Métodos de Programação I

Departamento de Matemática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Pedro Quaresma

2018/07/21 (v523)

Bibliografia Principal

Metodologia Wirth, Niklaus. 1989. *Algoritmos e Estruturas de Dados*. Rio de Janeiro: Prentice-Hall do Brasil. — 68P/WIR.Alg

C Kernighan, Brian, & Ritchie, Dennis. 1988. *The C Programming Language*. 2nd edn. Prentice Hall. — 68N/KER.C

C de Sá, Joaquim P. Marques. 2004. *Fundamentos de Programação Usando C*. FCA, Lisboa, Portugal. — 68N/SA

Conteúdo

I Metodologia de Programação **11**

1 Introdução **13**
1.1 Noções Gerais 13
1.2 Organização Básica de um Computador Digital 13
1.3 A Evolução do *Hardware* 14
1.4 Informação Digital 15
1.4.1 Informação Numérica 16
1.5 Programação 20
1.6 Linguagens de Programação 20
1.6.1 Diferentes Níveis de Abstracção 20
1.6.2 Ciclo de desenvolvimento de um programa 21

2 Metodologia de Programação **23**
2.1 Introdução 23
2.2 Abstracção e Modularidade 23
2.3 Metodologia de Programação Estruturada e Descendente 24
2.4 Objectivos a Atingir na Escrita de um Programa 25

II Algoritmia **27**

3 Programação Imperativa **29**
3.1 Estruturas de Dados Básicas em *C* 29
3.2 Declarações de Variáveis 32
3.2.1 Tipo Ponteiro 33
3.3 Entradas/Saídas 35
3.4 Programas em *C* 37
3.5 Instruções Simples 39
3.5.1 Sequência Linear de Instruções 39
3.5.2 Instrução de Atribuição 39
3.5.3 Condicionais 41
3.5.4 Ciclos 44
3.6 Modularidade Procedimental 46
3.6.1 Funções 47
3.7 Estruturação de um Programa 54
3.7.1 Separação de um Programa em Múltiplos Ficheiros 54

3.8 Estruturas de Dados Compostas 57
3.8.1 Estruturas de Dados Estáticas 57
3.8.2 Estruturas de Dados Dinâmicas 60

4 A Biblioteca Padrão do *C* **69**
4.1 A Biblioteca Padrão 69
4.1.1 Diagnósticos: `<assert.h>` 69
4.1.2 Funções Matemáticas: `<math.h>` 69
4.1.3 Casos Especiais para Argumentos de Funções: `<stdarg.h>` 70
4.1.4 stdlib `<stdlib.h>` 70
4.1.5 Teste de Classes de Caracteres: `<ctype.h>` 71
4.1.6 Limites Dependentes do Compilador: `<limits.h>` & float `<float.h>` 71
4.1.7 Saídas Alternativas de Funções: `<setjmp.h>` 71
4.1.8 Definições Padrão: `<stddef.h>` 71
4.1.9 Funções para ≪Strings≫: `<string.h>` 72
4.1.10 Localização: `<locale.h>` 72
4.1.11 Sinais: `<signal.h>` 72
4.1.12 Entradas e Saídas: `<stdio.h>` 72
4.1.13 Tempo e Datas: `<time.h>` 75

5 Ordenação & Pesquisa **77**
5.1 Pesquisa 77
5.1.1 Pesquisa Exaustiva 77
5.1.2 Pesquisa Binária 77
5.2 Ordenação 78
5.2.1 Borbulhagem 78
5.2.2 Selecção Linear 78
5.2.3 Inserção Ordenada 79
5.2.4 Fusão 79
5.2.5 Quick Sort 79

6 Complexidade Algorítmica **81**

A Documentação **83**
A.1 Documentação Interna 83
A.2 Documentação Externa 84
A.3 Programação Literária 84

B Construção de um Executável **85**
B.1 Compilador de C 85
B.1.1 Pré-processamento 86
B.1.2 Opções de Compilação 88
B.1.3 Utilizar uma Biblioteca 89
B.2 Makefile 90
B.2.1 O Programa `make` 91
B.2.2 O Ficheiro `Makefile` 91
B.3 Ambientes Integrados de Desenvolvimento 94

B.3.1 Editor de Textos Dedicado 94
B.3.2 Compilação & Makefiles 95
B.3.3 Depuração de Erros 95
B.3.4 Documentação 95
B.3.5 Alguns IDEs para o *C* 95
B.4 Comandos para a Consola 96

C Exemplos **97**
C.1 Exemplos Referentes ao Capítulo 3 97

D Forma Normal Estendida de Backus-Naur **99**

E Exercícios Práticos **101**
E.1 Leitura e Escrita 101
E.2 Instruções de Atribuição 103
E.3 Tipos de Dados Simples, Inteiros e Reais 104
E.4 Tipos de Dados Simples, Caracteres ("char") e Lógicos 106
E.5 A instrução condicional "if (P) *I1* else *I2*" 108
E.6 Instruções de Repetição (Ciclos) 111
E.7 Funções: <tipo> <nome> (<lista argumentos>) $\{I\}$ 113
E.8 Recursão 115
E.9 Tipo de Dados Compostos: Tabelas ("Array") 117
E.10 Estruturas não homogéneas: Registos ("struct") 118
E.11 Ficheiros 120
E.12 Ordenação e Pesquisa 122

Referências **125**

Lista de Figuras

1.1 Modelo van Newmann . . . 13
1.2 Mark I & ENIAC . . . 14
1.3 UNIVAC I & PDP 11 . . . 15
1.4 IBM XT . . . 15
1.5 Linguagens de Programação . . . 20
1.6 Relação entre Linguagens de Programação . . . 21
1.7 Ciclo de Desenvolvimento de um Programa . . . 22

3.1 Variáveis Estáticas vs. Variáveis Dinâmicas . . . 34
3.2 Separação binária . . . 41
3.3 Separação por Casos . . . 43
3.4 Ciclos ≪`while` (P) I≫, ≪`do` $I_1; \ldots; I_n$ `while` (P)≫ e ≪`for` (C_i;C_f;inc) I≫ . . . 45
3.5 Ligação entre Argumentos e Parâmetros . . . 49
3.6 Ligação por Valor – O estado do programa não se altera . . . 49
3.7 Ligação por Referência – O estado do programa de chamada pode ser alterado 50
3.8 Variáveis Locais numa Chamada Recorrente . . . 54
3.9 Mapa de Estradas (grafo) . . . 61
3.10 Variáveis Estáticas vs Variáveis Dinâmicas . . . 62

Lista de Tabelas

1.1 Estados Binários 16
1.2 Unidades de Medida Binária 17

3.1 Precedências e Associatividade 32
3.2 Especificadores de Formato Básicos 36

4.1 Funções Matemáticas em `math.h` 70
4.2 Modos de Abertura para Ficheiros 72

D.1 Formas de Controlo do Lado Direito das Regras EBNF 99

E.1 Especificadores de Formato 102

Parte I

Metodologia de Programação

Capítulo 1

Introdução

1.1 Noções Gerais

- Informática (Teoria da Informação) — Ciência do tratamento e transmissão da informação.
- Computador Digital — Sistema digital que permite armazenar grandes quantidades de informação, e realizar sobre essa informação, a velocidades muito elevadas, manipulações e operações aritméticas e lógicas elementares.

1.2 Organização Básica de um Computador Digital

Modelo de Von Newmann A arquitectura dos computadores digitais tem-se mantido constante nas sua base: uma unidade de cálculo; uma unidade de armazenamento; uma unidade de interface com o ≪exterior≫.

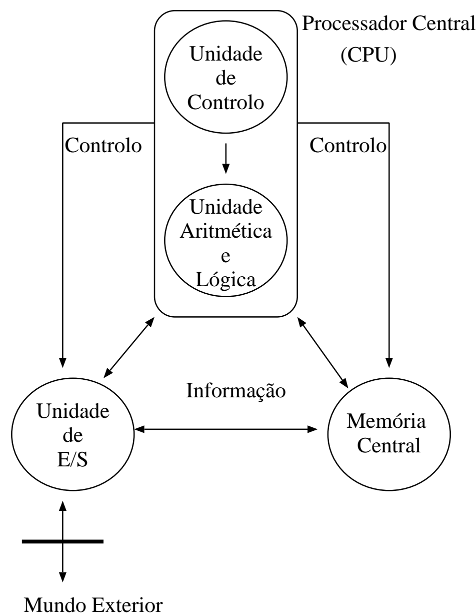


Figura 1.1: Modelo van Newmann

- Processador Central (CPU)
  - Unidade Aritmética e Lógica (UAL) — local onde se executam as operações aritméticas e lógicas elementares estipuladas pelos programas;
  - Unidade de Controlo — Extrai da memória as instruções dos programas e os dados, analisa-as e executa a consequente manipulação.
- Memória Central (*Random Access Memory*, RAM), local de armazenamento de:
  - programas — sequências de instruções que definem as tarefas a executar e por que ordem;
  - dados — informação sobre a qual se vão executar as operações (manipulações) definidas pelos programas;
  - resultados — informação resultante das operações (manipulações) efectuadas sobre os dados.
- Unidades de Entrada/Saída — unidades especializadas na troca de informação com o exterior. Estas unidades comunicam com os dispositivos periféricos, estabelecendo a ligação entre o mundo exterior e o processador central (ou a memória) e vice-versa.

1.3 A Evolução do *Hardware*

Primeiros Computadores Os primeiros computadores, ainda uma mistura de electromecânicos e electrónica (válvulas). Exemplares únicos: **Mark I**, Harvard University, 1940, 5t, 1 mult/6s (ver Figura; **ENIAC**, University of Pennsylvania, 1940, 357mult/s (ver Figura 1.2).

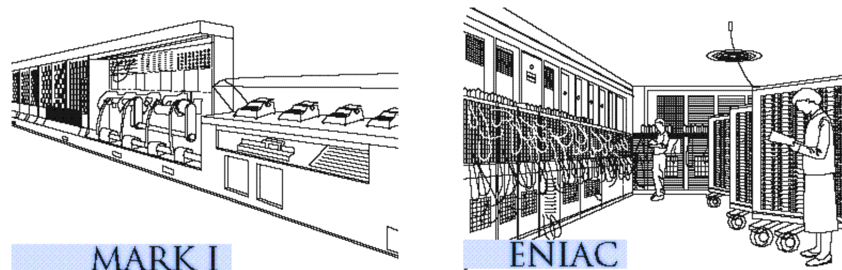


Figura 1.2: Mark I & ENIAC

Primeiros Computadores Comerciais Os primeiros computadores comerciais. Construídos em série para grandes empresas e universidades: **UNIVAC I**, 1951, construídas 48 unidades; **DEC, PDP 11** 1970, $10.800USD. Máquina que serviu de base ao desenvolvimento do sistema operativo Unix, assim como à linguagem de programação C (ver Figura 1.3).

A Revolução dos Computadores Pessoais O **IBM XT** é o primeiro computador em que a especificação era estabelecida por um dado fabricante (neste caso a IBM), mas em que o fabrico das diferentes componente era aberto (ver Figura 1.4).

A mudança de paradigma de um computador/sistema operativo, especificado e construído por uma só empresa, para um computador especificado por uma empresa mas fabricado por

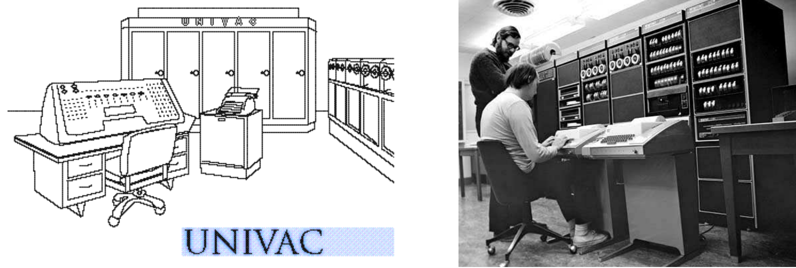


Figura 1.3: UNIVAC I & PDP 11

muitas empresas (em competição entre elas) e com um sistema operativo de outra empresa, permitiu o baixar de preço e, em consequência, uma enorme difusão.

Figura 1.4: IBM XT

1.4 Informação Digital

Com a introdução dos computadores faz-se a passagem de uma informação analógica (de *analogia*, isto é, guarda-se a informação estabelecendo uma analogia com o real), para uma informação digital. A informação passa a estar guardada num formato guardado e manipulado em máquinas (computadores) em termos de um conjunto finito de estados fixos, facilmente reconhecidos pela máquina.

Hardware Componente física (electrónica) de um computador.

Software Componente não-física (programas) de um computador.

Informação Analógica/Digital.

$$\begin{cases} \text{Analógica — variação contínua} \\ \text{Digital — variação discreta} \end{cases}$$

Informação Binária Por convenção a unidade básica de informação é binária, isto é, tem-se dois estados diferentes, 0 | 1.

O elemento de informação contido na alternativa ≪O ou 1≫ designa-se por **Dígito Binário**, em inglês, **B**inary **I**nformation **D**igit (*bit*). Por razões de simplificação é usual considerar a unidade *byte*, referente a oito *bits* (já capaz de guardar $2^8 = 256$ estados diferentes, ver Tabela 1.1). Finalmente é usual de usar a noção de ≪palavra≫, (*word*), que se refere ao número de *bits* (ou *bytes*) que num dado computador são tratados como unidade do processador (e por arrasto do restante *hardware*). Actualmente é usual ter-se computadores de 64bits e, ainda, alguns de 32bits.

1 bit	$\rightarrow$	0	2 estados diferentes
		1	
2 bits	$\rightarrow$	0 0	4 estados diferentes
		0 1	
		1 0	
		1 1	
3 bits	$\rightarrow$	...	8 estados diferentes
	$\vdots$		
n bits	$\rightarrow$	...	2^n estados diferentes

Tabela 1.1: Estados Binários

Byte (Binary Term): 8 bits.

Palavra: número de bits que constituem as unidades manipuláveis por um determinado computador (número de bits agrupados num elemento endereçável de uma só vez).

Memória: sequência numerada de células, cada uma permitindo o armazenamento de uma palavra. Cada célula é identificável pelo seu endereço.

A capacidade de memória mede-se pelo número de palavras que consegue armazenar. Inicialmente usou-se a base dois (*bit*) com medida e referia-se a 1000 vezes (1Kb) a unidade como sendo $2^{10} = 1024$. No entanto esta convenção entra em conflito com a convenção internacional usada em todas as unidades decimais, em que a a designação ≪kilo≫ (K) está reservada para $10^3 = 1000$. Actualmente tem-se que as designações convencionais respeitam a convenção internacional (de base 10, ver Tabela 1.2). Por razões históricas, assim como medida directamente relacionada com a ≪palavra≫ de um computador definem-se também as medidas *XbiByte*, baseadas na base 2 (ver Tabela 1.2).

1.4.1 Informação Numérica

A representação numérica guardada digitalmente separa-se (usualmente) em dois formatos distintos. A representação de números inteiros, (`int` na linguagem C, $n \in \mathbb{N}$), e números reais, (`float/double` na linguagem C, $x \in \mathbb{R}$).

