relacionais ( $=$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$ ) têm um conjunto de símbolos correspondentes a nível computacional, respetivamente,  
 $=$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$  ou  $!=$
- As expressões lógicas representam relações entre valores do mesmo tipo de dados.
- O resultado da avaliação de uma expressão lógica pode ter um de dois valores possíveis: **true** (verdadeiro) ou **false** (falso).

$A \leftarrow 20$

$A \leq 10/3 + 5$

A primeira relação tem valor falso e a segunda relação tem valor verdadeiro

$A < A + 10$

# Pseudo-código

## Operadores e operações lógicas

- A notação algorítmica inclui os seguinte operadores lógicos:

Operador	Notação
Negação	not
Conjunção	and
Disjunção	or

Precedência	Operador
1	Parêntesis
2	Aritméticos
3	Relacional
4	Lógico

Considere que NUM tem o valor 1

- (NUM < 2) and (NUM < 0)
- (NUM < 2) or (NUM < 0)
- not( NUM < 2)

As expressões 1,2 e 3 têm valores falso, verdadeiro e falso, respectivamente

# Pseudo-código

## Exercício: Máximo divisor comum

Pretende-se implementar em pseudo-código o máximo divisor comum entre dois números.

Recorde-se que o máximo divisor comum entre dois números é o maior número que é divisível por ambos.

# Pseudo-código

## Exercício: Máximo divisor comum

Read (M,N)

If M < N Then

    MIN ← M

Else

    MIN ← N

MDC ← MIN

FLAG ← 0

Do While FLAG = 0

    R1 ← mod(M, MDC)

    R2 ← mod(N, MDC)

    If R1 = 0 and R2 = 0 Then

        FLAG ← 1

    Else

        MDC ← MDC -1

Write('O máximo divisor comum de ', M, 'e', N, 'é igual a ', MDC)

Exit

# **Diagramas de Fluxo**

# **Diagramas de Fluxo**

## **Fluxogramas**

Um diagrama de fluxo representa de uma forma gráfica as instruções e respetivas operações incluídas em determinado algoritmo.

O objetivo principal é facilitar a compreensão e desenvolvimento de algoritmos para a resolução de problemas através da utilização e composição de símbolos gráficos

# Diagramas de Fluxo

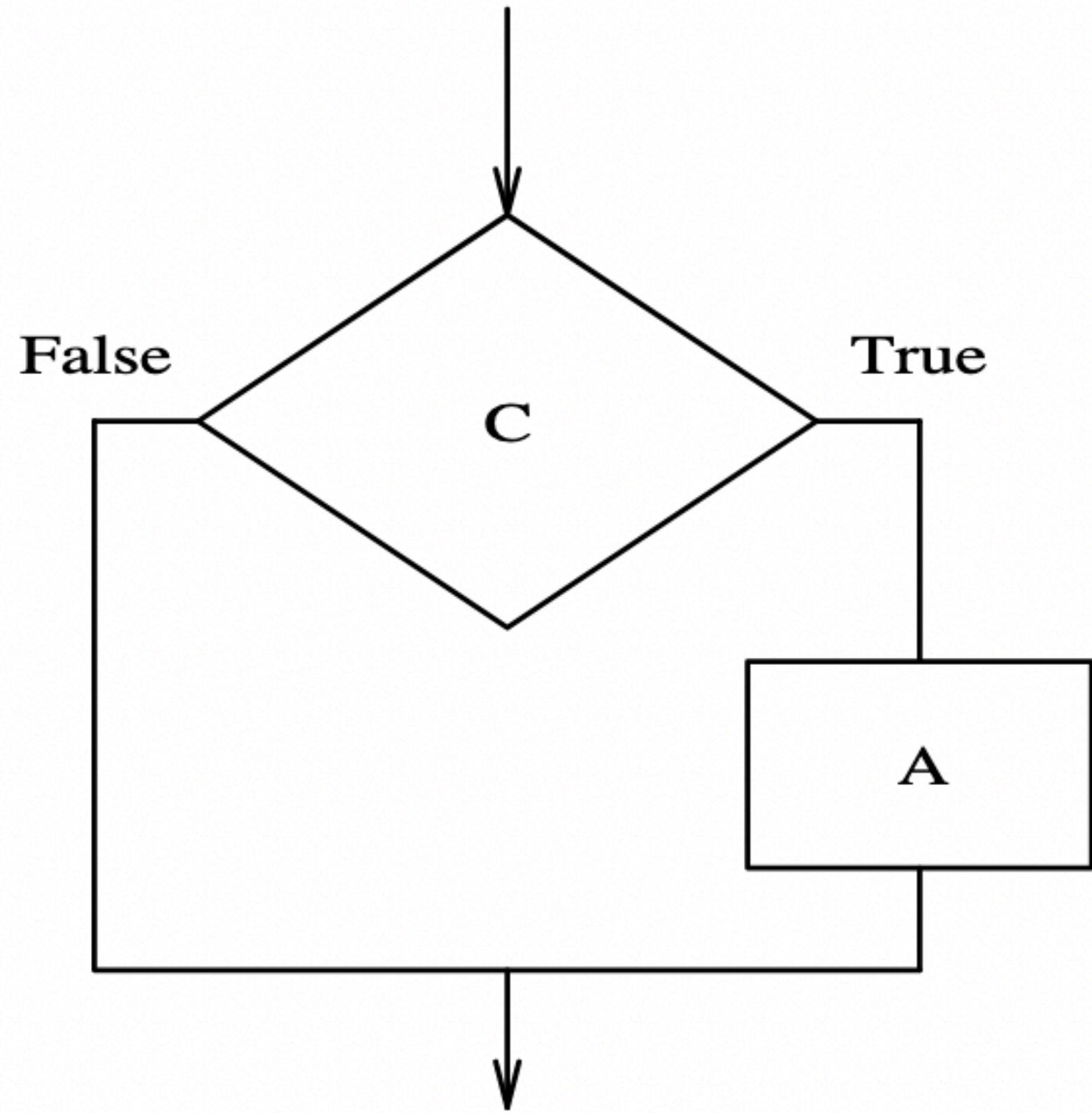
## Notação gráfica

Símbolo	Significado
(Retângulo)	Inicio / fim do algoritmo
(Trapezóide)	Leitura e escrita (entrada e saída) de dados
(Retângulo)	Instruções de atribuição
(Rombo)	Estrutura condicional

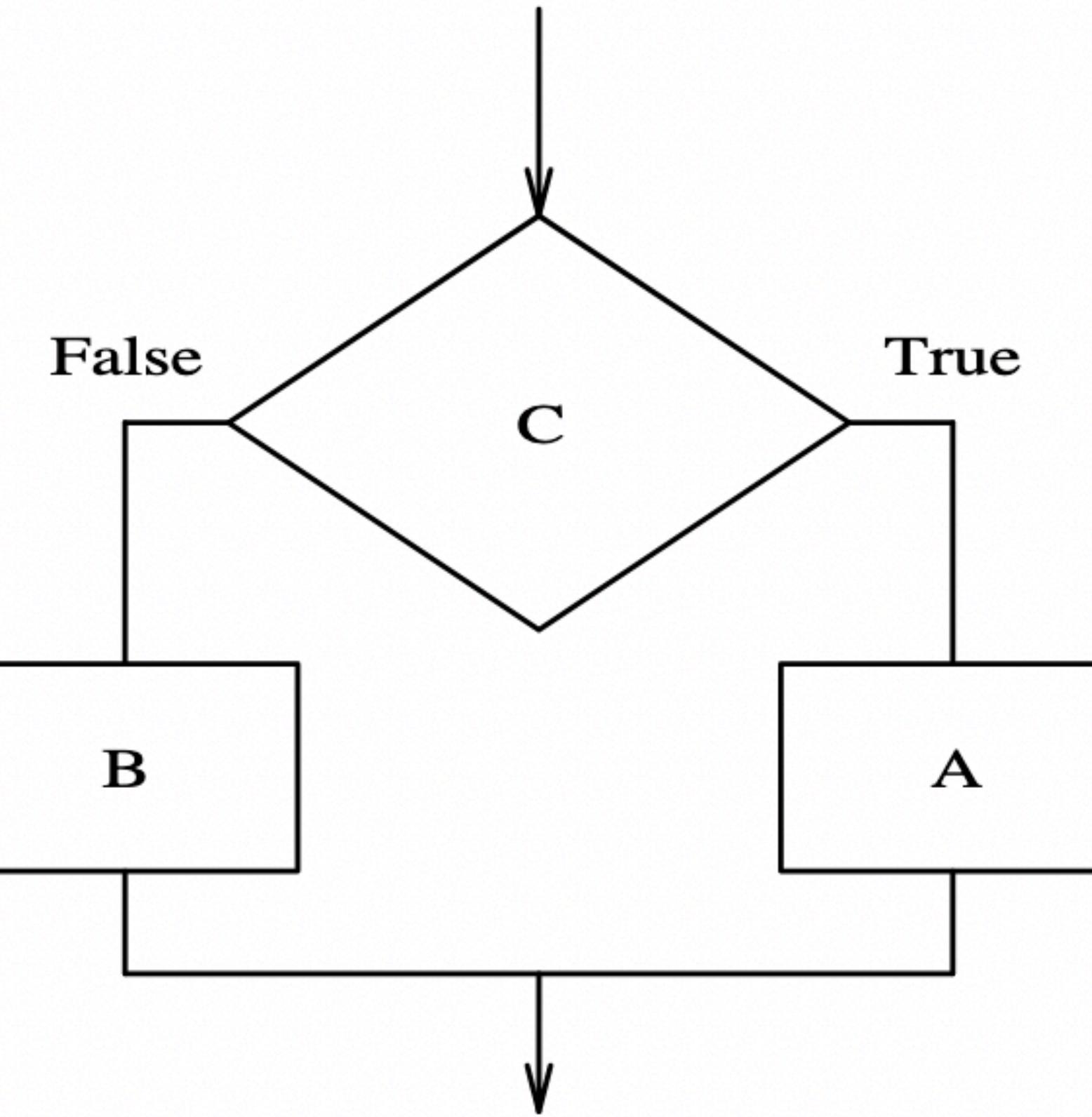
# Diagramas de Fluxo

## Estrutura If...Then...Else

IF C THEN A



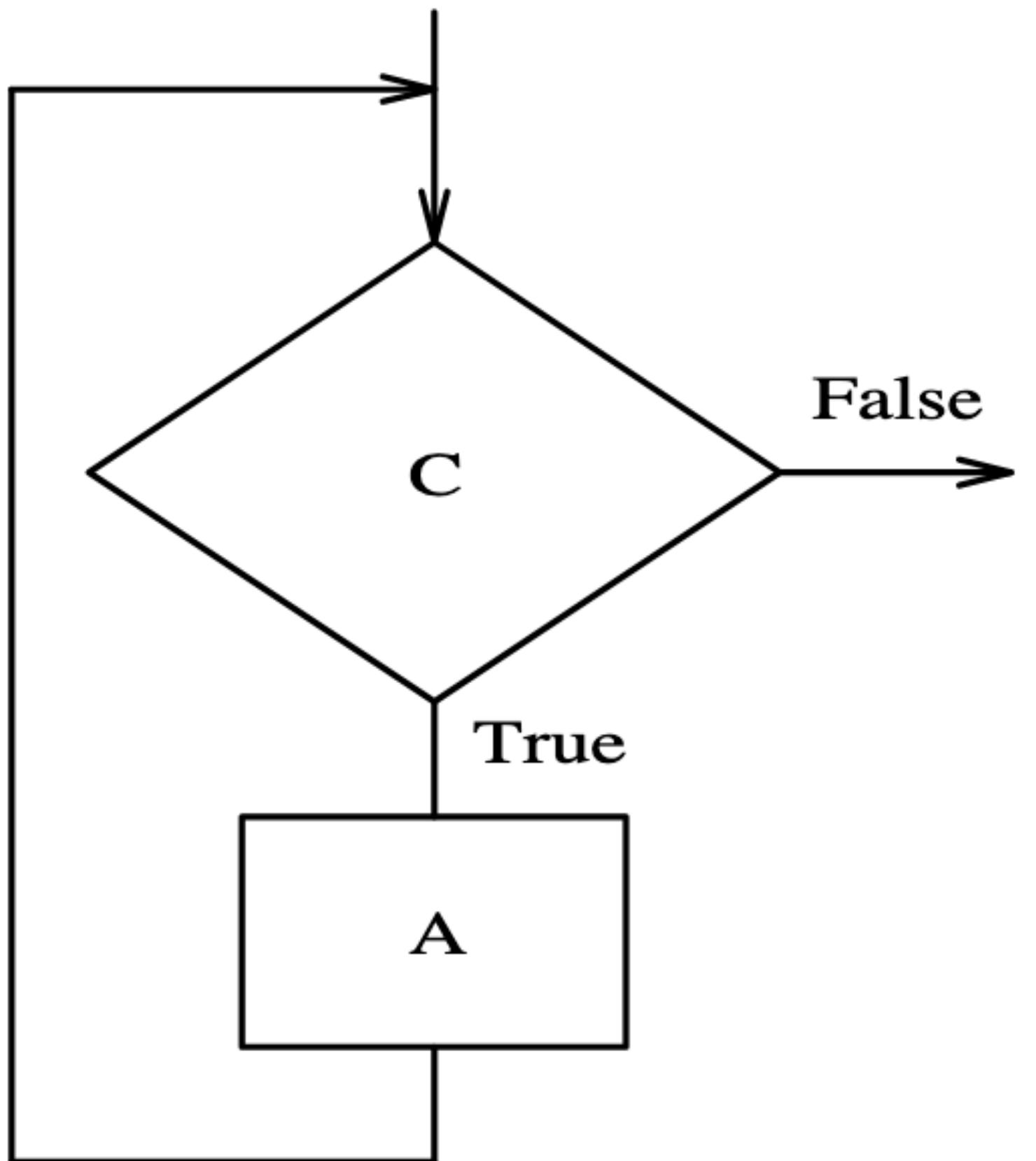
IF C THEN A ELSE B



# Diagramas de Fluxo

## Estrutura Do...While

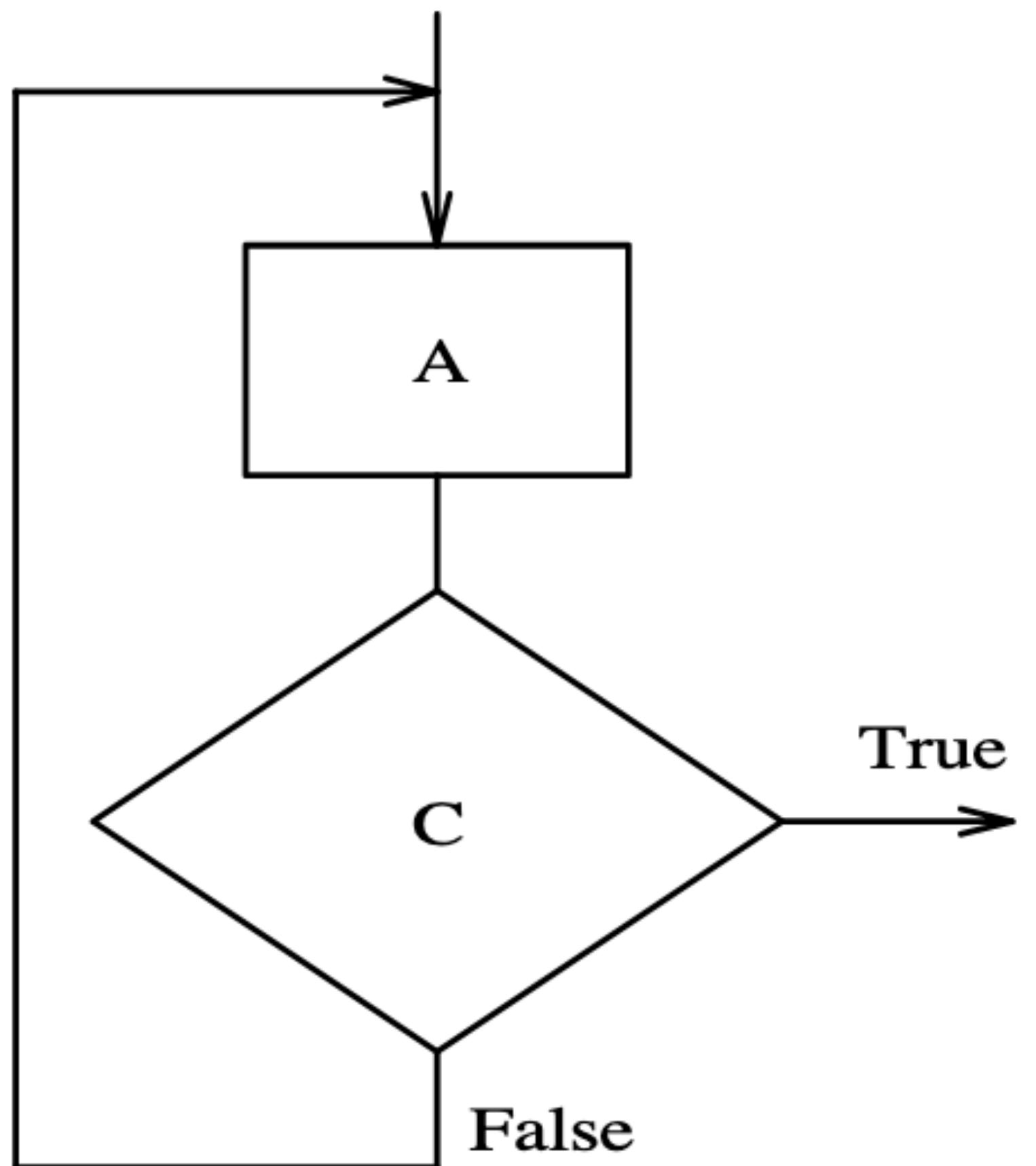
**WHILE C DO A**



# Diagramas de Fluxo

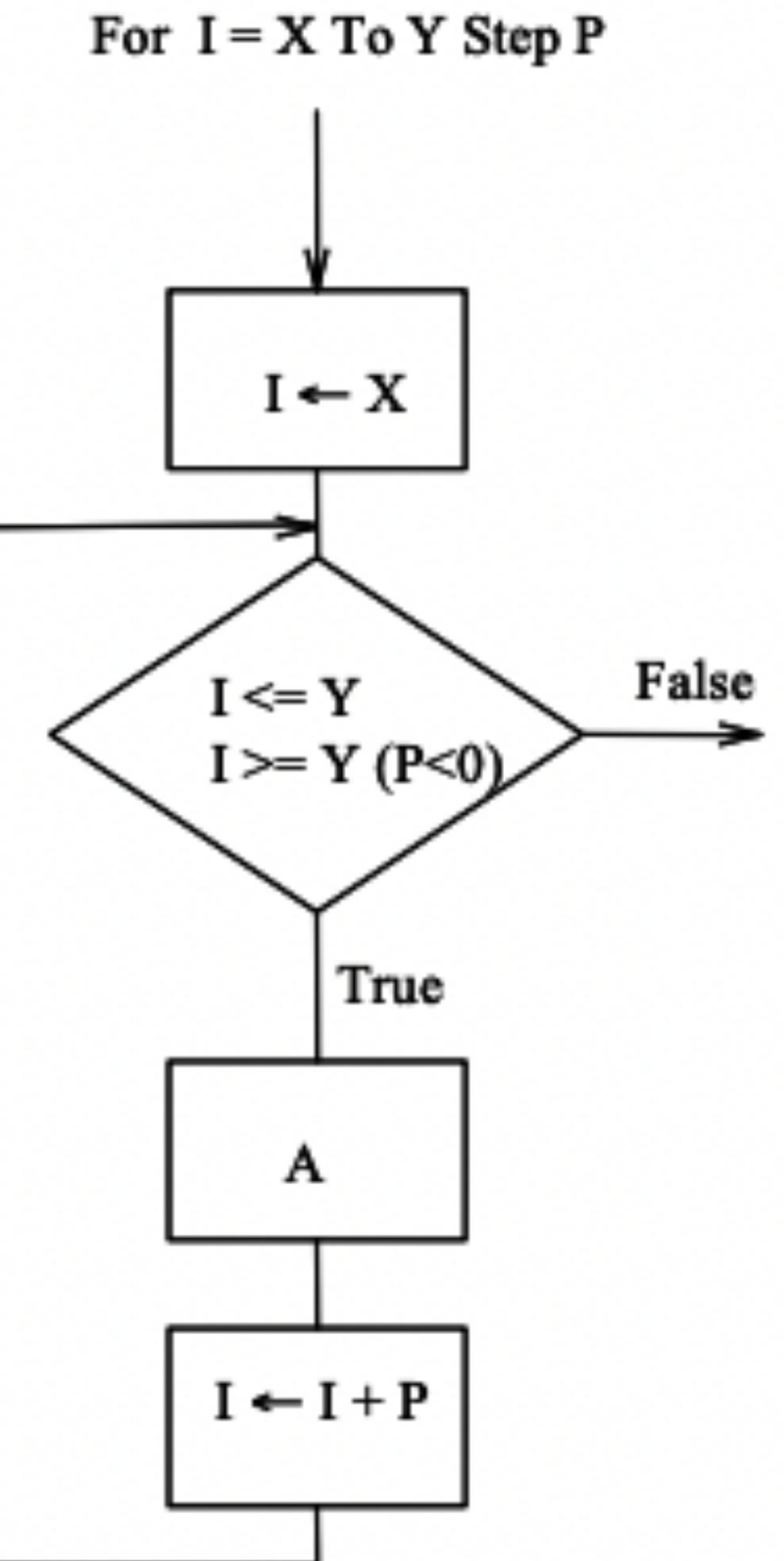
## Estrutura Repeat...Until

REPEAT A UNTIL C



# Diagramas de Fluxo

## Estrutura For...To



# Pseudo-código/Fluxograma

## Soma valores

Este algoritmo determina o somatório de um determinado número de valores numéricos a introduzir pelo utilizador.