Inteiros

A conversão é exacta. A menos de representações específicas, por exemplo as providenciadas pela biblioteca GMP, *The GNU Multiple Precision Arithmetic Library*[1], a representação é

[1] `http://gmplib.org/`

Nome	Símbolo	Valor	Nome	Símbolo	Valor
quilobyte	KB	10^3	quibibyte	KiB	2^{10}
megabyte	MB	10^6	mebibyte	MiB	2^{20}
gigabyte	GB	10^9	gibibyte	GiB	2^{30}
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}

Tabela 1.2: Unidades de Medida Binária

finita, por exemplo para os tipos básicos em C, `int / unsigned int` e `long int / unsigned long int` tem-se:

- `(int,+,-,*,/,%)` - implementação dos Inteiros em C.
  - representação exacta (conversão entre base 2 e base 10);
  - gama da variação finita:
    - `int`, 32bits/4bytes
      - signed: -2147483648 a 2147483647
      - unsigned: 0 a 4294967295
    - `long int`, 64bits/8bytes
      - signed: -9223372036854775808 a 9223372036854775807
      - unsigned: 0 a 18446744073709551615
  - operações com as propriedades usuais;
  - «/» é a divisão inteira, «%» é o resto da divisão inteira;
  - representação usual (na base 10).

Conversão base 2 para base 10

$$\begin{aligned} 10000110_2 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + \\ &\quad +0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 134_{10}(1 \times 10^2 + 3 \times 10^1 + 4 \times 10^0) \end{aligned}$$

Conversão base 10 para base 2 Dividir por 2, reter o resto e repetir tudo até não ser possível realizar a divisão, o número obtêm-se "lendo" os restos por ordem inversa.

dividendo	resto	
134	0	
67	1	
33	1	
16	0	
8	0	
4	0	
2	0	
1	1	↑ leitura de baixo para cima

- A representação é exacta.
- As operações não induzem em erros (excepto no que diz respeito à ultrapassagem da capacidade).

Reais

A conversão **não é** (em geral) **exacta**.[2]

$$\begin{aligned} 23.48_{10} & \\ 23_{10} &= 10111_2 \\ 0.48_{10} &= 0.01111010\ldots_2 \end{aligned}$$

A conversão da base 10 para a base 2 da parte decimal: multiplica-se por dois e retêm-se a parte inteira, repete-se todo o processo até se obter zero no resultado. Obtêm-se o número fazendo a leitura directa das partes inteiras que se foram obtendo.

Por exemplo: $0.1_{10} = 0.0(0011)_2$

$$\begin{array}{ccc} \downarrow & 0 & 0.1 \\ & & \times 2 \\ \hline & 0 & 0.2 \\ & & \times 2 \\ \hline & 0 & 0.4 \\ & & \times 2 \\ \hline & 0 & 0.8 \\ & & \times 2 \\ \hline & 1 & 1.6 \\ & & 0.6 \\ & & \times 2 \\ \hline & 1 & 1.2 \\ & & 0.2 \\ & & \times 2 \\ \hline & 0 & 0.4 \\ & & \vdots \end{array}$$

Representação Vírgula Flutuante A representação de números reais na, assim designada, *representação em vírgula flutuante* é usada para minimizar os erros de representação.

$$x = m \times B^e, \quad -E < e < E \quad -M < m < M$$

m – Mantissa; B – Base; e – Expoente.

Na forma canónica tem-se

$$\frac{M}{B} \leq m < M$$

[2] Vai-se usar a notação inglesa para a escrita de números, separação da parte inteira e da parte decimal com '.', por exemplo 23.48, dado que é essa a notação da linguagem C

Para $B = 10, M = 1$ tem-se $0.1 \leq m < 1$.
Por exemplo: $23.48 = 0.2348 \times 10^2$

Esta forma de representação permite fixar o número de *bits* a utilizar em ambas as secções (mantissa e expoente) permitindo uma representação, que não sendo exacta, tem boas características: para aumentar a gama de números representados acrescentam-se *bits* ao expoente; para aumentar a precisão, acrescentam-se *bits* à mantissa.

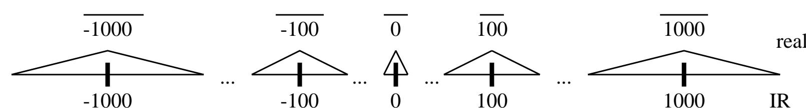
- `(float/double,+,-,*,/)`—implementação dos Reais em C.
  - representação não exacta (conversão entre base 2 e base 10);
  - representação interna na base 2, em vírgula flutuante;
  - gama de variação finita:
    - `float`, 32bits/4bytes, 1.175494e-38 a 3.402823e+38
    - `double`, 64bits/8bytes, 2.225074e-308 a 1.797693e+308
    - `long double`, 128bits/16bytes, 3.362103e-4932 a 1.189731e+4932
  - operações com as propriedades usuais;
  - representação usual e notação científica (notação inglesa): por exemplo, 3.1415 e 0.31415e1

Representação Interna ($4 \times 8 = 32$bits).

expoente	mantissa
8 bits	24 bits

Conclusão:

- A representação **não** é, em geral, exacta.
- As operações podem induzir mais erros (além da ultrapassagem da capacidade).
- Na forma normal a densidade de representantes decresce exponencialmente com o crescimento de $|x|$.



- O número de «bits» reservado para a mantissa dá o grau de precisão que uma dada representação interna possui.
- O número de bits reservado para o expoente dá a extensão da gama representada.
- `float` $\subsetneq \mathbb{R}$. Gama de variação finita e uma representação discreta (finita).

1.5 Programação

Transformar formas de resolução de problemas, em programas capazes de serem usados para obter as soluções do problema original.

> **Programas = Estruturas de Dados + Algoritmos**
>
> Niklaus Wirth

Estrutura de Dados: Um Conjunto de Valores e um Conjunto de Operações sobre esses valores.

Algoritmo: Procedimento mecânico (automático) que produz sempre o mesmo resultado ao fim de um número finito de passos.

1.6 Linguagens de Programação

1.6.1 Diferentes Níveis de Abstracção

Todo e qualquer programa tem de ser convertido para a ≪linguagem≫ da máquina, conjunto de instruções específico a cada um dos processadores centrais (CPU) existentes.

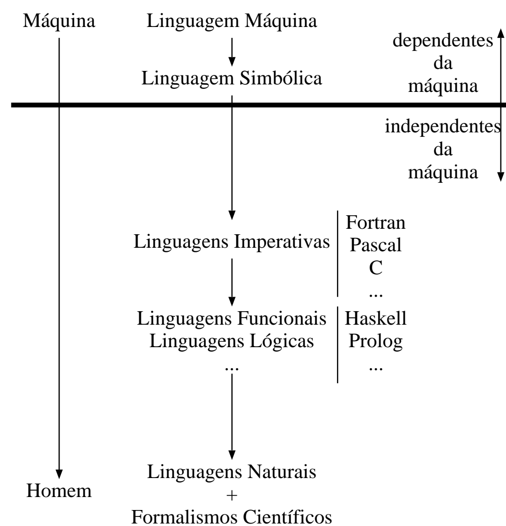


Figura 1.5: Linguagens de Programação

Como estas ≪linguagens≫ (de baixo nível) são ilegíveis (ou quase) por programadores foram criadas linguagens de programação que, por um lado possam ser usadas por programadores (alto nível), e que por outro lado possam ser convertidas automaticamente nas ≪linguagens≫ das máquinas.

Existem diferentes famílias de linguagens de programação que se podem distinguir pelo seu nível de abstracção e modularidade.

As diferentes famílias de linguagens são baseadas nos diferentes modelos teóricos da computação, a saber: a máquina de Turing (linguagens imperativas); o cálculo-λ (linguagens funcionais); e a lógica das cláusulas de Horn (linguagens lógicas). Foi provado que estes três formalismos têm o mesmo poder computacional.

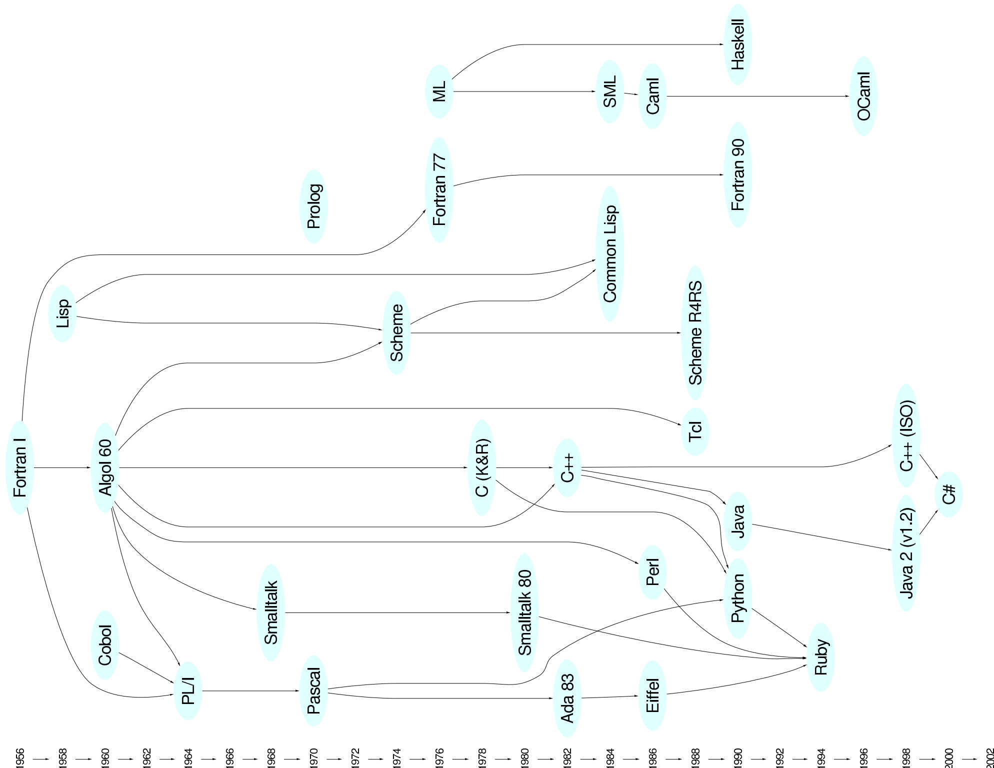


Figura 1.6: Relação entre Linguagens de Programação

1.6.2 Ciclo de desenvolvimento de um programa

O desenvolvimento de um programa da ideia inicial até ao produto final, um ≪programa executável≫ é um processo com muitos passos (Ver Figura 1.7).

Numa primeira fase há que transformar o problema que se quer resolver, nem sempre definido de forma clara e precisa, numa especificação, isto é uma descrição clara, precisa e o mais rigorosa possível do problema. Para este primeiro passo existem ferramentas e formalismos apropriadas, a sua complexidade está além do que se pretende com uma disciplina de iniciação.

Vai-se assim assumir que os problemas já estão descritos de uma forma clara e precisa. É então necessário trabalhar na construção do algoritmo, i.e. um procedimento mecânico que, traduzindo o problema que se quer resolver, devolve sempre o mesmo resultado, para um dado conjunto de valores iniciais.

No passo seguinte o algoritmo deve depois ser transformado num programa (numa dada linguagem de programação). Depois disso um programa especial (o compilador) traduz este programa numa sua representação no código da máquina destino.

O processo de compilação pode ser bem sucedido (algo sempre difícil de obter à primeira tentativa) ou falhar por ocorrência de uma ou mais erros.

Os erros de compilação são de diferentes tipos:

- erros léxicos — um erro na escrita de uma palavra ou símbolo reservado;
- erros sintácticos — um erro na escrita de uma das estruturas da linguagem, e.g. a omissão de um dado elemento;

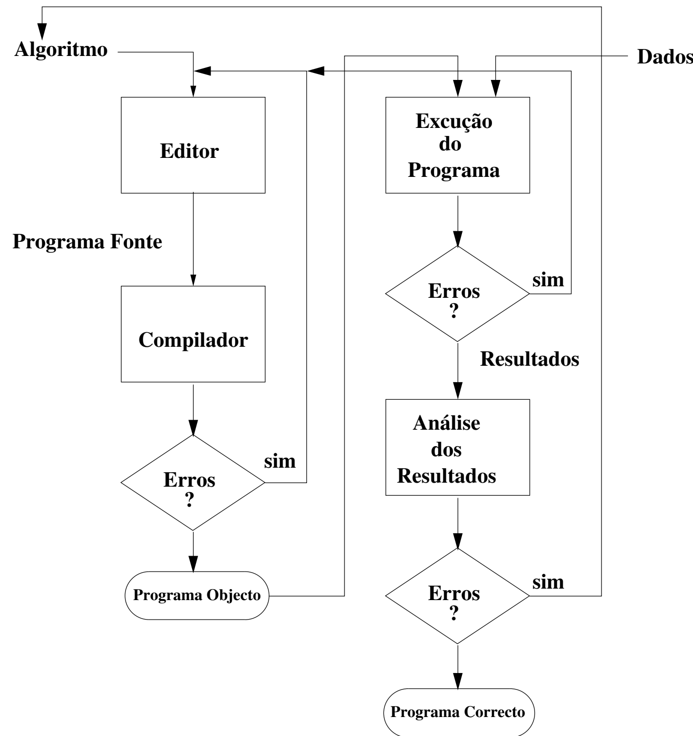


Figura 1.7: Ciclo de Desenvolvimento de um Programa

- erros semânticos — um erro relacionado com os tipos da variáveis usadas no programa.

Em qualquer dos casos o compilador irá assinar a situação de erro, dizendo aonde pensa que o erro ocorreu (linha do programa), e qual é o erro.

É importante ter em conta que, em geral, um primeiro erro vai, em cascata, provocar outros erros. Como tal é importante concentrar a nossa atenção no primeiro erro relatado pelo compilador, ler a sua descrição (em geral em Inglês) e localizá-lo, tendo em conta que o mesmo se encontra na linha assinalada, ou numa linha acima.

Se bem sucedido o processo de compilação termina com a produção de um executável. Será que o processo terminou?

Não exactamente, o compilador não detecta (nem pode detectar quando se considera as actuais ferramentas de compilação) erros lógicos, isto é, o programa pode funcionar mas sem que produza os resultados desejados.

A fase final na construção de um programa passa então pela análise dos resultados obtidos para um dado conjunto de valores de entrada, comparando-os com os resultados esperados.

A fase de análise dos resultados é um processo incompleto, na grande maioria dos casos o número de diferentes casos a considerar, por variação dos dados de entrada, é infinito. Como tal a verificação de um elevado número de casos irá dar-nos uma confirmação, mas não uma certeza, da correcção do programa.

A demonstração formal da correcção de um programa é uma questão em aberto, embora haja progressos importantes a registar nesta área ainda se está longe de um processo em que seja possível desenvolver as demonstrações formais da correcção de programas.