Read('Nº de valores a somar:', N)

SOMA  $\leftarrow$  0

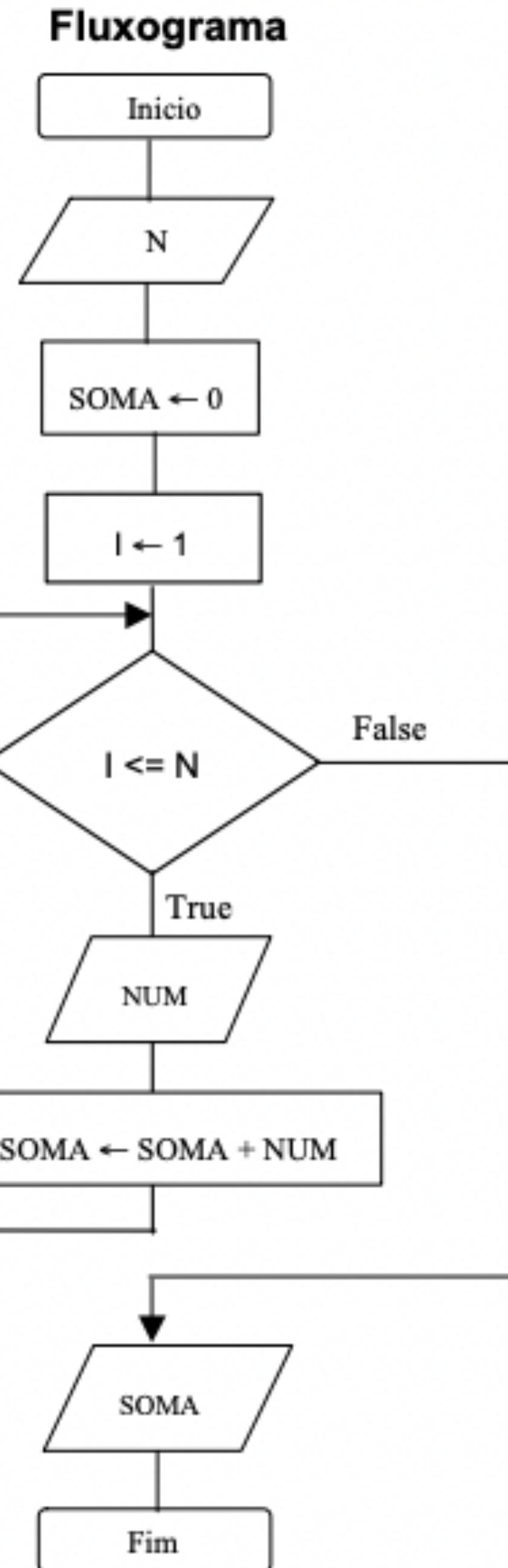
Do For I = 1 to N

    Read('Introduza o Nº:', NUM)

    SOMA  $\leftarrow$  SOMA + NUM

Write('O somatório é igual a ', SOMA)

Exit



# Prova e Teste de Algoritmos

A prova e teste de algoritmos e programas permite seguir a execução de um algoritmo passo a passo e verificar a evolução de todas as variáveis

Este processo é apresentado através de uma tabela em que deve constar todas as variáveis, testes lógicos efetuados, entrada de dados e saída de resultados.

M	N	MIN	mdc	R1	R	FLAG = 0	R1 = R2	Entrada/Saída
8	20							Read(8.20)
		8						
			8					
				0				
					4			
						V	F	
		7						
			1					
				6				
					V	F		
		6						
			2					
				2				
					V	F		
		5						
			3					
				0				
					0			
		4				V	F	
			0					
				0				
					F	V		Escreve(4)

Read (M,N)

If M < N Then

MIN ← M

Else

MIN ← N

MDC ← MIN

FLAG ← 0

Do While FLAG = 0

R1 ← mod(M, MDC)

R2 ← mod(N, MDC)

If R1 = 0 and R2 = 0 Then

FLAG ← 1

Else

MDC ← MDC -1

Write('O máximo divisor comum de  
, M, 'e',N, 'é igual a ', MDC)

Exit

# Linguagem C

# Linguagem C

## Fundamentos da Linguagem C

- Uma linguagem para **programação de sistema**
  - Sistemas de operação
  - Editores, compiladores e outras ferramentas
  - Bibliotecas
  - e.g: *kernel* linux, base de dados SQLite
- Desenvolvida nos anos 70 para o sistema UNIX
- Popularizado a partir dos anos 80 em outros sistemas
- Micro-computadores, PCs MS-DOS/Windows, Apple Macintosh,...
  - Micro-controladores, sistemas embutidos
- Nestas aulas vamos usar o C99

# Linguagem C

## Linguagens baseadas em C

- C++
  - Extende o C com abstrações para modularidade (classes, programação com objetos, tipos genéricos)
- Java
  - Linguagem baseada no C++ com gestão automática de memória e execução numa máquina virtual
- C#
  - Linguagem derivada do C++/Java com execução em plataforma .NET
- Go
  - Linguagem de programação de sistemas com sintaxe leve e mecanismos para concorrência

# Linguagem C

## Características do C

- Linguagem relativamente pequena
- Baixo-nível (e.g. próxima do funcionamento do computador)
- Permissiva ( e.g. permite ao programador controlo sobre a execução do programa)

# Linguagem C

## Vantagens do C

- Eficiência
  - Permite ao programar controlar o uso de recursos (tempo, memória)
  - Importante em sistemas críticos (e.g tempo real)
- Portabilidade
  - Existem compiladores de C para praticamente todos os processadores e sistemas operativos
- Expressividade
  - Suporta operações de nível de máquina
  - e.g. inteiros *signed* e *unsigned*, operações sobre *bits*, apontadores
- Bibliotecas
  - A linguagem C não inclui funções para I/O ou matemáticas
  - São disponibilizadas em bibliotecas externas
- Integração com sistema UNIX

# Linguagem C

## Desvantagens do C

- Obriga o programador a especificar muitos detalhes de implementação
  - e.g. Gerir a alocação/libertaçāo de memória explicitamente
- Programas em C podem ser difíceis de compreender e modificar
- É fácil introduzir erros de difícil deteção
  - *Buffer overflows, memory leaks, use-after-free,...*
- Atualmente uma das **maiores fontes de problemas de fiabilidade e segurança** em software.

# Linguagem C

## Aprendizagem

- Conhecer os perigos latentes mais comuns
- Aprender a usar as ferramentas para detetar erros (*warnings, debugger, testes*)
- Adoptar estilo de programação claro e conciso
- Evitar “truques” e código excessivamente complexo
- Programar dentro do *standard*

# Linguagem C

## Programas

- Programas C são ficheiros de texto
- Compostos usando um editor de texto( GNU Emacs, Atom,...)
- Convenção: nome do ficheiro termina com extensão .c (letra minúscula)

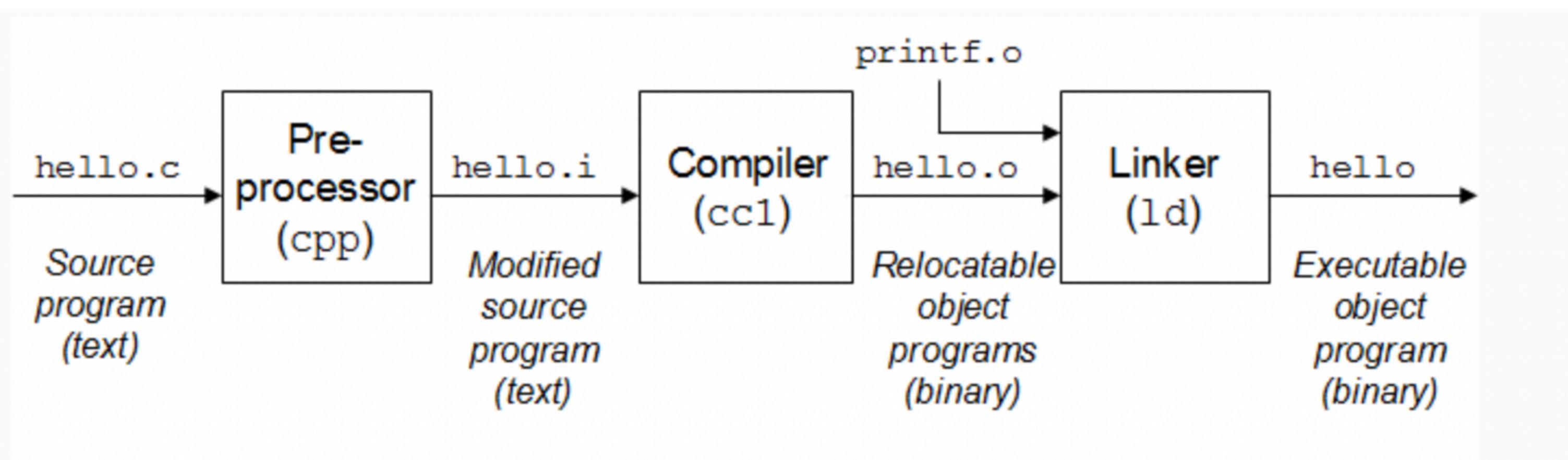
```
#include <stdio.h>

int main(void)
{
    printf("To C or not to C, ");
    printf("that is the question.\n");
    return 0;
}
```

# Linguagem C

## Execução

- Para ser executado um programa em C tem primeiro de ser traduzido para *código-máquina*.
- A tradução é efetuada por programa **compilador**
- Vamos usar o GCC (GNU Compiler Collection)



Em sistemas GNU/Linux: o pré-processador, compilador e ligado são executados em sequência pelo comando `gcc`.

### Pré-processamento

O pré-processador interpreta diretivas (linhas que começam por #)

### Compilação

O compilador traduz o código C para código-máquina

### Ligaçāo

O linker combina o código-máquina gerado com as bibliotecas necessárias

# Linguagem C

## Compilar, ligar e executar

Invocamos o compilador usando o interpretador de comandos linux (*shell*):

```
$ gcc -o hello hello.c
```

Produz um ficheiro *hello* que podemos executar:

```
$ ./hello
To C or not to C, that is the question.
```

# Linguagem C

## Estrutura de programas simples

diretivas

```
int main(void)
{
    instruções
}
```

### Diretivas

- Uma diretiva é indicada por uma linha começada em #; e.g:

```
#include <stdio.h>
```

- A linguagem C inclui ficheiros *header* com declarações de bibliotecas
- *stdio.h* contém as definições associadas a entrada/saída
- Exemplo: *printf* está declarado neste *header*

# Linguagem C

## Funções

- Uma função agrupa uma sequência de instruções com um nome
- Uma implementação da linguagem C disponibiliza várias *bibliotecas* com funções pré-definidas
- O resultado duma função é especificado com a instrução *return*

# Linguagem C

## Função *main*

- Um programa completo deve definir uma função *main* que é executada quando o programa inicia.
- O valor retornado de *main* representa o *código de erro* para o sistema operativo
- Retornar zero significa que o programa terminou corretamente

```
int main(void){  
    ....  
    return 0;  
}
```

- Em C99 podemos omitir o *return* (equivale a retornar 0)

# Linguagem C

## Instruções

- O corpo de um função é uma sequência de *instruções*

```
printf("To C or not to C, ");
printf("that is the question.\n");
return 0;
```

- Este exemplo usa apenas dois tipos de instruções: chamadas da função *printf* e *return*
- A chamada *printf*("...") imprime o texto entre aspas na saída-padrão (terminal)
- Imprime a seguinte mensagem: *To C or not to C, that is the question.*

# Linguagem C

## Instruções

- Cada instrução termina com ponto-e-vírgula (;)
- Uma instrução pode ser dividida em várias linhas:

```
printf(  
    "To C or not to C, "  
);
```

- Também podemos escrever várias instruções numa linha:

```
printf("To C or "); printf("not to C, ");
```

- As *diretivas* ocupam, normalmente, apenas uma linha: **não** necessitam de ponto-e-vírgula

# Linguagem C

## Comentários

- Um *comentário* começa com /\* e termina com \*/
- Comentários podem ocorrer em linhas separadas ou no meio de linhas de código
- Podem extender-se por várias linhas

```
/* Isto é um comentário */

/*
    Autor: Pedro Vasconcelos
    Ficheiro: hello.c
    Programa: Imprime uma mensagem de exemplo
*/
```

- **Atenção:** esquecer de fechar um comentário pode fazer com que o compilador ignore parte do programa

```
printf("To be ");      /* comentário aberto
printf("or not to be; "); /* fechado */
printf("that is the question.\n");
```

# Linguagem C

## Comentários

- Em C99, podemos também escrever comentários de uma só linha:

```
// Isto é um comentário
```

- Começa com // e termina no final da linha
- Mais sucinto para comentários curtos
- Evita o risco de esquecer fechar o comentário

# Linguagem C

## Variáveis e atribuições

- Os programas em C efetuam computação *modificando* valores em memória
- Os locais para guardar valores são designados usando *variáveis*.