Capítulo 2

Metodologia de Programação

2.1 Introdução

A, assim designada, *Crise na Programação* («Software Crises») surgiu quando se tornou evidente que os problemas que surgiam no desenvolvimento dos programas não eram algorítmicos mas sim de comunicação entre partes de um mesmo programa e de complexidade do programa como um todo.

A abstracção e a modularidade são conceitos determinantes para poder lidar com os problemas acima referidos.

2.2 Abstracção e Modularidade

Programas e problemas são, normalmente, sistemas grandes e complexos. A capacidade humana para entender tais sistemas é limitada e só consegue ter sucesso quando consegue aplicar uma de duas estratégias: a abstracção e a modularidade.

Abstracção A abstracção é a capacidade para agrupar um número elevado de casos concretos diferentes (mas semelhantes) numa única entidade a que podemos dar o nome de *classe*, isto é, as propriedades comuns a todos os casos concretos de modo a que, face a essas propriedades, seja possível:

- prever os efeitos das transformações a que, eventualmente, estejam sujeitos os diferentes casos dentro da classe;
- validar frases lógicas que se apliquem universalmente a todos os casos da classe.

A estratégia de abstracção baseia-se em três técnicas fundamentais:

Instanciação Dada uma classe criar um dos seus casos (uma instância da classe);

Abstracção Dados dois ou mais casos concretos, caracterizar a «menor classe» que os contém como instâncias;

Validação Dada uma classe e um caso concreto verificar se o caso é uma instância da classe ou
Prever os efeitos de transformações em instâncias de classe

ou
Validar frases lógicas quantificadas universalmente às instâncias da classe.

Modularidade A modularidade é a capacidade para decompor casos complexos numa colecção de casos mais simples (chamados módulos) e em regras de composição de tal modo que:

- A análise (ou síntese) de qualquer um dos módulos seja independente da análise (ou síntese) dos restantes. De preferência cada módulo deve ser uma instância de uma classe de módulos bem conhecida.
- É possível construir o significado do caso complexo por aplicação das regras de composição ao significado dos módulos. Analogamente, se o problema for de síntese, as regras de composição devem permitir construir o sistema global a partir dos módulos.

2.3 Metodologia de Programação Estruturada e Descendente

Para as linguagens de programação imperativas desenvolveu-se uma metodologia de programação, designada por *metodologia de programação estruturada e descendente* que, aplicando os princípios acima descritos, é apropriada para a construção de programas, fáceis de escrever, ler, modificar e, que por isso mesmo, serão correctos ou facilmente modificados para ficarem correctos (pelo menos assim se espera).

A *metodologia de programação estruturada e descendente* é o assunto principal do capítulo 2.

A *Metodologia de Programação Estruturada e Descendente* surge como forma de construção de programas que, tendo em atenção os conceitos de abstracção e modularidade pretende ultrapassar a crise na programação.

A metodologia de programação estruturada e descendente é caracterizada da seguinte forma:

> Decompor o problema global em sub-problemas específicos (por tarefa bem determinada), provar que se cada sub-problema é resolvido correctamente, e se as várias soluções parciais podem ser combinadas de forma correcta, então o problema global será resolvido correctamente.
>
> Repetir este processo até atingir sub-problemas simples.
>
> S. Alagic e M.A. Arbib (Alagič & Arbib, 1978)

Numa aproximação ideal a esta forma de construção de programas os programas seriam estruturados em uma série de módulos independentes e interligados entre si de forma a que respeitassem o seguinte princípio:

Proposição 1 (Princípio de Parmas (David Parmas))

1. *Deve-se dar ao utilizador de um módulo toda a informação necessária para este usar o módulo correctamente, e nada mais.*
2. *Deve-se dar ao programador de um módulo toda a informação necessária para este completar o módulo, e nada mais.*

No caso de uma linguagem procedimental como o C a modularidade expressa-se pela divisão de um dado programa em funções independente entre si. As ligações entre funções é efectuada pelos argumentos das funções e pelo resultado das mesmas. O programa é então um conjuntos de funções, cada uma com uma tarefa simples e bem determinada para resolver, que se combinam para atingir o um dado fim.

2.4 Objectivos a Atingir na Escrita de um Programa

Em C um programa é escrito num, ou mais, ficheiro(s) sendo constituído obrigatoriamente por uma função designada por `main`, a qual determina o início do algoritmo e, eventualmente, por outras funções.

$$\text{Objectivos a atingir} \begin{cases} \text{Correcção} \\ \text{Clareza} \\ \text{Eficiência} \end{cases}$$

Pretende-se obter um programa **correcto** que seja **fácil de compreender** e/ou **modificar**, mesmo que seja por uma programador diferente daquele que o escreveu.

Além disso entre duas soluções, ambas correctas e escritas de uma forma clara escolhe-se, naturalmente, a **mais eficiente**.

Correcção: O Programa deve estar de acordo com a Especificação.

- Simplicidade.
- Aproximação sistemática—Metodologia de Programação Estruturada e descendente.
- Testes e/ou Verificação formal.

Clareza: O Programa deve reflectir a estrutura do algoritmo, e além disso, deve ser fácil de ler.

- Separação em blocos funcionais.
- Uso adequado das estruturas da linguagem.
- Escolha cuidadosa dos identificadores.
- Documentação interna.
- Aspecto gráfico:
  - linhas em branco;
  - indentação.

Eficiência: Comparação entre várias soluções. Avaliada em termos de:

- tempo gasto pelo computador;
- espaço de memória usado.

Deve-se privilegiar a correcção acima de tudo, não faz sentido ter um programa muito eficiente, mas incorrecto. A clareza é uma forma de nos ajudar a obter um programa correcto.

Parte II

Algoritmia

Capítulo 3

Programação Imperativa

Programas = Algoritmos + Estruturas de Dados

Niklaus Wirth

Do ponto de vista da programação procedimental um computador é uma máquina de estados. De um estado inicial, isto é, os dados de entrada, guardados num conjunto de variáveis que em conjunto definem o estado do programa, pretende-se atingir um dado estado final, definidor do resultado que se pretende obter com o programa.

Um programa é então uma sequência linear de transições de estado. O algoritmo define essa sequência, as estruturas de dados determinam o estado, o qual vai variando temporalmente à medida que se processam as diferentes instruções definidas no algoritmo.

3.1 Estruturas de Dados Básicas em *C*

As estruturas matemáticas que podemos considerar como básicas na modelização de situações numéricas são:

- $(\mathbb{Z}, +, -, \times, /, 0, 1)$ - corpo dos inteiros;
- $(\mathbb{R}, +, -, \times, /, 0, 1)$ - corpo dos reais;
- $(\mathbb{B}, \wedge, \vee, \neg, \mathscr{V}, \mathscr{F})$ - corpo dos Booleanos (lógicos)
- Letras - alfabeto (ASCII/ISO-8859-1/etc.)
- Letras* - palavras formadas por sequências de símbolos do alfabeto.

em *C* temos as seguintes estruturas de dados (os valores concretos são referentes ao compilador *gcc version 7.2.0* e ao sistema operativo *SMP Debian 4.14.7-1 (2017-12-22)* (ver secção C.1):

- `(int,{+,-,*,/,%})`—implementação dos Inteiros.
  - representação exacta (conversão entre base 2 e base 10);
  - gama da variação finita:
    - `char`, 8bits/1byte
      - signed: -128 a 127

        · unsigned: 0 a 255
    * `short int`, 16bits/2bytes
        · signed: -32768 a 32767
        · unsigned: 0 a 65535
    * `int`, 32bits/4bytes
        · signed: -2147483648 a 2147483647
        · unsigned: 0 a 4294967295
    * `long int`, 64bits/8bytes
        · signed: -9223372036854775808 a 9223372036854775807
        · unsigned: 0 a 18446744073709551615
    * `long long int`, 64bits/8bytes
        · signed: -9223372036854775808 a 9223372036854775807
        · unsigned: 0 a 18446744073709551615
  - operações com as propriedades usuais;
  - ≪/≫ é a divisão inteira, ≪%≫ é o resto da divisão inteira;
  - representação usual (na base 10).

- `(float/double,{+,-,*,/})`—implementação dos Reais;
  - representação não exacta (conversão entre base 2 e base 10);
  - representação interna na base 2 (em vírgula flutuante);
  - gama de variação finita:
    * `float`, 32bits/4bytes, -3.40282e+38 a 3.402823e+38;
      Valor ínfimo: 1.175494e-38
      Grau de precisão: 1.19209e-07
    * `double`, 64bits/8bytes, -1.797693e+308 a 1.797693e+308
      Valor ínfimo: 2.225074e-308
      Grau de precisão: 2.22045e-16
    * `long double`, 128bits/16bytes, -1.189731e+4932 a 1.189731e+4932
      Valor ínfimo: 3.362103e-4932
      Grau de precisão: 1.0842e-19
  - operações com as propriedades usuais;
  - representação usual e notação científica: por exemplo, `3.1415` e `0.31415e1` (notação inglesa, '3.14' em vez de '3,14')

- Os Booleanos são implementados em (C) através do tipo `int`. Tem-se então `(int,{&&,||,!})`
  - $\mathscr{V}$: qualquer inteiros diferente de zero;
  - $\mathscr{F}$: 0 (zero).
  - Conectivas lógicas com as propriedades usuais da lógica proposicional.
    * `&&` – conjunção;
    * `||` – disjunção;
    * `!` – negação.

- Símbolos relacionais com as propriedades usuais.
  - == – igualdades;
  - != – diferença;
  - <, <= – menor e menor ou igual;
  - >, >= – maior e maior ou igual.
- (char,{+,-,*,/,%})—Alfabeto interno;
  - Aparte o alfabeto *ASCII* não existe uma norma globalmente aceite, este facto tem como consequência que nem sempre é possível assegurar que um carácter acentuado seja representado de forma correcta entre plataformas computacionais diferentes.
  - Como vimos acima o tipo char pode ser visto como um sub-tipo dos inteiros, dito de outra forma, internamente o *C* lida com os caracteres em termos da sua codificação não fazendo distinção entre esses valores e os outros valores do tipo int. Note-se no entanto que o tipo char tem uma gama de variação de somente 256 elementos (8bits).
- (char*,{})—sequências de símbolos (*strings*) do Alfabeto (*C*).
  - As *string* não são um tipo básico em (*C*). A sua manipulação é feita por um conjunto de funções definidas na biblioteca padrão da linguagem (ver Capítulo 4.1).
  - As constantes deste tipo são dadas por uma qualquer sequência de caracteres entre aspas (angulares). Por exemplo: "Olá Mundo".
- (<tipo>*,{+,-,*,/,%})—ponteiros para um dado tipo de dados. Neste caso não temos verdadeiramente um tipo de dados mas sim referências, isto é, o explicitar da ligação entre os identificadores (nomes das variáveis de um dado tipo) e os seus valores (células de memória aonde são guardados os valores). Os ponteiros não são mais do que valores naturais (entre 0 e o valor máximo da memória RAM que um dados sistema operativo suporta) que identificam as células de memória aonde os valores das variáveis são guardados (ver Figura 3.1)
  - valores inteiros entre 0 e o valor máximo que o sistema operativo suporta (Linux 64bits, 256GiB ($= 256 * 2^{30}$).
  - todas as operações com inteiros. No entanto dado se tratar de ponteiros, isto é referências a células de memória, a sua manipulação directa como valores inteiros deve ser feita com extremo cuidado.

Os tipos int, float, double e char são tipos atómicos, isto é, os seus elementos não são decomponíveis em partes menores, o tipo char* é já um tipo composto, os seus elementos são sequências de elementos do tipo char, ver-se-á mais à frente como é possível construir outros tipos compostos, sejam em termos de estruturas estáticas (§ 3.8), seja em termos da construção de estruturas dinâmicas (em Métodos de Programação II).

O tipo ponteiro é um tipo simples apropriado para criar estruturas compostas, vamos ver já de seguida como manobrar as variáveis deste tipo quando os elementos apontados são de um tipo atómico. A sua utilização para criar estruturas dinâmicas será visto mais à frente (em Métodos de Programação II).

A construção de expressões utilizando variáveis e constantes dos tipos acima descritos seguem as regras usuais da escrita de expressões matemáticas. Mais concretamente na tabela 3.1, podemos ver as regras da linguagem *C* referentes à precedência dos operadores assim como a associatividade dos mesmos.

<table>
<tr><th></th><th>Operadores</th><th>Associatividade</th></tr>
<tr><td rowspan="10">Precedências</td><td>() [] -> .</td><td>esq. para dir.</td></tr>
<tr><td>! ++ -- +(unário) -(unário) *(ponteiros) &</td><td>dir. para esq.</td></tr>
<tr><td>*(multiplicação) / %</td><td>esq. para dir.</td></tr>
<tr><td>+ -</td><td>esq. para dir.</td></tr>
<tr><td><< >></td><td>esq. para dir.</td></tr>
<tr><td>< <= > >=</td><td>esq. para dir.</td></tr>
<tr><td>== !=</td><td>esq. para dir.</td></tr>
<tr><td>&&</td><td>esq. para dir.</td></tr>
<tr><td>||</td><td>esq. para dir.</td></tr>
<tr><td>= += -= *= /= %=</td><td>dir. para esq</td></tr>
</table>

Tabela 3.1: Precedências e Associatividade

3.2 Declarações de Variáveis

Já tendo visto quais são os tipos básicos disponíveis coloca-se agora a questão de como declarar variáveis desses tipos, isto é, como construir o estado do programa.

Em *C* a declaração de uma variável pode ser feita em qualquer ponto do programa, a variável irá ser incorporada ao estado do programa a partir desse ponto.

Por razões metodológicas é usual agrupar todas as declarações na zona inicial de cada uma das funções constituintes do programa, nomeadamente a função `main`, desse modo é fácil de saber quais são as diferentes componentes definidoras do estado do programa.

Temos então as seguintes variantes:

- Declaração de uma variável:

```
char c;
int i;
```

- Declaração de várias variáveis:

```
float a,aux,delta;
```

- Declaração e inicialização:

```
double x=7.4;
```

- Declaração de uma constante:

```
const double pi=3.14159;
```

De uma forma mais geral tem-se (ver o Apêndice D para uma explicação da meta-sintaxe):

```
DeclaraçãoVariável  ::= [ const ] <tipo> <LstVarInicializações> ;

LstVarInicializações ::= <identificador> [= <valor> ] [, <LstVarInicializações> ];
```

Após as declarações o estado do programa passa a conter os valores das variáveis, os quais estão associados aos nomes das variáveis. No exemplo acima ter-se-ia:

`c`	???
`i`	???
`a`	???
`aux`	???
`delta`	???
`x`	7.4
`pi`	3.14159

Quando uma variável é declarada e não inicializada o seu valor deve se tido como indeterminado, o assumir de um qualquer valor inicial, definido por omissão, é um erro. A inicialização, ou não, e o valor que é fixado por omissão, são dependentes do compilador que se está a usar, não havendo nenhuma garantia que o mesmo tipo de inicialização se verifique em dois compiladores diferentes.

3.2.1 Tipo Ponteiro

Como foi dito no princípio deste capítulo um programa do tipo procedimental é uma máquina de estados. De um estado inicial as diferentes instruções de atribuição vão modificando esse mesmo estado inicial até se obter os resultados desejados.

O estado de um programa é caracterizado por um conjunto de identificadores (nomes de variáveis) e de valores (posições de memória) que lhe estão associados. As variáveis estáticas (sem ponteiros) e as variáveis dinâmicas têm um comportamento diferente na forma como associam estas duas entidades, nomes e posições de memória. No primeiro caso as referências são geridas automaticamente pelo programa, no segundo essa é já parte das responsabilidades do programador.

Vejamos uma situação em concreto (ver Figura 3.1): a declaração de uma variável estática do tipo inteiro, `x`, e a sua posterior inicialização; e a declaração de uma variável dinâmica, um ponteiro para um inteiro, `z`, a inicialização do espaço e a posterior inicialização do valor apontado.

Aquando da declaração de um variável estática temos as seguintes acções:

1. guardar o nome na tabela de identificadores;
2. reservar a memória necessária para guardar um valor do tipo associado ao identificador;
3. inicializar a referência associada (ponteiro, número natural referente a uma posição de memória) com o valor correspondente à posição de memória que foi anteriormente reservada.

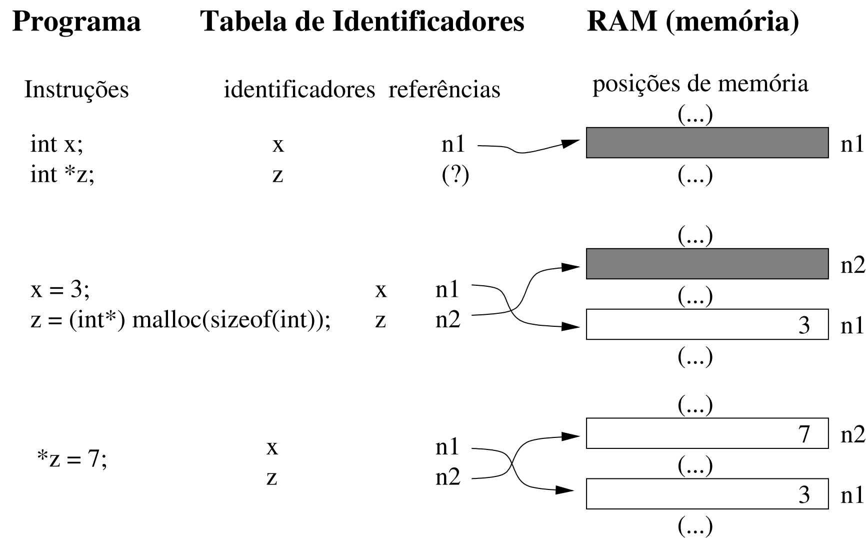


Figura 3.1: Variáveis Estáticas vs. Variáveis Dinâmicas

o valor associado à variável começa por ser indefinido.

Aquando da declaração de uma variável dinâmica temos a seguinte acção: guardar o nome na tabela de identificadores.

A referência associada a este nome começa por ser indefinida, qualquer tentativa de acesso ao valor (ainda inexistente) associado é um erro de acesso à memória.

Para uma variável dinâmica é necessário fazer uma reserva de memória de forma explícita (através do operador `malloc`) e só depois é que é possível associar-lhe um valor.

Para uma variável do tipo estático é possível aceder à sua referência (ponteiro) de forma explícita utilizando o operador de "`&`".

declaração	identificador	referência	valor (referenciado)
<tipo> `*x`	`x`	`&x`	`*x`

Para uma variável do tipo dinâmico é possível aceder ao valor que lhe está associado através do operador ≪`*`≫

identificador (do tipo ponteiro)	valor (referenciado)
`x`	`*x`

Retomando o exemplo apresentado na figura 3.1 vejamos um exemplo da declaração, afectação e atribuição de valor a uma variável do tipo ponteiro.

Declaração começa-se por declarar uma variável do tipo ponteiro para um dado tipo específico, por exemplo ponteiro para um inteiro;

```
int *z;
```

neste momento ainda não é possível atribuir um valor (inteiro) a esta variável. Só o espaço para o ponteiro é que foi criado e nem sequer está inicializado. O tentar aceder ao valor de uma variável do tipo ponteiro nesta situação é considerado uma situação de erro (figura 3.1, primeira secção).

Afectação ou seja é necessário criar explicitamente o espaço de memória (para um inteiro) e inicializar o ponteiro para que este aponte para essa posição de memória.

```
z = (int) malloc (sizeof(int));
```

a posição de memória em concreto não é inicializada (figura 3.1, secção do meio). Mais à frente (§3.8.2) falaremos com mais detalhe da funções de gestão de memória (`malloc` e `free`).

Atribuição Já tendo o espaço de memória assim como o ponteiro para essa posição, a atribuição e/ou utilização de valores é feita de forma (quase) normal, por exemplo:

```
*x = 7;
```

Como 'x' é uma variável do tipo ponteiro o acesso ao valor que lhe está associado é feito através do operador `*x` (figura 3.1, última secção).

Como é fácil de perceber a importância das variáveis do tipo ponteiro não está na sua utilização como alternativa às variáveis dos tipos simples. A importância dos ponteiros em *C* prende-se com a necessidade de fazer a passagem, por referência, de valores entre módulos distintos (ver §3.6.1), assim como com as estruturas de dados tanto as estáticas como as dinâmicas (ver §3.8).

3.3 Entradas/Saídas

Para ≪dar≫ valores a um programa e ≪receber≫ resultados é necessário considerar as, assim designadas, instruções de entrada e saída.

As instruções de entrada/saída estabelecem uma ligação entre o programa e o sistema operativo e permitem estabelecer canais de comunicação permitindo por essa via transferir valores de e para o programa:

```
int printf(char *formato, arg c, arg2, ...)
```

O formato, uma constante do tipo sequência de caracteres, poder conter dois tipos de objectos: caracteres ordinários, os quais são copiados para o fluxo de dados de saída, e as especificações de conversão. Este último são alinhados com os argumentos.

Cada especificador de conversão é do tipo:

```
%<opções><carácter de conversão>
```

Ver a tabela E.1 para os diferentes caracteres de conversão.

As opções são:

- o sinal menos, '-', que significa o adoptar de um alinhamento à esquerda (em relação à dimensão do argumento).
- um número (inteiro) que determina o tamanho mínimo da "impressão". Se o valor a imprimir for mais largo do que o valor especificado, o espaço é automaticamente alargado.
- um ponto final '.', o qual faz a separação entre o tamanho mínimo e o tamanho definido para a parte decimal (números reais).

- um número (inteiro) que especifica o tamanho máximo da parte decimal de um número real, ou o número mínimo de caracteres/dígitos a serem apresentado.
- um 'h' se o inteiro é para ser visualizado como `short` ou 'l' (letra éle) se é para ser visualizado como um `long`.

Carácter	Tipo Argumento — Visualizado como
d,i	int — representação decimal.
o	int — representação octal (sem zero inicial).
x, X	int — representação hexadecimal (sem um zero inicial, `0x` ou `0X`), usa-se `abcdef` ou `ABCDEF` para os dígitos de $10, \ldots, 15$.
u	int — representação decimal sem sinal.
c	int — um carácter.
s	char * — caracteres de uma «string» até à terminação '\0' ou o números de caracteres dado pela precisão.
f	double — [- 1m.dddddd, aonde o número de 'd's é dado pelo segundo campo numérico (por omissão 6).
e,E	double — [-1m.dddddde±xx or [-1m.ddddddE±XX, aonde o número de 'd's é dado pelo segundo campo numérico (por omissão 6).
g,G	double — usar %e ou %E se o expoente é menor do que -4 ou maior do que o segundo campo númerico — de outra forma usar '%f'. O ponto e os zeros finais são omitidos.
p	void * — ponteiro (representação dependente do compilador).
%	sem argumentos — '%'.
lf	double — leitura de um real do tipo `double`.
Lf	long double — leitura de um real do tipo `long double`.

Tabela 3.2: Especificadores de Formato Básicos

Leitura A leitura é feita associando uma lista de valores a uma lista de variáveis. Estas duas listas devem ser consistentes em termos de número e tipo dos valores e respectivas variáveis. Se forem dados mais valores do que aqueles que são esperados, os valores que estão a mais serão ignorados pela instrução de leitura, se forem dado menos valores do que os requeridos o programa ficará à espera de que os mesmos sejam fornecidos, isto no caso da leitura estar a ser feita através do teclado, ou ocorrerá um erro, para os casos em que a leitura possa estar a ser feita de outros meios, por exemplo ficheiros.

De onde dos canais de entrada, por omissão, a linha de comando e o teclado.

Como sequência de valores separados por um ou mais espaços e de acordo com a representação definida para os tipos das variáveis escolhidas.

Efeito afectação dos valores lidos às variáveis.

Exemplo:

```
int a,b,c;
...
scanf("%d%d%d",&a,&b,&c);
```

Escrita A escrita efectua-se a partir de uma lista de valores, sejam eles constantes, variáveis, ou mesmo expressões.

Para onde canal de saída, por omissão o ecrã.

Como cálculo dos valores das expressões, conversão dos valores de acordo com a forma normal para os diferentes tipos e sua visualização.

Efeito nenhum (as variáveis não são afectadas).

Exemplos:

```
float x;
...
printf("%f",x);
```

```
float x,y;
...
printf("O valor pretendido é %f\n",x+y);
```

Nestes último exemplo temos um valor constante do tipo ≪`string`≫, a qual termina com o carácter especial, `'\n'`, cujo efeito é o de provocar uma mudança de linha no canal de saída.

3.4 Programas em *C*

Algoritmo: Procedimento mecânico (automático) que produz sempre o mesmo resultado ao fim de um número finito de passos.

Um programa em *C* é constituído por uma função especial, `main`, ponto de início e fim (em condições normais) do funcionamento do programa e, eventualmente, por outras funções.

Um programa mínimo em *C* é dado pelo programa vazio:

```
int main() {}
```

sendo que:

`int` é o tipo do resultado (valor de saída) da função.

`main` é o nome da função;

`()` é a lista, neste caso vazia, de argumentos da função;

{ } é o corpo da função, neste caso vazio, que define o funcionamento da função.

A função `main` é um exemplo de uma função em *C* cuja sintaxe genérica é:

função ::= <tipo> <identificador> `(` <listaArgumentos> `)` { <corpoDaFunção> }
listaArgumentos ::= <tipo> <identificador> [`,` listaArgumentos]
corpoDaFunção ::= <listaInstruções>

sobre a lista de instruções falaremos mais adiante.

Um programa mínimo, mas já com efeitos, `eco.c`:

```
#include <stdio.h>

/*
 * Programa «eco»
 * -> uma palavra (na linha de comando)
 * <- a mesma palavra ecoada de volta
 */

int main(int argc, char *argv[]) {
  printf("Eco %s\n",argv[1]);
  return(0);
}
```

Para o compilar bastaria (ver Apêndice B):

```
gcc eco.c -o eco
```

e para ver o efeito, podemos fazer, por exemplo:

```
./eco olá
```

Neste exemplo temos:

`#include <stdio.h>` directivas de pré-processamento. Neste caso trata-se da inclusão de uma biblioteca do sistema, a biblioteca `stdio.h` que trata das entradas e saídas e que é necessária para poder ter acesso aos comandos `scanf` e `printf`. Mais pormenores em (ver Apêndice B).

Comentários a documentação interna do programa é feita através da escrita de *comentários*. No *C* podemos ter «**linhas de comentários**», tudo o que esteja à direita de um par de barras oblíquas para a direita («`//`») e até ao fim dessa linha. Podemos ter também «**blocos de comentários**» tudo (várias linhas) o que estiver entre o par de símbolos «`/*`» e o par «`*/`».

`argc`, `argv` são usualmente designados por «argumentos da linha de comandos». `argc` - «argument count» dá-nos o número de argumentos, `argv` «argument values», é uma lista («array») contendo esses mesmos argumentos. Todos estes valores são o resultado da interacção entre o interpretador de comandos do sistema operativo e o programa. O nome do próprio comando também conta, e também é guardado em `argv`. Temos então que no exemplo acima teria-se-ia que `argc` = 2, `argv[0]` = «./eco», `argv[1]` = «olá».

A «lista» (array) `argv` é um exemplo de estrutura de dados composta que será discutida mais à frente (§ 3.8).

`return` define o valor de saída que é «passado» ao interpretador de comandos do sistema operativo. Por norma comummente aceite o valor de saída zero indica uma terminação sem erros, um valor diferente de zero, dá-nos um código de erro. Os códigos de erro são próprios de cada programa. Este valor é devolvido ao interpretador de comandos do sistema sendo que, se nada for feito, o código é simplesmente ignorado.

3.5 Instruções Simples

Como já foi dito acima um programa em *C* é constituído por uma função especial, `main` e, eventualmente, por outras funções. Um programa é uma sequência linear de transições de estado. As variáveis definem o estado do programa.

Já vimos como definir o estado de um programa em termos de estruturas de dados simples (ver-se-á mais à frente as estruturas compostas (§3.8) e dinâmicas (§3.8.2). Falta-nos ver: como alterar o estado, como definir a sequência de linear de transições de estado e quais são as instruções básicas que podem ser incluídas na sequência linear de transições de estado.

3.5.1 Sequência Linear de Instruções

O corpo de uma função é então definido por uma lista de instruções separadas por ponto-e-vírgula ≪;≫. Algo como I_1; ou $I_1; I_2; \ldots; I_n$;.

É também possível agrupar um conjunto de instruções num só "bloco de instruções", o qual passará a contar com uma única instrução (composta). A forma de o fazer é através da utilização dos símbolos de agrupamento, por exemplo $\{I_1; I_2; \ldots; I_n;\}$.

3.5.2 Instrução de Atribuição

A instrução de atribuição é uma instrução fundamental numa linguagem como o *C* dado que é através dela que se acede ao estado do sistema e/ou se modifica o mesmo.

A sintaxe é a seguinte:

instrução de atribuição ::= <identificador_variável> `=` <expressão> `;`

Por exemplo:

```
x = x * y / 3;
```

A sua leitura tem de ser feita no tempo tendo em conta o estado do programa antes da sua execução e o estado do sistema após a sua execução.

Considere-se que o seguinte excerto de um programa em *C*.

```
int x,y=3;
x = 7;
x = x * y / 3;
```

A leitura no tempo destas instruções é a seguinte:

Pré	Instrução	Pós
∅	`int x,y=3;`	x ??? y 3
x ??? y 6	`x = 7;`	x 7 y 6
x 7 y 6	`x` (pós, 14) = `x * y / 3;` (pré, $7 \times 6/3$)	x 14 y 6

É de notar que a as inicializações no caso das instruções de declarações de variáveis não são mais do que uma simplificação de duas acções distintas, uma criação de um variável, a definição do seu valor através de uma instrução de atribuição. De igual modo a instrução de leitura também «esconde» um(ou mais) atribuição(ões) dos valores lidos às variáveis que estão a ser usadas.