# Linguagem C

## Tipos

- As variáveis em C têm associado um *tipo*
- Tipos numéricos básicos: *int* e *float*
- Uma variável do tipo *int* pode guardar valores inteiros positivos e negativos; e.g.:
  - 0 1 -23 397
- Os valores mínimo e máximo de *int* dependem da implementação; e.g:
  - O GCC em Intel x86/x64 usa 32 bits
  - Inteiros de  $-2^{31}$  a  $+2^{31} - 1$
- Uma variável *float* guarda valores de *vírgula-flutuante* com precisão simples
- Pode representar valores fracionários:
  - 0.0253 -1.25 123.555
- Também valores de magnitudes muito grandes ou pequenas (aproximadamente entre  $10^{-38}$  e  $10^{+38}$ )
- Desvantagens:
  - Operações mais lentas do que com inteiros
  - Erro de arredondamento

# Linguagem C

## Declarações

- Até C89: todas as declarações têm de ocorrer antes das instruções

```
int main(void) {  
    /* declarações de variáveis */  
    int altura, largura;  
    float raio;  
  
    /* seguem-se instruções */  
    ...  
}
```

- Em C99: declarações e instruções podem ser misturadas (desde que a declaração ocorra antes do uso)

# Linguagem C

## Atribuição

- Podemos definir ou modificar o valor de uma variável usando uma *atribuição*

```
int altura; // declaração  
altura = 8; // atribuição
```

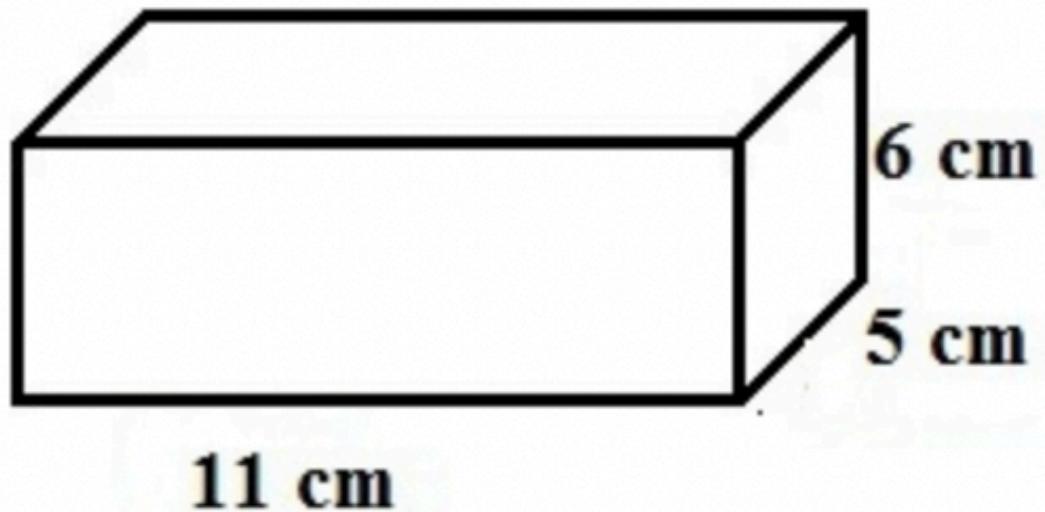
- A atribuição tem de ocorrer depois da declaração
- Neste caso: atribuímos a constante 8 à variável *altura*
- No lado direito duma atribuição podemos usar expressões e.g. constantes, variáveis e operações

```
int altura, largura, area;  
altura = 8;  
largura = 3;  
area = altura * largura;  
// area é 24
```

# Linguagem C

## Exemplo

Um programa para calcular o *volume*  $V$  de uma caixa retangular.



$$V = 11\text{cm} \times 5\text{cm} \times 6\text{cm} = 330\text{cm}^3$$

```
#include <stdio.h>

int main(void) {
    int l, w, h, v; // dimensões e volume

    l = 11; // comprimento
    w = 5; // largura
    h = 6; // altura
    v = l*w*h; // cálculo do volume

    printf("LxWxH: %d*%d*%d (cm)\n", l,w,h);
    printf("Volume: %d (cm^3)\n", v);
    return 0;
}
```

# Linguagem C

## Imprimir valores

Podemos usar a função *printf* da biblioteca *stdio.h* para imprimir valores das variáveis,

```
int alt;  
alt = 6;  
printf("Altura: %d cm\n", alt);
```

Imprime o texto:  
Altura: 6cm

*%d* é um campo que é substituído pelo valor duma variável inteira em decimal.

# Linguagem C

## Imprimir valores

Para valores *float* usamos o campo `%f`

```
Float custo;  
Custo = 123.45;  
printf("Custo: EUR %f\n", custo);
```

Resultado:

Custo: EUR 123.449997

Não é possível representar 123.45 de forma exata como *float*!

`%f` apresenta o resultado arredondado a 6 casas decimais

Para forçar a formatação com *n* casas decimais usamos `%.nf`

```
printf("Custo: EUR %.2f\n", custo);
```

Resultado:

Custo: EUR 123.45

# Linguagem C

## Imprimir valores

Podemos formatar vários valores num só *printf*

```
printf("Altura: %d cm; Custo: EUR %.2f\n", alt, custo);
```

Atenção:

- Especificar o mesmo número de campos do que argumentos
- Usar campos corretos para cada tipo (%d para inteiros, %f para float)

# Linguagem C

## Ler valores

- A função de biblioteca *scanf* é usada para ler valores de entrada-padrão (teclado)
- Tal como *printf* o 1º argumento é o formato dos dados
- Exemplo: ler um valor inteiro e guardar o resultado na variável *n*

```
int n;  
scanf("%d", &n);
```

- É **necessário** colocar o sinal & antes do nome da variável a ler (mais tarde veremos porquê)
- Para ler um *float* não necessitamos de especificar casas decimais

```
float f;  
scanf("%f", &f);
```

- Funciona com ou sem casas decimais na entrada; exemplos
  - 123
  - 123.4
  - 123.4567

# Linguagem C

## Exemplo revisto

Vamos alterar o programa de exemplo anterior para ler as dimensões da caixa

```
#include <stdio.h>

int main(void) {
    int l, w, h, v; // dimensões e volume

    printf("L=?"); scanf("%d", &l);
    printf("W=?"); scanf("%d", &w);
    printf("H=?"); scanf("%d", &h);
    v = l*w*h; // cálculo do volume

    printf("LxWxH: %d*d*d (cm)\n", l,w,h);
    printf("Volume: %d (cm^3)\n", v);
    return 0;
}
```

# Linguagem C

## Inicializações

- Variáveis em C não são inicializadas automaticamente
- Uma variável a que não atribuímos um valor diz-se **não-inicializada**:
- O resultado de usar variáveis não-inicializadas é *imprevisível*:
  - Valores diferentes em cada execução;
  - Terminar a execução com erro (*crash*)
- O compilador *gcc* pode detetar variáveis não-inicializadas usando a opção *-Wall*

# Linguagem C

## Inicializações

- Podemos inicializar variáveis diretamente na declaração:

```
int alt = 8;
```

- Também para múltiplas variáveis:

```
int alt = 8, larg = 5, comp = 11;
```

- Cada variável necessita do seu inicializador:

```
int alt, larg, comp = 11;  
// só inicializa uma variável (comp)
```

# Linguagem C

## Identificadores

- Os nomes de variáveis, funções e outras entidades são *identificadores*
- Podem conter *letras, algoritmos e sublinhado* mas devem começar por letras ou sublinhado
- Apenas letras *não acentuadas* (i.e. ASCII)

Exemplos válidos:

times10 get\_Next\_Char \_done

Exemplos inválidos:

10times get-Next-Char máximo

# Linguagem C

## Identificadores

- Maiúsculas e minúsculas são distintas; por exemplo
  - *get\_next\_char*
  - *get\_next\_Char*
  - *get\_Next\_Char*são identificadores diferentes (seria confuso usá-los num mesmo programa...).
- Não há limite definido para comprimento dos identificadores

# Linguagem C

## Palavras reservadas

Não podemos usar as seguintes *palavras reservadas* como identificadores.

<i>auto</i>	<i>enum</i>	<i>restrict</i>	<i>unsigned</i>
<i>break</i>	<i>extern</i>	<i>return</i>	<i>void</i>
<i>case</i>	<i>float</i>	<i>short</i>	<i>volatile</i>
<i>char</i>	<i>for</i>	<i>signed</i>	<i>while</i>
<i>const</i>	<i>goto</i>	<i>sizeof</i>	<i>_Bool</i>
<i>continue</i>	<i>if</i>	<i>static</i>	<i>_Complex</i>
<i>default</i>	<i>inline</i>	<i>struct</i>	<i>_Imaginary</i>
<i>do</i>	<i>int</i>	<i>switch</i>	
<i>double</i>	<i>long</i>	<i>typedef</i>	
<i>else</i>	<i>register</i>	<i>union</i>	

# Linguagem C

## Definir constantes

- É por vezes necessário usar *constants* ou *parâmetros*
- Constantes espalhadas pelo programa podem ofuscar o sentido
- Em vez disso: podemos usar diretivas `#define` para definir *macros*

```
#define INCHES_PER_METER 39.3701
/* factor de conversão:
   polegadas por cada metro */
```

- Convenção: nomes de constantes em maiúsculas

# Linguagem C

## Pré-processamento

- O pré-processador substitui as *macros* textualmente

```
inches = meters * INCHES_PER_METER;
```

- Após processamento fica:

```
inches = meters * 39.3701;
```

# Linguagem C

## Exemplo

Um programa para calcular a **área de uma circunferência**.