Por seu lado a escrita de resultados pode ser vista como o lado direito de uma instrução de atribuição, sendo que o valor obtido não vai alterar o estado mas sim ser «escrito» no dispositivo de saída seleccionado.

Nota importante: é importante ter em conta que na linguagem *C*, assim como nas derivadas do *C*, a instrução de atribuição tem um valor final (a própria instrução) que é igual ao valor calculado no lado direito e atribuído à variável do lado esquerdo. Este efeito colateral da instrução de atribuição pode ser usado para, por exemplo, fazer uma atribuição em cadeia, `x = y = 23;`, tem como efeito que y toma o valor de 23, como por efeito colateral a instrução `y=23` tem 23 como valor final, então x toma, também o valor de 23;

Em *C* tem-se acesso a um conjunto vasto de abreviações para instruções de atribuição. A sintaxe é a seguinte:

abreviação de operação/atribuição ::= <identificador_variável> <operador>`=` <expressão> `;`
abreviação de pós-incremento/decremento ::= <identificador_variável><operador><operador>`;`
abreviação de pré-incremento/decremento ::= <operador><operador><identificador_variável>`;`

Por exemplo

Abreviação	Instrução
`x += n`	`x = x + n`
`x++`	`x = x + 1`
`++x`	`x = x + 1`

Nos dois últimos casos a ordem pela qual aparecem os dois operadores, à direita ou à esquerda do operando, pode ser significativa. No caso em que estas abreviações são usadas no contexto de uma expressão, então no primeiro caso, `x++`, o valor de `x` no estado actual do sistema é usado para o cálculo é somente após a conclusão do cálculo é que o valor de `x` é incrementado, no outro caso, `++x`, o valor de `x` é incrementado e é esse o valor que é usado para o cálculo da expressão.

Um exemplo muito simples em que essa distinção altera o resultado final.

```
y = 4;
x = y++;
```

e

```
y = 4;
x = ++y;
```

No primeiro caso os valores finais de `x` e `y` são respectivamente 4 e 5. No segundo caso os valores finais são ambos iguais a 5.

3.5.3 Condicionais

As instruções condicionais dão no a possibilidade de ramificar (dividir) a sequência de instruções em diferentes ramos consoante o valor lógico de uma proposição.

De manhã podemos dizer, "se as provisões meteorológicas apontarem para chuva levo o guarda-chuva, senão deixo-o em casa", temos que se vai tomar uma de duas (mas não ambas) acções consoante o valor lógico da proposição "as provisões meteorológicas apontarem para chuva", se for verdadeira faço uma acção, se for falsa faço outra.

Nas linguagens de programação as instruções condicionais são, em geral, de dois tipos: a separação binária, o decidir entre uma de duas possibilidades consoante o valor lógico de uma dada proposição; a separação por casos, em que dado o valor de uma expressão numérica, de um tipo enumerável, se vai fazer um de entre múltiplos casos possíveis.

Vejamos como é que a linguagem *C* implementa este tipo de instruções.

Separação binária

A separação binária em *C* tem duas variantes "**se** Proposição **então** Instrução" e "**se** Proposição **então** Instrução1 **senão** Instrução2".

instrução condicional ::= `if ( ` <proposição> `)` <instrução> `;`
instrução condicional ::= `if ( ` <proposição> `)` <instrução> `else` <instrução> `;`

Que correspondem aos seguintes dois diagramas de fluxo[1] da figura 3.3.

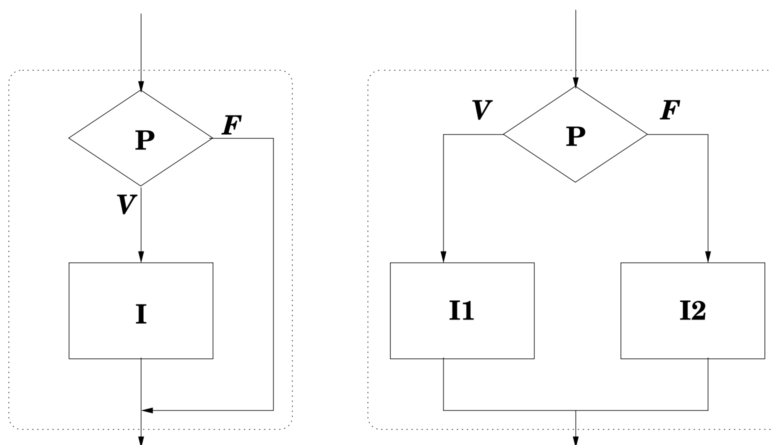


Figura 3.2: Separação binária

Por exemplo:

```
if (x<0)
  absX = -x; // x < 0
else
  absX = x;   // ~(x<0) <=> x >= 0
```

Não há restrições ao tipo de instrução que se pode usar na instrução contida nos ramos da instrução condicional.

[1] Os diagramas de fluxo permitem estabelecer o funcionamento, fluxo de instruções, de um programa através da indicação das diferentes ramificações que a sequência de instruções pode tomar.

```
if (P1) {
  I1; // P1 = V
}
else {
  if (P2) {
    I2; // P1 = F e P2 = V
  }
  else {
    I3; // P1 = F e P2 = F
  }
  I4; // P1 = F
}
```

Na construção de uma instrução condicional deve-se ter o cuidado de não deixar nenhum caso sem tratamento. Por exemplo (exercício 3.5.3), para o cálculo dos valores de uma função definida por ramos:

$$f(x) = \begin{cases} e - e^{\cos x} & \text{se } x \in [0, 2\pi[\\ \log \cos x & \text{se } x \in [-2\pi, -\frac{3}{2}\pi[\bigcup] - \frac{\pi}{2}, 0[\end{cases}$$

ter-se-á a tentação de usar uma instrução condicional somente com dois ramos, correspondentes aos dois ramos da definição da função.

```
if ((x>=0) && (x<(2*M_PI)))
  fx = exp(x)-exp(cos(x));
else if (((-2*M_PI<=x)&&(x<-3/2*M_PI))||((-M_PI/2<x)&&(x<0)))
       fx = log(cos(x));
```

Se do ponto de vista matemático esta definição faz sentido (a função cos é uma função periódica), do ponto de vista da implementação em C ela deixa de fora os valores de x que não pertençam ao intervalos definidos. Devemos acrescentar um terceiro caso que complete a gama de valores possíveis para x.

```
if ((x>=0) && (x<(2*M_PI)))
  fx = exp(x)-exp(cos(x));
else if (((-2*M_PI<=x)&&(x<-3/2*M_PI))||((-M_PI/2<x)&&(x<0)))
        fx = log(cos(x));
     else
        printf("Erro: valor de x incorrecto\n");
```

Como se vê pelos exemplos anteriores a proposição P não é necessariamente uma proposição atómica, muito pelo contrário, pode ser constituída por um número (teoricamente) não limitado de proposições combinadas pela conectivas lógicas da negação, $!P$, da conjunção, $P_1 \&\& P_2$, e da disjunção, $P_1 || P_2$. A avaliação do valor lógico final é feita de acordo com as tabelas de verdade da lógica proposicional (Mendelson, 1968).

É de notar que o C adopta um avaliação inteligente, *lazy evaluation*, das expressões lógicas de forma a evitar efectuar cálculos desnecessário, ou seja: a avaliação de uma conjunção pára assim que uma das proposições tenha o valor de verdade ≪Falso≫. De forma correspondente a avaliação de uma disjunção pára assim que o valor de uma das proposições tenha o valor de verdade ≪Verdade≫. Isto é a avaliação de uma proposição pára assim que o seu valor final é conhecido por se ter obtido o valor absorvente da conectiva lógica que se está a considerar.

Deve-se ter em conta este processo de cálculo do C, o qual pode ser útil em algumas situações, como se verá mais à frente (ver §3.8).

Além do condicional existe em C uma instrução que nos dá a separação por casos. Esta instrução não é estritamente necessária, no entanto em situações em que há muitos casos para tratar, a sua utilização pode simplificar a escrita de um programa.

Separação por Casos

Para as situações em que se tem uma expressão com valores do tipo inteiro, a qual pode assumir um conjunto alargado de casos, o C possuí uma instrução que permite fazer a separação por casos de uma forma simples.

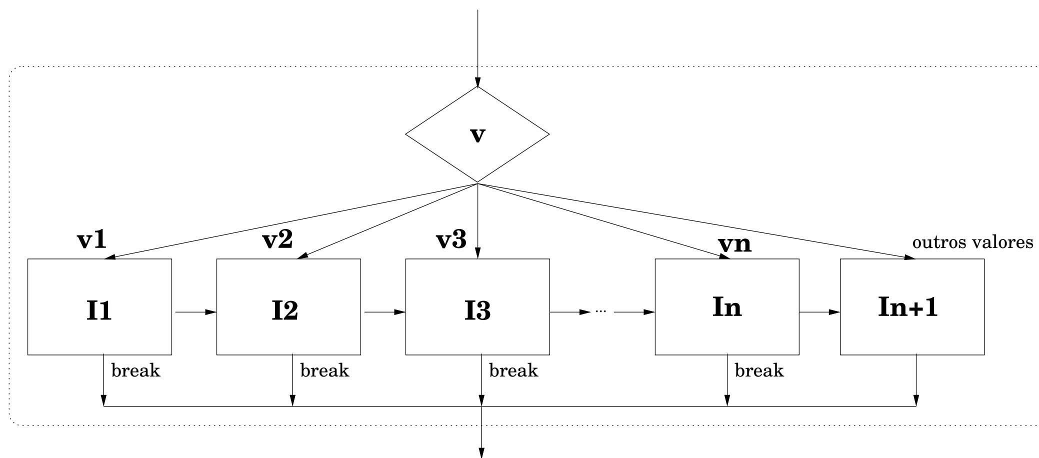


Figura 3.3: Separação por Casos

```
switch (<expressão_inteira>) {
 case <constante1>:
     I1; I2; ...; In;
     break;
 case <constante2>:
     ...
 default:
     ...
}
```

temos então:

```
switch (<expressão_inteira>) { ... }
```

É a instrução que faz a separação por casos, sendo que os casos são separados consoante o valor, que tem de ser do tipo inteiro, da expressão entre parêntesis.

```
case n: In1; In2; ...
```

Define quais as instruções a executar para o caso em que a expressão inteira toma o valor n. No caso do C, a exemplo do que já acontecia na linguagem C, após a execução de todas as instruções referentes a este caso a execução do programa continua na instrução imediatamente a seguir ao caso em que se está. Este efeito de continuar no próximo caso («fall through»)

tem o efeito de se poder especificar o mesmo conjunto de instruções para diferentes casos. Por exemplo:

```
case n1:
case n2:
case n3: In1; In2; ...
```

para todos os valores da expressão iguais a `n1, n2` e `n3` vão se executar as instruções `In1; In2; ...`

Para contrair esse efeito de «continuar para o próximo caso», é necessário incluir a instrução `break` no fim da sequência de instruções referentes ao caso em questão. O efeito da instrução `break` é o de «quebrar» a continuidade da instrução `switch` e saltar de imediato para o fim da mesma.

O caso `default` (por omissão) é opcional, o seu efeito, a exemplo do que já foi dito acima para a instrução `if`, é o de fechar a instrução de selecção de casos com um caso que se verifica sempre que nenhuma dos outros casos se verificou. O caso `default` deve ser, por razões óbvias, o último caso a ser considerado, e deve ser sempre considerado para que não fiquem casos por considerar.

3.5.4 Ciclos

Os ciclos são, como o próprio nome indica, instruções repetitivas e para os quais é importante identificar uma condição de paragem para não se entrar num ciclo (infinito).

Os ciclos são importante sempre que se identifique uma tarefa resolvível por sucessivas aplicações de um dado conjunto de passos. Por exemplo o cálculo do somatório de uma dada função.

$$\sum_{i=1}^{n} f(x_i) = \underbrace{\underbrace{\underbrace{\underbrace{f(x_1)}_{i=1} + f(x_2)}_{i=2} + f(x_3)}_{i=3} + \cdots + f(x_n)}_{i=n}$$

Para cada ciclo deve-se identificar:

O caso inicial neste caso, um somatório, podemos fazer `somatorio = 0`, dado zero ser o elemento neutro da adição.

O invariante isto é a tarefa repetitiva. No caso presente podemos dizer que a cada passagem do ciclo temos o valor do somatório para $i = k-1$ ($\sum_{i=1}^{k-1}$) ao qual se vai somar a parcela de ordem k. `somatorio = somatorio + f(x_k)`.

O caso de paragem o valor final para o qual a variável (ou expressão) de controlo do ciclo vai convergir. No caso do somatório ter-se-ia `i=n` e `somatorio = somatorio + f(x_n)`.

Em C existem três tipos distintos de instruções de ciclo:

`while (P) I;` enquanto P faz I, isto é enquanto P é verdadeira executa a instrução I (lado esquerdo da Figura 3.4). Com este tipo de ciclo temos que o invariante é executado zero ou mais vezes.

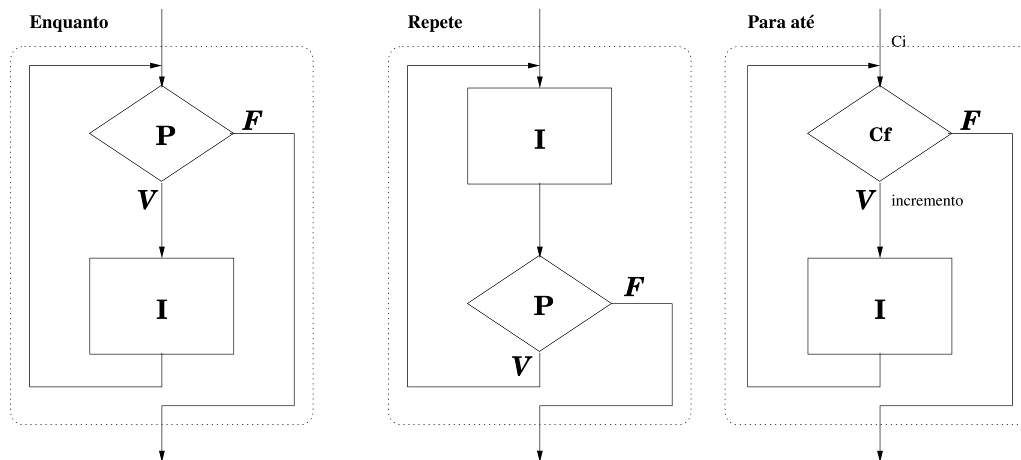


Figura 3.4: Ciclos ≪`while` (P) I≫, ≪`do` $I_1;\ldots;I_n$ `while` (P)≫ e ≪`for` $(C_i;C_f;$inc$)$ I≫

`do I1;...;In; while (P);` repete I1;...;In; enquanto P, isto é, repete a instrução I enquanto a condição P for verdadeira (lado direito da Figura 3.4). Com este tipo de ciclo temos que o invariante é executado uma ou mais vezes.