A área de uma circunferência de raio  $r$  é  $A = \pi r^2$  (onde  $\pi$  é a constante 3.14159...).

```
#include <stdio.h>

#define PI 3.14159

int main(void) {
    float raio, area;
    printf("Raio da circunferência?");
    scanf("%f", &raio);
    area = PI * raio * raio;
    printf("Área: %f\n", area);
    return 0;
}
```

# Linguagem C

## Estrutura de programas

Um programa em C é uma sequência de símbolos (“tokens”):

- Identificadores
- Palavras reservadas
- Operadores
- Pontuação
- Constantes
- Cadeiras de caracteres literais

# Linguagem C

## Estrutura de programas

Exemplo: a instrução

```
printf("Área: %f\n", area);
```

contém sete símbolos:

printf	identificador
(	pontuação
"Área : %f\n"	Cadeia de caracteres
,	pontuação
area	Identificador
)	Pontuação
;	Pontuação

# Linguagem C

## *Layout de programas*

- Os espaços entre símbolos não são normalmente importantes
- Podemos mesmo omitir espaços (exceto quando dessa forma dois símbolos distintos se fundem)
- Contudo, isto *dificulta a legibilidade* dos programas

```
#include <stdio.h>
#define PI 3.14159
int main(void) {float raio,area;printf(
"Raio da circunferência?");scanf("%f",&raio);
area=PI*raio*raio;printf("Área: %f\n",area);
return 0;}
```

# Linguagem C

## *Layout de programas*

Devemos usar espaços, tabulações e mudança de linha para aumentar a legibilidade do programa:

- Inserir espaços após vírgulas ou entre operadores;
- Usar tabulações para alinhar instruções;
- Usar linhas em branco para separar visualmente blocos de código;
- Inserir comentários entre linhas (ou mesmo no meio de uma linha)

# Linguagem C

## Salientar a sintaxe

- Os editores de texto têm modos especiais para linguagens de programação
- Entre outras coisas salientam automaticamente símbolos com *cores* e/ou *estilos*
- Auxiliam a leitura e escrita de programas

# Linguagem C

## Exemplo

```
#include <stdio.h>

#define PI 3.14159

int main(void) {
    float raio, area;

    printf("Raio da circunferência?");
    scanf("%f", &raio);
    area = PI * raio * raio;
        // calcular a área
    printf("Área: %f\n", area);
    return 0;
}
```

# Linguagem C

## Expressões

- Expressões são constituídas por variáveis, constantes e operadores
- A linguagem C inclui operadores para:
  - aritmética (+, -, \*, /)
  - comparações
  - operações lógicas
  - atribuição
  - pré- e pós-incremento e decremento

# Linguagem C

## Operadores aritméticos

- Operadores binários (sobre dois valores)

Adição	$a+b$
Subtração	$a-b$
Multiplicação	$a*b$
Divisão	$a/b$
Resto da divisão	$a\%b$

$+, *, e /$  permitem misturar operandos inteiros e vírgula-flutuante

Quando combinados operandos *int* e *float*, o resultado é um *float*

Ex:  $2 + 0.5$  dá  $2.5$ ;  $3.5/2$  dá  $1.75$

- Operadores unários (sobre um valor)

Menos unário	$-a$
Mais unário	$+a$

Necessitam apenas de um operando; exemplos:

$$i = + 1;$$

$$j = - i;$$

O operador + unário não tem qualquer efeito; é usado principalmente para sinalizar constantes positivas

# Linguagem C

## Divisão inteira

- Se os operandos são inteiros, o resultado de `/` é o *quociente*:
  - `3/4` dá 0
  - `10/3` dá 3
- `i%j` é o *resto* da divisão inteira de `i` por `j`
  - `10 % 3` dá 1
  - `12 % 4` dá 0

# Linguagem C

## Cuidados com / e %

- Ambos os operandos de % devem ser inteiros
- Se o lado direito de / ou % for 0, o resultado é *indefinido* (divisão por zero)
- Se o lado direito de % for negativo:
  - Em C99,  $i \% j$  tem o mesmo sinal de i
  - Em C89, depende da implementação

# Linguagem C

## Precedência de operadores

- Como interpretar  $i + j * k$ ?
  - “Somar i com o resultado de multiplicar j com k”
  - “Somar i com j e multiplicar o resultado por k”
- Podemos usar parêntesis para tornar o sentido não ambíguo:
  - $i + (j * k)$  ou  $(i + j) * k$
- Na ausência de parêntesis: a *precedência* entre operadores desambigua o significado
  1. Primeiro são avaliados + e - unários
  2. Depois \*, /, %
  3. Por fim + e - binários

$i + j * k$	$i + (j * k)$
$-i * -j$	$(-i) * (-j)$
$+i + j / k$	$(+i) + (j/k)$

# Linguagem C

## Associatividade de operadores

- A *associatividade* define como interpretar dois ou mais operadores com igual precedência
- Os operadores binários (+, -, \*, /, %) associam à esquerda
- Os operadores unários associam à direita

$i - j - k$	$(i - j) - k$
$i * j / k$	$(i * j) / k$
$- +i$	$- (+i)$

# Linguagem C

## Atribuição simples

- A atribuição  $var = expr$  calcula o valor de  $expr$  e copia-o para a variável  $var$
- O lado direito pode ser uma constante, uma variável ou uma expressão complexa

```
i = 5;          /* valor de i: 5 */
j = i;          /* valor de j: 5 */
k = 10 * i + j; /* valor de k: 55 */
```

- Se a variável e expressão não forem do mesmo tipo dá-se uma **conversão implícita** de tipos.

```
int i;
float f;

i = 72.99;    // valor de i: 72
f = 136;      // valor de f: 136.0
```

# Linguagem C

## Atribuições são expressões

- Na linguagem C uma atribuição é também uma expressão
- O resultado de  $var = expr$  é o valor que foi atribuído
- Exemplo:

```
int i, j, k;  
i = 1;  
k = 1 + (j = i);  
// j: 1, k: 2
```

Cuidado: usar atribuições no meio de expressões pode dificultar a compreensão de programas

# Linguagem C

## Atribuições em sequência

- Podemos atribuir o mesmo valor a várias variáveis:

$$i = j = k = 0;$$

- Como a atribuição associa à direita, isto é equivalente a:

$$i = (j = (k = 0));$$

# Linguagem C

## Valores esquerdos (*Lvalues*)

- O lado esquerdo de uma atribuição corresponde a um *local* na memória (“lvalue”)
- Uma variável é um lvalue, mas as constantes ou expressões compostas não são
- O compilador assina erro numa atribuição com lado esquerdo inválido: “*invalid lvalue in assignment*”

```
i = 12;      // OK: i é um lvalue
12 = i;      // erro: 12 não é um lvalue
1+j = 12;    // erro: 1+j não é um lvalue
```

# Linguagem C

## Atribuição composta

É frequente atribuir a uma variável um novo valor que depende do seu valor atual.

Exemplo:

$$i = i + 2;$$

Nestes casos podemos usar uma **atribuição composta**:

$$i += 2;$$

# Linguagem C

## Operadores de atribuição composta

$v += e$

Adicionar  $e$  a  $v$ , guardando o resultado em  $v$

$v -= e$

Subtrair  $e$  a  $v$ , guardando o resultado em  $v$

$v *= e$

Multiplicar  $e$  por  $v$ , guardando o resultado em  $v$

$v /= e$

Dividir  $v$  por  $e$ , guardando o resultado em  $v$

$v \% = e$

Calcular o resto da divisão de  $v$  por  $e$ , guardando o resultado em  $v$

# Linguagem C

## Incremento e decremento

É frequente somar ou subtrair uma variável inteira de uma unidade.

$i = i + 1;$

$j = j - 1;$

Também aqui podemos usar uma atribuição composta:

$i += 1;$

$j -= 1;$

Em alternativa, podemos usar operadores de incremento ou decremento

Podem ser usados de forma **prefixa** (`++i` ou `--i`) ou **pósfixa** (`i++` ou `i--`)

```
++i;      // equivalente a i = i + 1
--j;      // equivalente a j = j - 1
```

# Linguagem C

## Pré- e pós-incremento

`++i` modifica a variável `i` (incremento de uma unidade) e dá o valor resultante

```
i = 1;  
printf("%d\n", ++i); // imprime 2  
printf("%d\n", i); // imprime 2
```

`i++` dá o valor atual de `i` e depois modifica a variável `i` (incremento de 1 unidade)

```
i = 1;  
printf("%d\n", i++); // imprime 1  
printf("%d\n", i); // imprime 2
```

`++i` incrementa imediatamente, enquanto `i++` incrementa mais tarde

O *standard* de C não especifica exatamente quando acontece o pós-incremento, mas é seguro assumir que `i` será incrementado antes da próxima instrução

# Linguagem C

## Pré- e pós-decremento

O operador de decremento comporta-se de forma análoga ao de incremento:

```
i = 1;  
printf("%d\n", --i); // imprime 0  
printf("%d\n", i); // imprime 0  
i = 1;  
printf("%d\n", i--); // imprime 1  
printf("%d\n", i); // imprime 0
```

# Linguagem C

## Operadores relacionais

- Operadores binários para comparações entre expressões numéricas:

<b>Menor que</b>	<
<b>Maior que</b>	>
<b>Menor ou igual</b>	<=
<b>Maior ou igual</b>	>=
<b>igual</b>	==
<b>diferente</b>	!=

- O resultado é **inteiro**: 0 se a condição for falsa, 1 se a condição for verdadeira

# Linguagem C

## Precedência e associatividade

- Operadores relacionais têm *precedência inferior* do que os aritméticos
  - $i + j < k - 1$  equivale a  $(i + j) < (k - 1)$
- Operadores relacionais associam à esquerda
- A expressão  $i < j < k$  é válida mas **não testa** se j está entre i e k:
  - $<$  associa à esquerda, logo a expressão acima é equivalente a  $(i < j) < k$
- A expressão correta usa a **conjunção** de duas condições:  
 $i < j \& j < k$

# Linguagem C

## Comparações de igualdade

- Dois operadores: `==` (igual) e `!=` (diferente)
- Resultado é 0 ou 1 (falso ou verdadeiro)
-  não confundir atribuições com comparações:  
`i = j` modifica o lado esquerdo; resultado é o valor atribuído  
`i == j` compara os lados esquerdo e direito; resultado é 0 ou 1

# Linguagem C

## Instrução if

- A instrução *if* executa condicionalmente uma instrução conforme o resultado de uma expressão
- Forma mais simples:  
*if (expressão) instrução*
- Calcula o valor da expressão; se o resultado for diferente de zero, então executa a instrução
- Em qualquer caso, continua executando as instruções seguintes

```
if ( line_num == MAX ) line_num = 0;  
// continuação do programa  
...
```

# Linguagem C

## Instrução if

- Se quisermos executar condicionalmente mais do que uma instrução devemos agrupá-las num bloco:

{ *instruções*}

- As chavetas forçam o compilador a tratar um bloco de instruções como uma só.
- Cada instrução dentro do bloco termina com ponto-e-vírgula
- Não acrescentamos ponto-e-vírgula depois de fechar chavetas

```
if (line_num == MAX){  
    line_num = 0;  
    page_num ++;  
}
```

# Linguagem C

## Alternativa `else`

- A instrução `if` pode incluir uma alternativa `else`:

*if (expressão) instrução else instrução*

- A instrução a seguir ao `else` é executada se a expressão tiver valor 0

`if (i > j) max = i; else max = j;`

# Linguagem C

## Alternativa `else`

- Podemos incluir instruções *if* dentro de outros ifs
- Nesses casos será útil usar *identação* para explicitar a estrutura

```
if (i > j)
    if (i > k)
        max = i;
    else
        max = k;
else
    if (j > k)
        max = j;
    else
        max = k;
```

# Linguagem C

## Alternativa `else`

- Podemos também adicionar chavetas para auxiliar a compreensão:

```
if (i > j) {  
    if (i > k)  
        max = i;  
    else  
        max = k;  
} else {  
    if (j > k)  
        max = j;  
    else  
        max = k;  
}
```

# Linguagem C

## Alternativa else

- Há quem prefira colocar sempre chavetas:

```
if (i > j) {  
    if (i > k){  
        max = i;  
    } else {  
        max = k;  
    }  
} else {  
    if (j > k){  
        max = j;  
    } else {  
        max = k;  
    }  
}
```

Ao colocarmos sempre chavetas:

- Torna-se mais fácil acrescentar instruções dentro do if ou do else;
- Evitamos erros resultantes de esquecer as chavetas ao acrescentar instruções

# Linguagem C

## If em “cascata”

- Para testar condições em sequência escrevemos múltiplas instruções *if* em “cascata”.

```
if (n < 0)
    printf("n negativo\n");
else if (n == 0)
    printf("n zero\n");
else
    printf("n positivo\n");
```

# Linguagem C

## Exemplo: calcular comissões

- Quando um corretor vende ações cobra uma comissão cujo valor depende do montante transacionado
- Vamos escrever um programa para calcular a comissão conforme a tabela seguinte
- A comissão mínima a cobrar deve ser €39

Montante	Comissão
Até €2500	€30 + 1.7%
€2500-€6250	€56 + 0.66%
€6250-€20K	€76 + 0.34%
€20K-€50K	€100 + 0.22%
€50K-€500K	€155 + 0.11%
Acima de €500K	€255 + 0.09%

# Linguagem C

## Exemplo: calcular comissões

- O programa *corretor.c* lê o valor da transação, calcula a comissão e imprime-a:

```
Introduza o valor: EUR 30000
```

```
Comissão: EUR 166.00
```

- O núcleo do programa é uma sequência de *if* em cascata para determinar em que intervalo está o valor
- No final incluímos uma condição extra para garantir que cobramos sempre o valor mínimo

# Linguagem C

## Exemplo: calcular comissões

```
#include <stdio.h>

int main(void)
{
    float valor, comissao;

    printf("Introduza o valor: EUR ");
    scanf("%f", &valor);

    if(valor < 2500.0)
        comissao = 30.0 + 0.017 * valor;
    else if(valor < 6250.0)
        comissao = 56.0 + 0.0066 * valor;
    else if(valor < 20e3)
        comissao = 76.0 + 0.0034 * valor;
    else if(valor < 50e3)
        comissao = 100.0 + 0.0022 * valor;
    else if(valor < 500e3)
        comissao = 155.0 + 0.0011 * valor;
    else
        comissao = 255.0 + 0.0009 * valor;

    if (comissao < 39.0)
        comissao = 39.0;

    printf("Comissão: EUR %.2f\n", comissao);
}
```

# Linguagem C

## Operadores lógicos

- Podemos construir condições complexas partindo de outras mais simples usando **operadores lógicos**.

Conjunção	<code>&amp;&amp;</code>
Disjunção	<code>  </code>
Negação	<code>!</code>

- Os operadores lógicos são frequentemente usados para combinar comparações
- O operador `!` é unário enquanto `&&` e `||` são binários
- Operam sobre inteiros
- Qualquer valor diferente de 0 é considerado verdadeiro;
- O resultado de um operador lógico é 0 ou 1

# Linguagem C

## Cuidados com o if

- Um erro comum é trocar == (igualdade) por = (atribuição)  
`if (i == 0)...` testa se i é igual a 0  
  
`if(i = 0)...` atribui 0 a i e depois teste se o resultado é diferente de 0 (o que é sempre falso)
- Recomendação: o *gcc* avisa possíveis erros deste tipo compilando com a opção `-Wall`

# Linguagem C

## Cuidados com o else

- Ao colocarmos *if* dentro de outro, temos de ter cuidado de “casar” corretamente os *else*:

```
if (y != 0)
    if (x != 0)
        result = x / y;
else
    printf("erro: y igual a 0\n");
```

- A indentação sugere que o *else* associa ao *if* mais exterior
- Mas a regra em C é que o *else* associa ao *if* mais próximo (o interior)

- Corretamente indentado:

```
if (y != 0)
    if (x != 0)
        result = x / y;
else
    printf("erro: y igual a 0\n");
```

- Para associar o *else* ao *if* exterior temos de delimitar o *if* interior usando chavetas:

```
if (y != 0) {
    if (x != 0)
        result = x / y;
} else
    printf("erro: y igual a 0\n");
```

# Linguagem C

## Cuidados com o else

- A indentação sugere que o `else` associa ao *if* mais exterior
- Mas a regra em C é que o `else` associa ao *if* mais próximo (o interior)
- Corretamente indentado:

```
if (y != 0)
    if (x != 0)
        result = x / y;
else
    printf("erro: y igual a 0\n");
```

# Linguagem C

## Funções

- Uma função agrupa uma sequência de instruções com nome
- Cada função pode *receber argumentos e retornar um valor*
- Cada função é um sub-programa com as suas declarações e instruções próprias

# Linguagem C

## Funções - Objetivo

- Dividir o program em **componentes separadas**
  - Cada função tem um objetivo bem identificado
  - Argumentos e resultados esperados bem definidos
- Podem ser desenvolvidas e estudadas independentemente
- Podem ser testadas separadamente
- Podem ser re-utilizadas em diferentes programas

# Linguagem C

## Funções - Exemplo

- Uma função que calcular a média aritmética de dois valores:

```
float media(float a, float b){  
    float m = (a + b) / 2.0;  
    return m;  
}
```

- O identificador da função é *media*
- O tipo *float* antes do identificador indica o *tipo do resultado*
- Os *parâmetros* *a* e *b* são os dois valores do tipo *float*, que devem ser fornecidos para executar a função

# Linguagem C

## Funções - Corpo

- O corpo da função está delimitado entre chavetas:

```
{  
    float m = (a + b) / 2.0;  
    return m;  
}
```

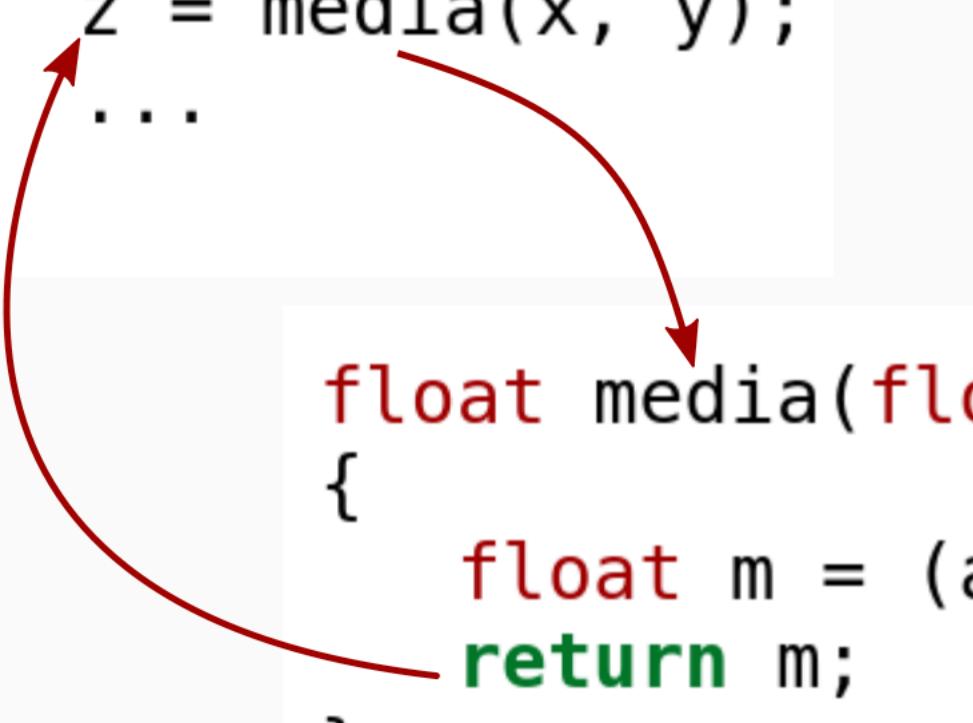
- Calcula o valor de média (usando uma variável auxiliar *m*)
- A instrução *return* termina a função e devolve o resultado ao contexto onde a função foi chamada

# Linguagem C

## Funções - Invocação

- Uma expressão: *identificador(arg1, arg2, ...)*

```
int main(void) {  
    ...  
    z = media(x, y);  
}  
  
float media(float a, float b)  
{  
    float m = (a + b) / 2.0;  
    return m;  
}
```



The diagram illustrates the flow of control during a function call. A red curved arrow originates from the assignment statement `z = media(x, y);` in the `main` function and points to the beginning of the `media` function definition. This visualizes how the program's execution is temporarily transferred to the `media` function to evaluate its expression, before returning to the `main` function to continue with the assignment.

# Linguagem C

## Funções - fluxo de execução

- A execução do programa começa pelo *main*
- *main* pode chamar outra função e assim sucessivamente
- Apenas são executadas as funções chamadas pelo *main* (direta ou indiretamente)

`z = media(x*0.5, y+1);`

- Podemos usar o resultado imediatamente em vez de o guardar numa variável

`printf("%f\n", media(x*0.5,y+1));`

# Linguagem C

## Funções - exemplo

- Vamos escrever um programa que lê três números e calcula as médias dois a dois

```
#include <stdio.h>

float media(float a, float b) {
    float m = (a + b)/2;
    return m;
}

int main(void) {
    float x, y, z;
    printf("Introduza 3 números: ");
    scanf("%f %f %f", &x, &y, &z);
    printf("Médias\n");
    printf("%f e %f: %f\n", x, y, media(x,y));
    printf("%f e %f: %f\n", y, z, media(y,z));
    printf("%f e %f: %f\n", x, z, media(x,z));
    return 0;
}
```

Introduza 3 números: 3.5 9.6 10.2  
Médias  
3.5 e 9.6: 6.55  
9.6 e 10.2: 9.9  
3.5 e 10.2: 6.85

# Linguagem C

## Funções - declarações e definições

- Se colocarmos a definição da função antes do seu uso não temos de declarar mais nada.
- Mas se usarmos a função antes da definição devemos colocar uma declaração de protótipo

```
float media(float, float);
```

- Em C89/90: se usarmos uma função sem a declararmos o compilador assume que retorna *int*.
- Em C99: usar uma função antes da definição ou declaração resulta num erro de compilação

# Linguagem C

## Funções - declarações e definições - exemplo

```
float media(float, float); // protótipo

int main(void) {
    ...
    printf(..., x, y, media(x,y));    // uso
    ...
    return 0;
}

float media(float a, float b) { // definição
    m = (a + b)/2.0;
    return m;
}
```

# Linguagem C

## Funções sem valor de retorno

- Por vezes podemos querer definir funções que não retornam um valor
- Executamo-las apenas pelos seus *efeitos colaterais*
  - e.g. imprimir mensagens na saída-padrão
- Nesse caso o tipo do resultado é *void*
- Não necessitamos de *return*
- Não usamos o resultado

# Linguagem C

## Funções sem valor de retorno - exemplo

```
void print_time(int n)
{
    printf("T minus %d and counting\n", n);
}

int main(void) {
    print_time(3);
    print_time(2);
    print_time(1);
    return 0;
}
```

# Linguagem C

## Instrução *return*

- Uma função com tipo diferente de *void* deve usar a instrução *return* para especificar o resultado
- A forma geral é:

**return** expressão;

- Não são necessários parêntesis à volta da expressão
- Por vezes a expressão é uma constante ou variável, mas pode ser uma expressão complexa:

**return** 0;

**return** n;

**return** (x + y) / 2.0;

# Linguagem C

## Instrução *return*

- Podemos usar *return* para terminar a execução da função a meio do corpo.

```
int max(int a, int b)
{
    if(a >= b)
        return a; // termina imediatamente

    // se a execução chegar a este ponto,
    // então a < b; logo o máximo é b
    return b;
}
```

- Também podemos usar *return* em funções que não retornam resultados (void)
- Nesse caso *return* serve apenas para terminar a execução da função
- Omitimos a expressão

```
void print_time(int n)
{
    if (n < 0)
        return; // terminar imediatamente
    printf("T minus %d and counting\n", n);
}
```

# Linguagem C

## Quando usar múltiplos *return*

- Usar múltiplos *return* pode dificultar a compreensão do fluxo de execução
- Recomendação: usar apenas para terminar uma função em casos especiais (ex: de erro)

# Linguagem C

## Passagem de argumentos

- Os argumentos das funções em C são *passados por valor*
  - cada expressão é avaliada e o seu valor é copiado para um parâmetro local à função
- Logo, os parâmetros de funções comportam-se como variáveis *temporárias*
  - Modificações dos argumentos **não são visíveis** depois do retorno da função

# Linguagem C

## Passagem de argumentos

```
// máximo de 2 valores
// NB: modifica o primeiro argumento
int max(int a, int b) {
    if (b > a)
        a = b;
    return a;
}

int main(void) {
    int x = 1, y = 2;
    printf("%d\n", max(x,y)); // imprime 2
    printf("%d %d\n", x, y); // imprime 1, 2
    return 0;
}
```

```
// Tenta trocar os valores de a, b;
// não funciona porque a,b são temporários
void trocar(int a, int b) {
    int t;
    t = a;
    a = b;
    b = t;
}

int main(void) {
    int x = 1, y= 2;
    trocar(x, y);
    printf("%d %d\n", x, y); // imprime 1, 2
    return 0;
}
```

# Linguagem C

## Decomposição funcional

- As funções permitem decompor problemas em partes mais simples
- Objetivo: combinar as partes para resolver o problema original
  - Analogia: peças de Lego
  - Esta metodologia de “dividir para conquistar” permite construir programas simultaneamente elegantes e eficientes

# Linguagem C

## Exemplo: imprimir algarismos

- Escrever um programa que imprime os algarismos das dezenas e unidades de um inteiro até 99
- Exemplos de execução:

12

um dois

78

sete oito

7

zero sete

# Linguagem C

## Sub-problema

- Dado o valor de um algarismo de 0-9, imprimir o texto correspondente em português:

0	“Zero”
1	“Um”
2	“Dois”
3	“Três”
4	“Quatro”
5	“Cinco”
6	“Seis”
7	“Sete”
8	“Oito”
9	“Nove”

# Linguagem C

## Função auxiliar

- Vamos definir uma função auxiliar:

```
void imprime_algarismo( int a);
```

- O argumento é um inteiro ( o valor do algarismo)
- A função imprime o texto e não retorna um valor útil (resultado *void*)
- A definição da função é longa mas simples: uma sequência de condições *if* em cascata

# Linguagem C

## Função auxiliar

```
void imprime_algarismo(int a) {
    if (a == 0)
        printf("zero");
    else if (a == 1)
        printf("um");
    else if (a == 2)
        printf("dois");
    else if (a == 3)
        printf("três");
    else if (a == 4)
        printf("quatro");
    else if (a == 5)
        printf("cinco");
    else if (a == 6)
        printf("seis");
    else if (a == 7)
        printf("sete");
    else if (a == 8)
        printf("oito");
    else if(a == 9)
        printf("nove");
    else
        printf("algarismo inválido!");
}
```

# Linguagem C

## Solução do problema original

- Usar a função auxiliar duas vezes:
  - Imprimir o algarismo das dezenas
  - Imprimir o algarismo das unidades
- Podemos obter estes algoritmos usando *divisões sucessivas por 10*
- Se  $n$  é um inteiro entre 0 e 99:
  - $n \% 10$  são as *unidades*;
  - $n / 10$  remove unidades, restam as *dezenas*

# Linguagem C

## Função principal

```
int main(void) {
    int n, d, u;
    scanf("%d", &n);
    if (!(n>=0 && n<=99)) {
        printf("número inválido\n");
        return -1; // erro
    }
    u = n % 10; // unidades (0..9)
    d = n / 10; // dezenas (0..9)
    imprime_algarismo(d); printf(" ");
    imprime_algarismo(u); printf("\n");
    return 0; // OK
}
```

## Observações

- O programa principal fica bastante simples - apenas calcula os algarismos e executa a função
- É fácil modificar o programa
  - Exemplo: imprimir também o algarismo das centenas
- Também podemos melhorar a implementação da função
  - Nas aulas seguintes iremos ver alternativas às “cascatas” de ifs

## Conclusão

- A decomposição em funções permitiu:
  - Escrever programas em partes mais simples
  - Considerar parte do problema de cada vez
  - Combinar as soluções para resolver o problema original

# Linguagem C

## Ciclos

- Um *ciclo* é uma instrução que executa várias vezes outras instruções (o corpo do ciclo)
- Os ciclos em C são controlados por uma *expressão*
- A expressão é avaliada a cada *iteração*
  - Se o seu valor for zero o ciclo termina
  - Se for diferente de zero, o ciclo continua

# Linguagem C

## Instruções de ciclo

### **while**

É usado para ciclos em que a expressão é testada antes de executar o corpo do ciclo

### **do...while**

É usado para ciclos em que a expressão é testada depois de executar o corpo do ciclo

### **for**

É uma forma conveniente para ciclos com uma variável de controlo

# Linguagem C

## Instrução while

**while** ( expressão ) instrução

- A expressão controla a terminação do ciclo
- A *instrução* é o corpo do ciclo

Execução:

1. Primeiro avalia a expressão.
2. Se for zero, o ciclo termina imediatamente;  
se for diferente de zero, executa *instrução* e repete 1.

O corpo pode ser um *bloco de instruções* em vez de uma só:

```
i = 1;
while (i < 10) {
    printf("%d\n", i);
    i = i * 2;
}
```

# Linguagem C

## Instrução while - terminação

- O ciclo *while* termina quando o valor da expressão for 0 (falso)
  - E.g., se a expressão for  $i < 10$  então o ciclo termina quando  $i \geq 10$
- O corpo pode não chegar a executar (porque a expressão de controlo é testada primeiro).
- Se a expressão de controlo for sempre diferente de zero o *ciclo não termina* (excepto se usarmos instruções especiais para sair do ciclo - mais tarde)

```
while (1) {  
    ... /* ciclo infinito */  
}
```

# Linguagem C

## Instrução while - exemplo 1

- Um programa para imprimir uma tabela de quadrados
- O utilizador introduz o limite superior

```
Limite superior: 5
1      1
2      4
3      9
4      16
5      25
```

```
#include <stdio.h>

int main(void) {
    int i, n;

    printf("Limite superior: ");
    scanf("%d", &n);
    i = 1;
    while (i <= n) {
        printf("%d\t%d\n", i, i*i);
        i++;
    }
    return 0;
}
```

# Linguagem C

## Instrução while - exemplo 2

- Um programa para somar uma sequência de números
- O tamanho da sequência não é conhecido antecipadamente
- Ideia:
  - Ler cada valor dentro do ciclo
  - Acumular o total numa variável auxiliar
  - Terminar quando lermos um valor especial (zero)

```
/*
    somar.c
    Somar uma sequência de números
*/
#include <stdio.h>

int main(void) {
    int n, soma = 0;
    printf("Introduza valores; 0 termina.\n");
    scanf("%d", &n);          // primeiro valor
    while (n != 0) {          // enquanto não terminou
        soma += n;           // acumular
        scanf("%d", &n);     // ler próximo valor
    }
    printf("A soma é: %d\n", soma);
    return 0;
}
```

# Linguagem C

## Instrução do...while

```
do instrução while ( expressão );
```

Execução:

1. Primeiro executa a *instrução*.
2. Depois avalia a expressão.
3. Se for zero, o ciclo termina;  
se for diferente de zero, repete o passo 1.

# Linguagem C

## Instrução do...while - exemplo

```
/*
    somar2.c
    Somar uma sequência de números (alternativa)
*/
#include <stdio.h>

int main(void) {
    int n, soma = 0;
    printf("Introduza valores; 0 termina.\n");
    do {
        scanf("%d", &n);    // próximo valor
        soma += n;          // acumular
    } while (n != 0); // enquanto não terminou
    printf("A soma é: %d\n", soma);
    return 0;
}
```

- Como a condição é sempre testada depois da execução não necessitamos de ler o primeiro valor fora do ciclo
- Somar zero não altera o resultado: podemos sempre acumular o valor lido

# Linguagem C

## Instrução do...while - exemplo 3

- Calcula o número de algarismos de um inteiro positivo:

```
Introduza um inteiro positivo: 5633
4 algarismo(s)
```

- Vamos usar um ciclo para fazer divisões inteiras por 10
- Terminamos quando chegar a zero
- O número de iterações efetuadas dá-nos a contagem de algarismos
- O ciclo **do** é mais conveniente do que o **while** porque qualquer número positivo tem pelo menos um algarismo.

```
/* algarismos.c
   Contar algarismos de um inteiro positivo
*/
#include <stdio.h>

int main(void) {
    int digits = 0, n;
    printf("Inteiro positivo: ");
    scanf("%d", &n);
    do {
        n /= 10;      // quociente divisão por
10
        digits++;    // mais um algarismo
    } while (n > 0);
    printf("%d algarismo(s)\n", digits);
    return 0;
}
```

# Linguagem C

## Instrução for

`for ( expr1; expr2; expr3 ) instrução`

- *expr1* é a inicialização
- *expr2* é a condição de repetição
- *expr3* é a atualização após cada iteração
- *instrução* é o corpo do ciclo
- Exemplo:

```
for (i = 0; i < 10; i++)
    printf("%d\n", i);
```

- O *for* é conveniente para ciclos que necessitam de contar de um valor inicial até um valor final.
- Poderíamos usar o *while* em vez do *for*, mas o *for* é mais claro

```
for (i = 0; i < 10; i++)
    printf("%d\n", i);
```

É equivalente a:

```
i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

# Linguagem C

## Instrução for

Formas comuns de *for*:

```
// contagem ascendente de 0 até n-1
for(i = 0; i < n; i++) ...
```

```
// contagem ascendente de 1 até n
for(i = 1; i <= n; i++) ...
```

```
// contagem descendente de n-1 até 0
for(i = n-1; i >= 0; i--) ...
```

```
// contagem descendente de n até 1
for(i = n; i > 0; i--) ...
```

- Erros comuns no *for*:
  - Trocar a ordem das comparações
  - Contagens ascendentes devem usar < ou <=
  - Contagens descendentes devem usar o > ou >=
  - Usar == em vez de <, <=, >, >=
  - Podemos “saltar” a terminação
  - “Errar por um” a condição de terminação
    - e.g. usar i<n em vez de i<=n

# Linguagem C

## Instrução break

- Usualmente um ciclo termina apenas quando a condição é testada
  - Antes de uma iteração de **while** ou **for**
  - Depois de uma iteração **do... while**
- Também podemos usar a instrução **break** para terminar um ciclo em qualquer momento

- Exemplo: procurar o primeiro divisor próprio de um número n (um divisor entre 2 e n-1)

```
int i, n;  
... /* obter valor de `n' */  
for (i = 2; i < n; i++) {  
    if (n%i == 0) break;  
}
```

- Este ciclo pode terminar de duas formas:
  - Se esgotou os possíveis divisores ( $i \geq 0$ )
  - Se encontrou um divisor ( $i < n$ )

# Linguagem C

## Instrução continue

- A instrução **continue** transfere a execução para o ponto imediatamente *antes* do fim do corpo
- Enquanto o **break** termina o ciclo, o **continue** continua o ciclo
- Exemplo: ler e acumular 10 inteiros não-negativos

```
int i = 0, n, soma = 0;
while (i < 10) {
    scanf("%d", &n);
    if (n < 0)
        continue;
    soma += n;
    i++;
    /* continue salta para aqui */
}
```

# Linguagem C

## Caracteres

- O tipo *char* é usado para representar caracteres
- Os caracteres são representados pelo seu *código numérico*
- O padrão da linguagem não define a codificação
- O código mais comum é ASCII (American Standard Cod for Information Interchange)
- O ASCII contém 128 símbolos e apenas as letras do alfabeto latino sem acentos
- O UTF-8 é uma extensão do ASCII que suporta outros alfabetos e acentos

# Linguagem C

## Processar caracteres

- Ler e escrever:
  - Podemos dar o *scanf* e o *printf* para ler e escrever caracteres

```
scanf("%c", &ch); // ler um carater  
printf("%c", ch); // escrever um carater
```

- Mas é mais comum usar as funções mais básicas:

```
ch = getchar(); // ler um carater  
putchar(ch); // escrever um carater
```

# Linguagem C

## Instrução switch

- Podemos usar uma sequência de *if/else* para escolher entre múltiplos casos:

```
if(nota == 4)
    printf("Excelente");
else if(nota == 3)
    printf("Bom");
else if(nota == 2)
    printf("Razoável");
else if(nota == 1)
    printf("Fraco");
else
    printf("Nota inválida");
```

- Em alternativa, podemos usar uma instrução *switch*:

```
switch(nota) {
    case 4: printf("Excelente");
              break;
    case 3: printf("Bom");
              break;
    case 2: printf("Razoável");
              break;
    case 1: printf("Fraco");
              break;
    default: printf("Nota inválida");
              break;
}
```

# Linguagem C

## Instrução switch

- A instrução **switch** pode ser mais legível que a sequência *if/else if* equivalente
- Poderá também executar mais eficientemente
- Forma geral:

```
switch ( expressão ) {  
    case constante1: instruções  
    ...  
    case constanteN: instruções  
    default: instruções  
}
```

- A expressão scrutinada deve ser do tipo inteiro
  - E.g: *int, long, char*
- Caracteres também são inteiros em C.
- Não pode haver vírgula flutuante ou texto
- Após cada *case* segue-se uma sequência de instruções
- A última instrução de cada sequência é normalmente *break*
- Não são necessárias chavetas
- Não podemos repetir casos
- A ordem dos *case* não importa se colocarmos sempre o *break*

# Linguagem C

## Instrução switch

- Podemos agrupar múltiplos case com uma só sequência de instruções

```
switch(nota) {  
    case 4:  
    case 3:  
    case 2: printf("Aprovado");  
              break;  
    case 1: printf("Reprovado");  
              break;  
    default: printf("Nota  
inválida");  
              break;  
}
```

- **Efeito do break:**

- O *break* faz com que a execução continue na instrução seguinte ao *switch*
- Se omitirmos o *break*, a execução continua no caso seguinte

# Linguagem C

## Arrays

- Um array é um agregado de valores todos dum mesmo tipo
- Os valores (elementos) podem ser acedidos por *índices*.
- A forma mais simples de arrays tem apenas uma dimensão
- Exemplo: declarar um array com 4 elementos

```
int a[4];
```

a[0]	a[1]	a[2]	a[3]
------	------	------	------

- Todos os elementos são do mesmo tipo (int)
- O índice do primeiro elemento é 0
- O índice do último elemento é o número de elementos menos 1

# Linguagem C

## Arrays - Declaração

- O tamanho dum array deve ser uma constante inteira positiva:

```
int a[4], b[5];
```

- Podemos usar *macros* para melhor a legibilidade e facilitar modificações de programas:

```
#define SIZE 10  
...  
int a[SIZE], b[SIZE+1];
```

# Linguagem C

## Arrays - Indexação

- Acedemos a um elemento com  $a[i]$ 
  - $i$  é o índice (inteiro)
- $a[i]$  pode ser usado tal como uma variável simples:
  - Numa expressão;
  - No lado esquerdo duma atribuição;
  - Com operadores de incremento, etc.

```
a[i] = 1;  
printf("%d", a[i]);  
++ a[i];
```

# Linguagem C

## Arrays - Indexação

- Os ciclos **for** são convenientes para processar elementos de arrays.
- Exemplos com um array de tamanho **N**:

```
for(i = 0; i < N; i++) // Colocar zeros  
    a[i] = 0;
```

```
for(i = 0; i < N; i++) // Ler valores  
    scanf("%d", &a[i]);
```

```
soma = 0;  
for(i = 0; i < N; i++) // Somar elementos  
    soma += a[i];
```

# Linguagem C

## Índices válidos

- A linguagem C **não** faz verificação dos índices de arrays
- Isto permite que a implementação seja o mais eficiente possível
- Mas se o programa aceder a índices inválidos, o comportamento é indefinido:
  - Pode terminar abruptamente
  - Pode não terminar
  - Pode dar resultados errados
- Erro comum: aceder ao elemento  $a[N]$  dum array com tamanho N

# Linguagem C

## Onde está o erro?

```
int main(void) {
    int a[10], i;

    for(i = 0; i <= 10; i++)
        a[i] = 0;
    ...
}
```

- Este programa modifica a[0], a[1],..., a[10]
- Mas os elementos são a[0], a[1],..., a[9]
- Com alguns compiladores isto pode fazer com que o ciclo não termine
-

# Linguagem C

## Arrays - Inicialização

- Tal como as variáveis simples, podemos *inicializar*, os arrays na declaração
- A inicialização mais comum é uma lista de valores (entre chaves e separados por vírgulas)

```
int a[5] = {1, 2, 3, 4, 5};
```

- Se a lista de valores for mais curta do que o array, os restantes elementos são preenchidos com zeros

```
int a[10] = {1, 2, 3, 4, 5};  
/* valores iniciais são  
{1, 2, 3, 4, 5, 0, 0, 0, 0, 0} */
```

- Isto é útil para inicializar todos os elementos com zero

```
int a[10] = {0};  
/* valores iniciais são  
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

- Podemos omitir o tamanho da variável; nesse caso o compilador infere o tamanho pela lista de valores

```
int a[] = {1, 2, 3, 4, 5};  
// equivalente a declarar a[5]
```

# Linguagem C

## Funções e arrays

- Podemos passar arrays como argumentos duma função
- Uma função **não** pode retornar uma variável indexada
  - Mais tarde vamos ver que podemos retornar um apontador
- Mas a função pode modificar o conteúdo de um array passado como variável.

# Linguagem C

## Argumentos de funções

- Quando uma função tem variável indexada como argumento não necessitamos de especificar o tamanho:

```
int fun(int a[]) {  
    // tamanho de a[] não especificado  
    ...  
}
```

- Contudo, desta forma a função não tem forma de saber o tamanho com que foi declarado o array
- Se necessitarmos de saber o tamanho dentro da função, temos de o **passar explicitamente**.

# Linguagem C

## Argumentos de funções

```
int somar_vec(int a[], unsigned size) {
    int i, soma = 0;
    for (i = 0; i < size; i++)
        soma += a[i];
    return soma;
}
```

Para usar esta função devemos passar dois argumentos:

- O array
- e o seu tamanho

```
int v[100];
...
s = somar_vec(v, 100);
```

Também podemos usar esta função para somar apenas um segmento inicial:

```
int v[100];
...
s = somar_vec(v, 50);
```

Desta forma podemos:

- Declarar a variável com um **tamanho máximo**
- Operar com um **tamanho variável**(desde que não ultrapasse o máximo)

# Linguagem C

## Modificar arrays

- Se uma função modificar um array isso reflete-se no argumento passado
- ```
void colocar_zeros(int a[], unsigned size) {
    int i;
    for(i = 0; i < size; i++) {
        a[i] = 0;
    }
}
```
- Executar esta função modifica o array passado como argumento
  - Isto parece contraditório com o caso das variáveis simples
  - Quando estudarmos apontadores veremos a razão deste comportamento

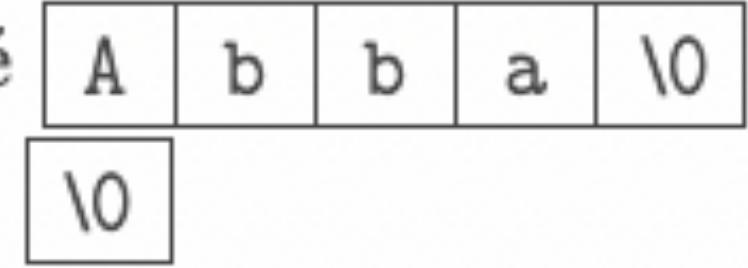
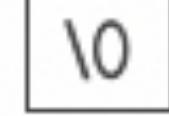
# Linguagem C

## Arrays de caracteres

- Podemos usar arrays de caracteres para representar strings.
- Tal como outros arrays devemos declarar com um tamanho:  
`char text[100]; // 100 caracteres máximo`
- Para facilitar modificações podemos usar uma *macro*:  
`#define MAX_SIZE 100`  
...  
`char text[MAX_SIZE];`

# Linguagem C

## Strings

- As strings em C são terminadas com um caracter de código zero (representado como '\0')
  - a cadeia "Abba" é 
  - a cadeia vazia "" é 
- O caracter '\0' é um *terminator* que marca o final da string
  - Não faz parte do texto
  - O comprimento da string é determinado pela posição do terminador

# Linguagem C

## Inicialização de strings

- Inicializar usando constantes de caracteres:

```
char text[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

- Podemos também inicializar com os caracteres entre aspas:

```
char text[6] = "Hello";
```

- Desta forma, o terminador '\0' é automaticamente adicionado