`for (Ci;Cf;Inc) I;` Este ciclo é uma versão compacta do ciclo `while`. Temos que: `Ci` estabelece a condição inicial; `Cf` a condição final; e `Inc` é uma instrução que deve assegurar a convergência desde a condição inicial à condição final. Este ciclo é executado zero ou mais vezes.

É de ter em conta que o ciclo `for` tem em *C* uma semântica bem diferente de muitas outras linguagens, nomeadamente o *Pascal* (Gottfired, 1994; Welsh & Elder, 1979),e o *FORTRAN* (Adams *et al.*, 1997). Nestas linguagens as condições iniciais e finais são avaliadas ao início, dessa avaliação saí o número de iterações a efectuar e o incremento que a variável de controlo (de um tipo enumerável) vai tomar, seguindo-se o executar repetidas vezes da instrução contida no ciclo sem que as condições iniciais e finais e de incremento voltem a ser avaliadas. Como efeito colateral tem-se que a variável que é afectada pelas condições iniciais e finais pode ser usada dentro do ciclo mas a sua modificação não tem nenhum significado no funcionamento do mesmo.

No caso do *C* não é isso que acontece, a cada iteração a condição de paragem `Pf` é verificada e a instrução de `Inc` executada. Temos então que, ao contrário do que foi dito acima, no caso do *C* a(s) variável(eis) presentes em `Pf` podem ser usadas e alteradas dentro do ciclo a sua eventual modificação terá uma consequência directa na condição de paragem do ciclo.

Vejamos como calcular o valor da função factorial utilizando as diferentes instruções de ciclo

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases}$$

Este definição por recorrência do factorial não nos serve (ver-se-á mais à frente como tratar estes casos), é necessário poder definir o que se pretende calcular como um processo que se repete de um valor inicial até um valor final.

$$\begin{aligned} 0! &= 1 \\ n! &= \underline{\underline{\underline{1} \times 2} \times 3} \times \ldots \times n = \Pi_{i=1}^{n} i \end{aligned}$$

Temos então:

Caso inicial $0! = 1! = 1$

```
factorial = 1;
```

invariante $i! = (i-1)! \times i$, com $1 \leq i \leq n$

```
factorial = factorial * i;
i = i + 1;
```

caso final (caso de paragem) $i = n$.

Utilizando o ciclo `while`

```
factorial = 1; // caso inicial 0!=1!=1
i = 2;
while (i <= n) { // condição de paragem i=n
  factorial = factorial * i;  // invariante i!=(i-1)*i
  i = i + 1;                  // incremento de i
}
```

Utilizando o ciclo `do while`

```
factorial = 1; // caso inicial 0!=1
i = 1;         // este ciclo é executado pelo menos uma vez
do {
  factorial = factorial * i;  // invariante i!=(i-1)*i
  i = i + 1;                  // incremento de i
} while (i <= n); // condição de paragem
```

Utilizando o ciclo `for`

```
factorial = 1; // caso inicial 0!=1!=1
for (i=2 ; i <=n ; i=i+1) // c.inicial; c.final; incremento
  factorial = factorial * i;
```

3.6 Modularidade Procedimental

Como já foi dito atrás (§2.2) "a modularidade é a capacidade para decompor casos complexos numa colecção de casos mais simples (chamados módulos) e em regras de composição" sendo que o suporte à modularidade é uma característica muito importante numa linguagem de programação.

Em termos do algoritmo a modularidade é dado pela divisão de um algoritmo em sub-algoritmos. Em *C* a modularidade é dada pela possibilidade que a linguagem dá ao programador de dividir um dado programa em funções independente entre si. As ligações entre funções é efectuada pelos argumentos e pelo resultado das mesmas.

3.6.1 Funções

Como já foi dito acima um programa em *C* é constituído por uma função especial, `main`, ponto de início e fim (em condições normais) do funcionamento do programa e, eventualmente, por outras funções. A organização física destas funções é feita em ficheiros sendo que é necessário declarar algo antes de o poder usar, isto é, a ordem pela qual se declaram as funções num dado ficheiro não é irrelevante. Na secção 3.7 falar-se-á desta questão mais em detalhe, nomeadamente da organização de um programa em múltiplos ficheiros.

Em *C* uma função tem uma semântica (significado) à partida igual às funções matemáticas, isto é:

- n argumentos;
- 1 resultado (associado ao nome da função);
- uma sintaxe de chamada, $f(x)$, igual à usualmente usada em matemática.

Em termos operacionais temos que acrescentar:

- ligação entre os parâmetros (definição) e os argumentos (chamada) feita por cópia do valor;
- é possível definir e usar uma estrutura de dados local (e não global) à função.

Em termos sintácticos temos que considerar dois momentos distintos: a declaração da função; e a sua utilização.

Declaração, isto é a definição/construção da função através da escrita do algoritmo que, dados os valores de entrada, produz o resultado esperado.

```
<tipo_saída> <nome_função> (<lista_declarações_parâmetros>) {
 <corpo_da_função>
}
```

O corpo da função é uma sequência de instruções simples. O *C* não admite funções dentro de outras funções, sendo que deve existir pelo menos uma instrução `return(<expressão>)` que defina o valor de saída (resultado) da função.

A utilização de uma função ocorre no âmbito do cálculo de uma expressão. Isoladamente, ou com outros elementos. Por exemplo:

```
 x = sin(x)*3;
```

Exemplo de definição e utilização de uma função.

```
 /*
  * Potência natural de um real
  * -> x (real), expoente (natural)
  * <- x^expoente
```

```
  */
double potencia(double x, int expoente) {
 int i;
 double pot=1; // x^0 = 1
 for (i=1; i<=expoente; i++)
   pot = pot*x;
 return pot;
}
```

É de notar a escrita da instrução `return` sem se utilizarem parêntesis. Isso é possível desde que não haja ambiguidade, por exemplo, um só valor após o identificador `return`. Em caso de dúvida pode-se sempre colocar os parêntesis.

É importante acrescentar a especificação da função como comentário, logo após o cabeçalho da função. Uma descrição breve do que a função é suposto fazer, os seus parâmetros de entrada (pré-condição) e o resultado esperado (pós-condição).

A utilização desta função será feita num outro ponto do programa: ou numa outra função; ou na função `main`. Por exemplo:

```
 printf("%f^%d=%f\n",x,n,potencia(x,n));
```

Parâmetros e Argumentos

A ligação entre módulos é feita pela ligação entre os argumentos (utilização) e os parâmetros (definição). Na linguagem *C* essa ligação é feita por cópia dos valores dos argumentos para os parâmetros.

Para melhor compreender como se processa a ligação entre módulos recuperemos o exemplo da função potência, definindo com ela um programa completo (a numeração à esquerda é só para referência).

```
0 double potencia(double base, int expoente) {
1 int i;
2 double pot=1; // x^0 = 1
3 for (i=1; i<=expoente; i++)
4   pot = pot*base;
5 return pot; }

0 int main() {
1  double x=7.4, res;
2  int n=2;
3  res = potencia(x,n);
4  printf("%f\n",res);
5  return 0; }
```

Vejamos então, passo a passo, como é que o estado do programa se vai alterando (ver Figura 3.5).

Na linguagem *C* a ligação entre sub-programas (funções) é sempre feita através de uma *passagem por valor* (ver Figura 3.6). Em contraponto a esta forma de ligação temos que no *FORTRAN* as ligações são somente feitas *por referência* e em *Pascal* as ligações podem ser de ambos os tipos.

O que são exactamente estes dois modos de ligação, quais as diferenças e como é que isso afecta a construção de um programa em *C*?

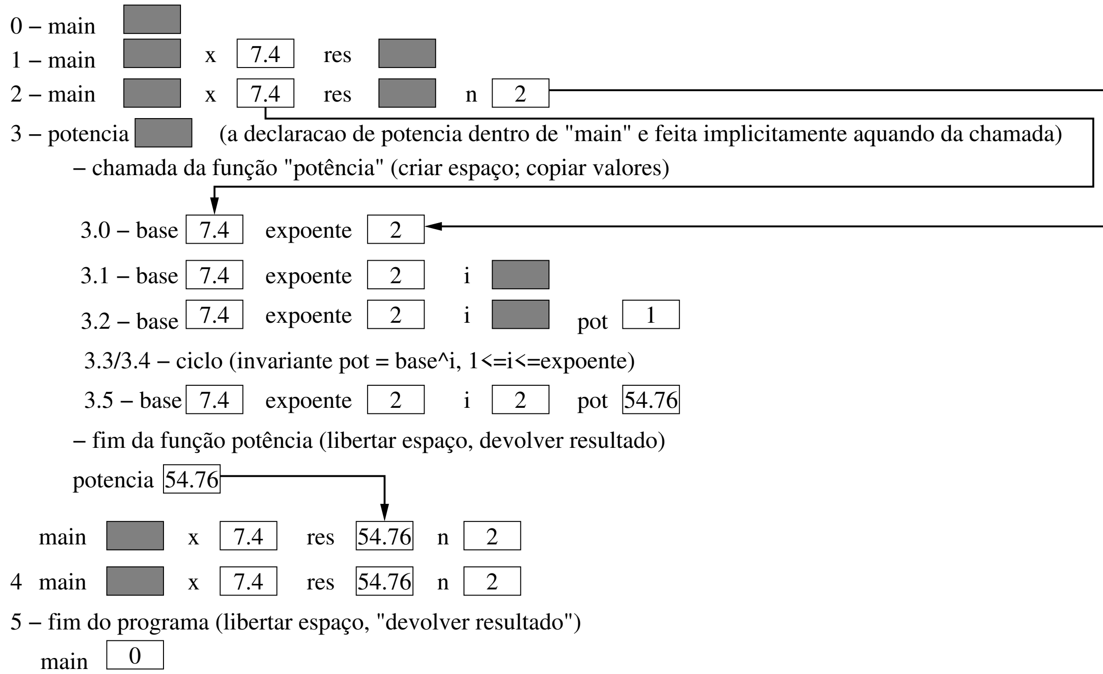


Figura 3.5: Ligação entre Argumentos e Parâmetros

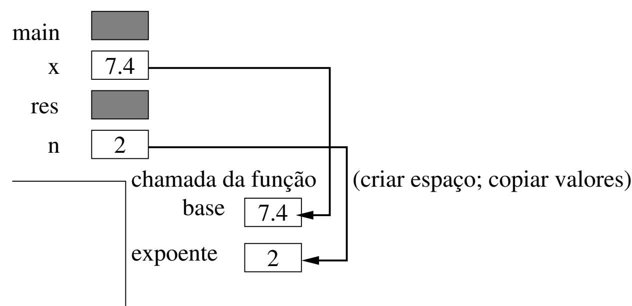


Figura 3.6: Ligação por Valor – O estado do programa não se altera

Passagem por Valor vs. Passagem por Referência. Na passagem (de valores, ou ligação entre sub-programas) por valor os argumentos e os parâmetros são unidades de memória distintas, aquando da chamada do sub-programa é feita a cópia dos valores dos argumentos, no programa de chamada, para os respectivos parâmetros, no sub-programa que está a ser chamado. Após esta cópia, não existe mais nenhuma ligação entre estes dois conjuntos de posições de memória.

Em contraponto a isso temos a passagem por referência.

Passagem por referência. No caso da passagem por referência os argumentos e os parâmetros são nomes distintos para a mesma posição de memória. Isto é aquando da chamada do sub-programa é feita a cópia das referências para a memória (ponteiros) respeitantes aos argumentos, no programa de chamada, para os respectivos parâmetros, no sub-programa que está a ser chamado. Após esta cópia, os parâmetros não são mais do que um nome que se referem (apontam) para as posições de memória dos argumentos respectivos (ver Figura 3.7).

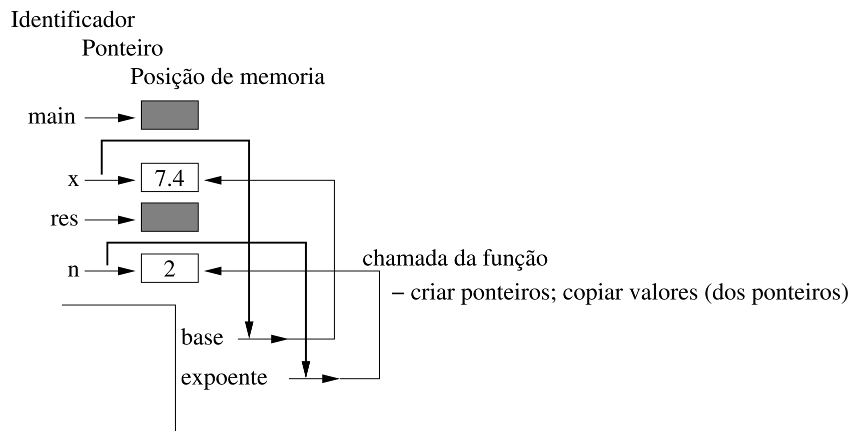


Figura 3.7: Ligação por Referência – O estado do programa de chamada pode ser alterado

Durante a execução da função, o estado do programa de chamada pode ser afectado. Sempre que os parâmetros são usados no sub-programa o estado do programa de chamada é usado ou alterado, conforme as circunstâncias (cálculo de uma expressão, ou instrução de atribuição).

Quais são as implicações, vantagens, desvantagens desta aproximação (só fazer a passagem por valor) da linguagem *C*?

Vantagens Dado que os sub-programas em *C* são só do tipo função a ligação por valor é a mais correcta do ponto de vista formal. Após a cópia dos valores, no momento da chamada, não há mais nenhuma interacção com o programa de chamada, que não seja a devolução do valor final da função. Não há efeitos colaterais.

Desvantagens Mas, e se se pretender que haja modificações nos argumentos? Por exemplo as funções de leitura de valores, por exemplo uma função que faça a troca dos valores entre os seus dois argumentos. Isto é se se pretender que o sub-programa tenha um comportamento de um módulo que recebe vários argumento e devolve vários resultados, e que permite que alguns desses resultados sejam alterações ao valor dos argumentos. Nestes casos é necessário estabelecer a ligação por referência.

Uma outra desvantagem prende-se com as estruturas de dados do tipo tabela (ver secção 3.8.1). Nestes casos a cópia de uma tabela, com a subsequente criação de uma variável local da mesma dimensão, pode-se revelar muito penalizadora, ou, em última instância, ser impeditiva do funcionamento do programa. Como veremos mais há frente as tabelas, em *C* são sempre passadas por referência.

Como em *C* não existe o mecanismo de passagem de valores por referência, então, para obter esse efeito, é necessário passar por valor as referências. Isto é em vez de ter como argumento uma variável `x` do tipo inteiro (por exemplo), coloca-se como argumento a referência respectiva `&x`. Como parâmetro coloca-se uma variável do tipo referência (ponteiro) para um inteiro.

Vejamos um exemplo: um programa para efectuar a troca do valor de duas variáveis do tipo inteiro.

Primeiro a função que faz a troca dos valores:

```
void troca(int *a, int *b) {
 int aux;
 aux = *a;
 *a = *b;
 *b = aux;
}
```

temos então que ter variáveis do tipo ponteiro para inteiros. Para poder manipular os valores associados com as mesmas temos de recorrer ao operador `*`. Como a função não é suposto devolver nenhum resultado o seu tipo de saída é `void` que é um tipo especial em *C*, o tipo vazio. Não há nenhuma instrução de `return` pela mesma razão.

De seguida o programa de chamada:

```
int main() {
 int x=3,y=7;

 troca(&x,&y);
 printf("%d%d\n",x,y);
 return 0;
}
```

aquando da chamada da função `troca` os argumentos têm de ser passados por referência, ou mais correctamente, os argumentos, a ser passados por valor, têm de ser as referências correspondente às variáveis que queremos manipular. Desta forma '`x`' e '`a`' não são mais do que dois nomes referenciando a mesma posição de memória, todas as manipulações que se fizerem sobre '`a`' serão como sendo feitas através de '`x`'.

Como a função `troca` não tem um valor de saída definido a sua utilização é feita independentemente do cálculo de uma qualquer expressão (que é o caso usual de utilização de funções).

Âmbito das Variáveis

No que se disse até aqui sobre sub-programas em *C* ficou sub-entendido que o estado de um sub-programa é independente do estado do programa de chamada. Isto é parcialmente verdadeiro, iremos ver de seguida as questões relacionadas com o âmbito das variáveis, isto

é, quando e como é que podemos aceder aos seus valores, nomeadamente as questões sobre variáveis globais e variáveis locais.

Por âmbito de uma variável entende-se o grupo de instruções aonde se consegue aceder a um determinado valor associado a um identificador (variável).

O *C* permite que uma variável seja declarada em qualquer ponto do(s) ficheiro(s) que contêm as funções, nomeadamente a função `main`, que constituem o programa. O âmbito da variável vai desse ponto em diante, sujeito às seguintes restrições:

Variáveis Globais as variáveis que são declaradas fora de uma qualquer função têm um âmbito global, isto é, são utilizáveis em qualquer ponto do programa. Como excepção a esta regra temos a possibilidade de definir uma variável com o mesmo nome mas contida numa função, esta segunda definição, local, sobrepõe-se à global.

Variáveis Locais as variáveis que são declaradas dentro de uma qualquer função têm um âmbito local, isto é, só são válidas na função em que são declaradas.

Não Re-declaração não é possível re-declarar, dentro do mesmo nível (local ou global), uma dada variável.

Parâmetros de Funções (estáticos) os parâmetros de uma função definem variáveis declaradas localmente e cujos valores iniciais são os valores que recebem dos argumentos aquando da chamada. As variáveis locais (parâmetros) são completamente independentes dos argumentos (variáveis de uma outra função).

Parâmetros de Funções (dinâmicos) os parâmetros de uma função que sejam do tipo referência (ponteiro), definem variáveis (do tipo ponteiro) declaradas localmente. As variáveis locais não são mais do que um segundo conjunto de nomes correspondente aos argumentos (variáveis de uma outra função) e seus valores

A este conjunto de regras deve-se acrescentar um conjunto de boas práticas:

Não Declaração de Variáveis Globais a razão de tal atitude é simples, a metodologia de programação estruturada e descendente define os programas como colecções de módulos nos quais a comunicação entre os módulos é feita de forma a que possamos ter encapsulamento da informação e re-utilização do código. As variáveis globais vão de contra a estes propósitos.

Declarações no Inicio das Funções embora seja possível declarar variáveis em qualquer ponto de uma função o concentrar-se as declarações logo nas primeiras linhas de cada função permite ter de imediato uma compreensão de qual é o conjunto de variáveis que vai definir o estado desse sub-programa.

Recorrência

Em *C* é possível a uma função chamar-se a si própria, a este tipo de utilização designa-se chamada recorrente (ou recursiva).

A recorrência define um processo repetitivo, como tal é necessário definir-se:

Caso Inicial conjunto de valores dos argumentos válidos (pré-condição);

Caso Recorrente caso repetitivo, deve assegurar que o cálculo pretendido é efectuado (invariante da recorrência) e que o processo converge para a condição de paragem.

Condição de Paragem caso não repetitivo que deve assegurar que a recorrência para.

A necessidade de que, de um caso de inicial se caminhe em direcção a um caso final (condição de paragem) num número finito de passos, leva a que a recorrência só é possível sob conjuntos de valores (para os argumentos) que possuam uma boa ordenação. Em termos do *C* isso quer basicamente dizer que o argumento de uma função recorrente que assegura a convergência para o caso final deve ser do tipo inteiro (`int`).

Cada chamada da função implica o criar de um novo conjunto de variáveis locais. É necessário ter isto em conta dado que o mesmo pode ser muito pesado caso haja muitas chamadas por recorrência e/ou as estruturas locais sejam de grande dimensão.

O exemplo clássico de função recorrente é a função factorial.

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

A sua implementação em *C* é quase imediata:

```
long int factorial(int n) {
  if (n==0) return 1
  else return (n*factorial(n-1));
}
```

como pré-condição temos que o n tem de ser um número natural, isto é, $n \geq 0$.

Um outro exemplo também muito usado é o de um programa que faça a inversão de uma frase lida do terminal. Este é um caso interessante para analisar como se processam as sucessivas chamadas recorrentes com as consequentes declarações de variáveis locais.

Vejamos o programa:

```
#include <stdio.h>

void inverte () {
  char c;
  c = getc(stdin);
  if (c != '.') {
    inverte();
    printf("%c",c);
  }
}

int main() {
  printf("Escreva uma frase (termine com um '.'): ");
  inverte();
  printf("\n");
  return 0;
}
```

Analisando o funcionamento deste algoritmo do ponto de vista das variáveis locais das sucessivas chamadas recorrentes verifica-se que, a inversão da ordem das letras na palavra decorre automaticamente da forma como a recorrência se processa, primeiro, na fase da leitura,

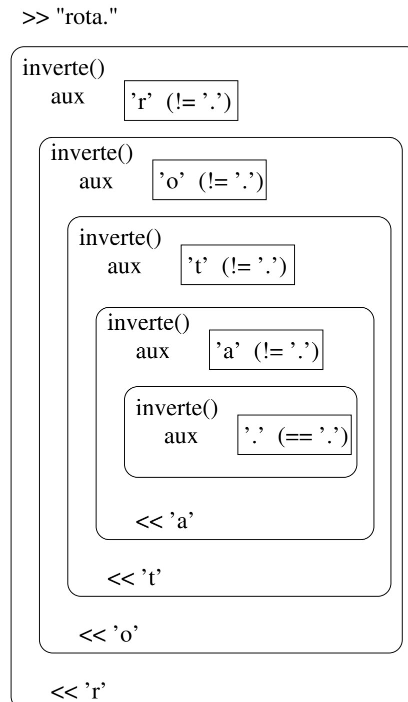


Figura 3.8: Variáveis Locais numa Chamada Recorrente

indo do exterior para níveis cada vez mais profundos, depois, na fase da escrita, do nível mais profundo para o exterior (ver Figura 3.8).

É importante notar que a cada chamada recorrente é criada uma nova variável `aux`, no exemplo apresentado isso significou um total de cinco variáveis do tipo `char`. O que aqui não é significativo pode ser determinante noutras situações, não só no número total de chamadas recorrentes (por exemplo, neste caso a escolha de uma frase muito comprida) como nas estruturas de dados a utilizar (neste caso, se em vez de optar pela declaração `char a` se se tivesse optado, por exemplo, pela declaração `char a[1000]`).

Um nota final, a função `main` não é passível de chamadas recorrentes.

3.7 Estruturação de um Programa

Como já foi dito no capítulo 2 a metodologia de programação estruturada e descendente caracteriza-se por decompor o problema global em sub-programas específicos (por tarefa bem determinada) e pela sua correcta combinação.

Utilizando a linguagem *C* é usual dividir cada um dos módulos (uma ou mais funções com um fim específico) em ficheiros separados.

3.7.1 Separação de um Programa em Múltiplos Ficheiros

Esta separação em estruturas funcionais diferentes tem, em *C*, uma correspondência física. A separação de um programa em múltiplos ficheiros.

Em primeiro lugar a convenção em *C* é a de separar a componente das declarações da componente das implementações.

Declaração/Especificação: por declaração, ou especificação, de uma função entende-se a explicitação dos seus argumentos e do seu resultado, tanto em termos de número e tipo dos argumentos como do tipo do resultado.

Por exemplo.

```
long int factorial(int);
```

Basta-nos para saber que a função factorial necessita de um argumento do tipo inteiro e que produz um resultado também do tipo inteiro (mais especificamente do tipo ≪inteiro longo≫).

A especificação adiciona a isto as pré e pós-condições, isto é, quais são os valores aceitáveis para o(s) argumento(s) e qual é o valor do resultado que se espera para um dado argumento. Em *C* as pré e pós-condições não fazem parte da linguagem, podemos (devemos) incluí-las como comentários.

```
/*
 * Pré: n >= 0, n do tipo inteiro
 * Pós: factorial(n) = n!
 */
long int factorial(int);
```

Num programa em que haja um uso apropriado das estruturas da linguagem (sonegação da informação, não utilização de efeitos colaterais, não utilização de variáveis globais) a forma concreta como a função é implementada é independente da sua utilização.

Isto é, num dado programa, bastar-no-ia esta informação para utilizar a função factorial num outro ponto do programa.

Por outro lado para o compilador completar a sua análise léxico/gramatical do programa que chama a função factorial, a implementação desta é também irrelevante. Basta saber que os argumentos na chamada correspondem em número e tipo com os parâmetros na declaração, e que o tipo do resultado está de acordo com a variável, ou expressão, que vai receber esse valor.

Esta separação entre declaração e implementação possível em programas em *C* tem uma expressão física.

Por convenção, para um dado programa ≪exemplo≫ criar-se-ão dois ficheiros:

Ficheiro das Declarações (*headers file*) ficheiro com extensão ≪.h≫, `exemplo.h`.

Ficheiro das Implementações (*C file*) ficheiro com extensão apropriado à linguagem, por exemplo, ≪.c≫, `exemplo.c`.

Por exemplo, a construção de um programa para o cálculo do factorial de um número natural.

Num dado ficheiro `factorial.h` tem-se somente a declarações (cabeçalhos) da função factorial.

```
/*
 * Cálculo da função factorial
 * Pré: n >= 0, n do tipo inteiro
 * Pós: factorial(n) = n!
 */
long int factorial(int);
```

No segundo, ficheiro `factorial.c`, temos a implementação da função factorial.

```
long int factorial(int n) {
  if (n==0) return 1
  else return (n*factorial(n-1));
}
```

Finalmente ter-se-á o programa principal, o qual terá extensão ≪.c≫ dado se tratar de código, e que terá de conter a função `main`. Podemos designá-lo por `usaFactorial.c`.

```
#include <iostream>
#include "factorial.h"

int main() {
  int i;
  printf("Introduza um inteiro, maior ou igual a zero: ");
  scanf("%d",i);
  printf("%d! = %d\n",i,factorial(i));
}
```

É de notar a inclusão da directiva de compilação, `#include "factorial.h"`. Esta é necessária dado que estamos a usar a função, `factorial(i)`, sendo que não temos, neste ficheiro, nada em concreto sobre ela.

Para a fase análise léxico/gramatical só é necessário saber como usar a função, essa informação está contida no ficheiro dos cabeçalhos, e como tal a inclusão é referente ao ficheiro, `factorial.h`.

Para a fase da criação do código máquina (construção do programa executável, final) é necessário juntar ao nosso programa de chamada a componente da implementação da função factorial, isso é feito pelo compilador em dois passos:

Conversão do código *C* em código máquina: esta conversão é feita pelo compilador recorrendo à opção ≪`-c`≫.

```
cpp -c factorial.c
```

Este procedimento cria um ficheiro `factorial.o`, ≪`o`≫ de *object code*, isto é, código máquina.

Junção das diferentes componentes: de forma a criar o programa final (executável) é então necessário juntar todas secções do programa compilado (ficheiros ≪.o≫ e programa principal) num só ficheiro. Essa junção (*linking*) é feita pelo compilador explicitando todos os ficheiros a juntar

```
gcc factorial.o usaFactorial.c -o usaFactorial
```

A opção ≪`-o`≫ é meramente usada para dar um nome específico ao programa final. A convenção em Linux é que este não tenha extensão, outra convenção usual é usar a extensão ≪`.exe`≫.

Temos assim um programa dividido em três ficheiros, a divisão em três ou mais ficheiros deve ser ditada por uma organização funcional do código do programa. A forma como lidar com esta explosão no número de ficheiros a considerar pode, e deve, ser feita recorrendo a automatismos próprios do sistema operativo, veja a este propósito o apêndice B.

3.8 Estruturas de Dados Compostas

3.8.1 Estruturas de Dados Estáticas

O *C* possuí dois tipos de estruturas de dados pré-definidas: as tabelas e os registos.

Tabelas (estáticas)

As tabelas (*arrays*) são tabelas uni-dimensionais e homogéneas. Isto é são estruturas com uma só dimensão e de elementos todos do mesmo tipo.

Por exemplo um vector de inteiros:

```
int v[100];
```

neste caso temos uma tabela de inteiros com a capacidade de guardar 100 inteiros. O acesso aos elementos individuais da tabela é feita através de índices (de 0 (zero) até número de elementos-1), por exemplo, `v[4]`, dá-nos o 5º elemento da tabela.

Podemos declarar e inicializar elementos deste tipo.

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

Neste caso é usual não especificar o número de elementos da tabela, o mesmo é calculado automaticamente pelo compilador pela simples contagem dos elementos da lista de valores iniciais. Se se optar por explicitar o números de elementos da tabela então esse valor tem de coincidir com o número de elementos na lista de inicialização.

Uma variável do tipo tabela é, sempre, uma variável do tipo ponteiro. Este facto é importante quando se consideram funções. Por exemplo, pretende-se construir uma função que dado um vector de elementos inteiros o ordene (ver §5.2).

```
/*
 * Recebe um vector, assim como o num. de elementos e ordena-o
 * através do método Borbulhagem («Bubble sort»)
 */
int borbulhagem(int nElem, int v[]) {

  int i,aux;
  int ordenado;

  do {
    ordenado = 1;
    for (i = 0; i < nElem-1 ; i++) { // precorre o vector
      if (v[i] > v[i+1]) { // se necessário troca elementos
        aux = v[i];
        v[i] = v[i+1];
        v[i+1] = aux;
        ordenado = 0;
      }
    }
  } while (!ordenado); // repete até não haver mais trocas

  return 0;
}
```

A declaração do vector e chamada da função `borbulhagem` seria:

```
...
int main() {
  int nElem;
  int v[1000];

  ...
  borbulhagem(nElem,v);
  ...
}
```

Podemos desde já notar alguns pontos que de certa forma parecem colocar em questão o que foi anteriormente dito:

- Dado que o vector está a ser (aparentemente) passado por valor, o seu ≪valor≫ (isto é o seu conteúdo) não deveria ser alterado e como tal a ordenação não deveria ter consequências no programa de chamada (não é isso que acontece na realidade);
- A dimensão do vector não está a ser explicitada (`v[]`);

A primeira questão prendem-se com o facto já referido de que uma variável do tipo tabela é sempre (mesmo que isso não seja explícito) do tipo ponteiro. Como tal o vector está a ser passado ≪por referência≫ e tudo o que se fizer aos seus elementos reflecte-se no programa de chamada.

A segunda questão é respondida pelo compilador de C. Na chamada de uma função não é necessário explicitar a dimensão das tabelas. A dimensão é ≪dada≫ pelo programa de chamada. É de notar que se se optar por explicitar a dimensão da tabela (é sempre possível) a mesma tem depois de coincidir com a dimensão declarada no programa de chamada.

Tabelas n-dimensionais Podemos ter tabelas n-dimensionais se se declarar uma tabela com elementos do tipo tabela. Por exemplo `int matriz[10][10]`, dá-nos uma matriz de 10 por 10 de elementos do tipo inteiro;

Numa passagem de parâmetros do tipo tabela n-dimensional só o primeiro campo é que pode ficar por especificar, todos os outros têm de ser dados de forma explícita.

Para uma dada função de soma de duas matrizes as seguintes duas declarações são equivalentes:

```
void adicionaMatrizes(int nl, int nc, int m1[][10], int m2[][10],
                      int *resnl, int *resnc, int m3[][10])
```

e

```
void adicionaMatrizes(int nl, int nc, int m1[10][10], int m2[10][10],
                      int *resnl, int *resnc, int m3[10][10])
```

isto para declarações de matrizes do tipo visto acima. A declaração com as duas dimensões por expecificar não é válida em C.

Tabelas Dinâmicas As tabelas também podem ser declaradas de forma dinâmica, por exemplo, `int *v` pode ser vista como a declaração de uma tabela de elementos do tipo inteiro.

A vantagem deste tipo de definição é que a estrutura assim criada pode depois ter uma dimensão ajustada às necessidades, em contraponto com o outro tipo de declaração que fixa a dimensão da tabela. A desvantagem é que obriga à gestão da afectação da memória por parte do utilizador. Ver-se-à na secção 3.8.2 como proceder nesses casos.

Registos

Ao contrário das tabelas os registos definem uma estrutura de dados não homogénea. Isto é podemos criar um registo que integre um elemento do tipo inteiro e um outro do tipo real e um outro do tipo .... Os registos não definem, ao contrário das tabelas, estruturas com vários elementos todos do mesmo tipo.

Por exemplo: uma empresa pretende-se guardar e manipular a informação referente aos seus empregados. Em vez de colocar a informação referente aos empregados em estruturas não relacionas é vantajoso agrupá-las todas numa só estrutura. Temos então algo como:

```
struct empregado {
 char nome[60]; // nome do empregado
 int numero; // número de empregado
 int cc; // número de identificação (BI ou CC)
 int tipoId; // tipo de identificação (BI=1 ou CC=2 ou ...)
 int telefone;
};
```

Esta declaração cria um novo tipo de dados, o tipo ≪`struct empregado`≫. Caso se pretenda criar uma variável deste novo tipo podemos fazê-lo utilizando a sintaxe habitual:

```
struct empregado ep1,ep2;
```

Como podemos ver por este exemplo, um registo pode conter uma (ou mais) tabela. O contrário também é verdade. Por exemplo:

```
struct empregado listaEmpregado[100];
```

podia ser a estrutura aonde toda a informação referente aos empregados era guardada.

As variáveis do tipo registo podem ser declarados e inicializados de uma forma idêntica às variáveis do tipo tabela.

```
struct empregado eprg23 = {"Francisco",23,324543,1,230239239};
```

Na declaração de elementos de tipo `struct` é possível simplificar a definição introduzido um identificador de tipo, isto é definir um novo ≪nome≫ para o ≪novo≫ tipo que se está a definir.

Em *C* a definição de nomes de tipos é feito do seguinte modo:

```
typedef <definição_de_tipo> <identificador_nome_de_tipo>
```

por exemplo, traduzir os nomes dos tipos:

```
typedef int Inteiros;
typedef float Reais;
typedef char Caracteres;
```

No caso dos tipos simples, estas definições não são particularmente úteis. No caso dos registos a sua utilização torna mais claro a definição de elementos deste tipo.

```
typedef struct empregado {
 char nome[60]; // nome do empregado
 int numero; // número de empregado
 int cc; // número de identificação (BI ou CC)
 int tipoId; // tipo de identificação (BI=1 ou CC=2 ou ...)
 int telefone;
} Empregado;
```

Com a criação de um novo nome, `Empregado`, para o tipo `struct empregado`, pode-se simplificar a declaração de novas variáveis desse tipo, ter-se-ía:

```
Empregado ep1,ep2;
Empregado listaEmpregado[100];
Empregado eprg23 = {"Francisco",23,324543,1,230239239};
```

A forma de aceder a cada um dos membros de uma registo é feita através da utilização do operador '.' Por exemplo:

```
printf("%s%d%d\n",eprg23.nome,eprg23.numero,eprg23.cc);
```

Finalmente, no caso em que se queira definir um ponteiro para uma estrutura, por exemplo:

```
struct empregado* pep;
```

O acesso ao valor dos membros da estrutura tem de respeitar o que está definido para aceder aos valores apontados e o acesso aos diferentes membros da estrutura. Para esta definição ter-se-ia:

```
printf("%s%d%d\n",(*pep).nome,(*pep).numero,(*pep).cc);
```

Por questões de precedência do operador de acesso a membro de um registo ≪`.`≫ sobre o operador de ponteiro ≪`*`≫ a utilização de ≪`()`≫ é obrigatória. No caso do *C* a definição das estruturas e a sua utilização vai levar a que este último situação ocorra muitas vezes. Este facto terá estado na base do desenvolvimento de uma sintaxe alternativa, mais apelativa.

Em vez de `(*pep).nome` podemos escrever `pep->nome`.

```
printf("%s%d%d\n",pep->nome,pep->numero,pep->cc);
```

Os objectos do tipo estrutura podem ser argumentos de funções, assim como o resultado de funções. É também possível fazer atribuições entre variáveis do tipo estrutura.

3.8.2 Estruturas de Dados Dinâmicas

As estruturas de dados estáticas tal como as vistas na secção 3.8.1 levantam duas questões:

- têm uma dimensão (número de elementos) fixa;
- têm uma forma fixa.

Dimensão (número de elementos) Fixa Ao se definir uma tabela com um número de elementos fixos isso leva a que, aquando da utilização do programa esse número máximo de elementos não pode ser ultrapassado, isto é se a utilização do programa revelar que a estimativa para a dimensão da estrutura está errada a única forma de corrigir o problema é alterar a estrutura e recompilar o programa. Por outro lado se a estimativa se revelar muito exagerada há um desperdício de memória que é tento maior quantos mais elementos se utilizarem.

Um caso que pode servir para ilustrar esta situação é a da variável `listaEmpregado` no exemplo da secção 3.8.1. Com esta definição a empresa está limitada a 100 empregados, não é possível guardar a informação para 101, ou mais, empregados.

Por outro lado a definição do membro nome é uma apropriada para nomes portugueses (em geral muito grandes), já é totalmente exagerada para uma companhia que, por exemplo, tivesse a maior parte dos trabalhadores de nacionalidade inglesa (nomes em geral pequenos).

Forma Fixa Uma outra questão tem a ver com a implementação de estruturas cuja forma e dimensão são muito variáveis.

Por exemplo, como representar uma rede rodoviária, com as cidades e as estradas que as ligam? Uma resposta a esta questão é dada por uma estrutura dinâmica, tanto na forma como na dimensão, os grafos.

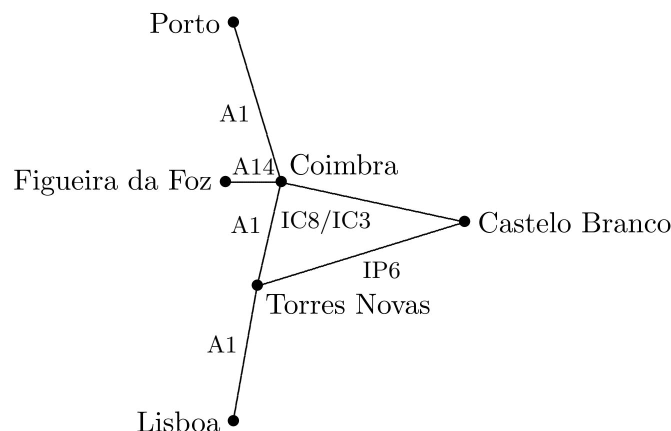


Figura 3.9: Mapa de Estradas (grafo)

Embora seja possível representar esta estrutura utilizando estruturas estáticas a utilização de estruturas dinâmicas capazes de se adaptarem tanto na forma como no número de elementos a guardar é muito vantajosa.

Estruturas Dinâmicas na Dimensão Podemos definir tabelas com um número de elementos não explicitado à partida. As seguintes declarações são equivalentes:

```
int* tabela1;
int tabela2[];
```

ambas definem ponteiros para inteiros.

Em ambos os casos a possibilidade de associar um série de posições de memória contiguas ao ponteiro assim definido permite-nos forma criar uma tabela de elementos do tipo pretendido

(neste caso o tipo `int`).

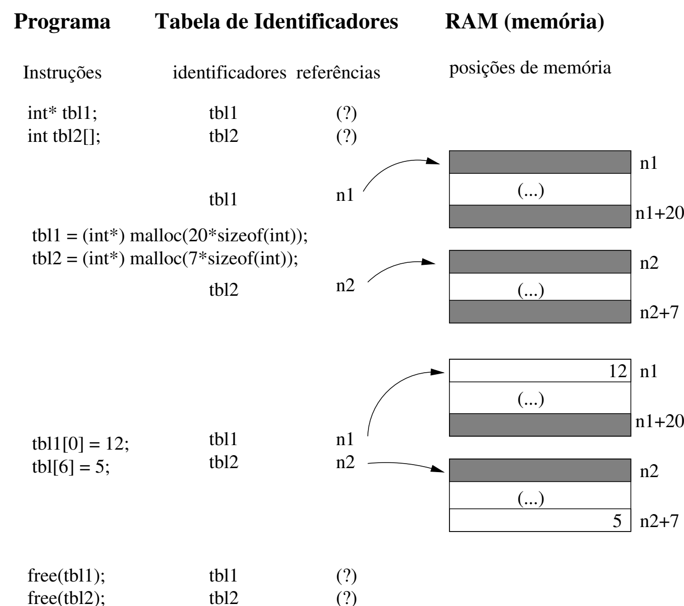


Figura 3.10: Variáveis Estáticas vs Variáveis Dinâmicas

A gestão de memória, a sua afectação e libertação está a cargo de duas funções específicas da linguagem *C*, as funções da biblioteca padrão `malloc` e `free`.

De forma a afectar memória usa-se a função `mallow`.

```
void *malloc(size_t tamanho)
```

A função `malloc` devolve um ponteiro para um objecto de dimensão `tamanho` ou o ponteiro `NULL` caso o pedido de memória não possa ser satisfeito. O espaço (apontado) não é inicializado.

Assim como a afectação, a libertação deste espaço de memória é também da responsabilidade do programador, o qual deve ter o cuidado de proceder à libertação do espaço logo que o mesmo deixe de ser necessário. Para este efeito existe, na biblioteca padrão, a função `free` cujo formato é o seguinte:

```
void free(void *ponteiro)
```

A função `free` liberta o espaço apontado pelo seu argumento (ponteiro); não faz nada caso o ponteiro seja o ponteiro `NULL`. O ponteiro deve apontar para um espaço que tenha sido criado anteriormente pela função `malloc`.

Em conclusão, no caso das tabelas podemos optar por uma declaração estática, ou por uma declaração dinâmica. Por exemplo:

```
char Estatica[20];

char tblDinamica[];
tblDinamica = (char) malloc(20*sizeof(char));
```

tendo afectado o mesmo número de posições de memória para as duas variáveis estas são equivalentes do ponto de vista de utilização (excepção feita para a possibilidade/necessidade

de libertar o espaço afectado no segundo caso). No entanto a segunda declaração dá-nos a liberdade de só afectar o espaço no momento em que ele é necessário e (eventualmente) já com o conhecimento do espaço exacto que é necessário.

No primeiro caso o espaço a afectar tem de ser estimado aquando da programação, no segundo caso o espaço a afectar pode ser determinado aquando da utilização, ajustando-o às necessidades.

Estruturas Dinâmicas na Forma (e Dimensão) Além das vantajens que advêm do ajustar a dimensão às necessidade, a gestão dinâmica da memória dada pela utilização dos ponteiros permite também a definição de estruturas dinâmicas, não só na sua dimensão mas também na sua forma.

Como exemplos de estruturas dinâmicas muito usadas em programação temos:

Listas: sequências lineares de elementos:

Pilhas: listas com acesso por um só ponto, o topo. Implementam a a disciplina *Last In First Out (LIFO)*, o último a entrar é o primeiro a sair. Entre muitas utilizações temos o cálculo de expressões em notação Polaca inversa.

$$Pilha = (\{pilhaVazia, elemento\!:\!Pilha\}, \{\text{cria, push, pop, top, vazia?}\})$$

As pilhas são definidas indutivamente. Temos a $pilhaVazia$ (caso de base) e temos as pilhas não vazias (caso indutivo) que são constituídas por um elemento seguido de uma pilha.

Para a definição das operações internas é necessário explicitar conjunto das pilhas não vazias $PilhaNãoVazia = Pilha \setminus \{pilhaVazia\}$.

As operações têm a seguinte especificação:

$$\begin{array}{rcccl} \text{cria} & : & \mathbf{1} & \longrightarrow & Pilha \\ & & * & \longmapsto & pilhaVazia \end{array}$$

$$\begin{array}{rcccc} \text{push} & : & Pilha \times Elementos & \longrightarrow & Pilha \\ & & (p, e) & \longmapsto & e : p \end{array}$$

$$\begin{array}{rcccc} \text{pop} & : & PilhaNãoVazia & \longrightarrow & Pilha \\ & & e : p & \longmapsto & p \end{array}$$

$$\begin{array}{rcccc} \text{top} & : & PilhaNãoVazia & \longrightarrow & Elementos \\ & & e : p & \longmapsto & e \end{array}$$

$$\begin{array}{rcccl} \text{vazia?} & : & Pilha & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathscr{V}, & \text{se p = pilhaVazia} \\ \mathscr{F}, & \text{se p} \neq \text{pilhaVazia} \end{cases} \end{array}$$

Filas listas com acesso apenas pelas extremidades, a entrada e a saída. Implementam a disciplina *First In First Out (FIFO)*, o primeiro a entrar é o primeiro a sair. A simulação de filas de espera, sejam elas caixas num dado hipermercado ou semáforos num dado cruzamento, podem ser modelizadas através deste tipo de estrutura.

$$Fila = (\{filaVazia, elemento\colon Fila\}, \{\text{cria, insere, retira, topo, vazia?}\})$$

A exemplo das pilhas, as filas são também definidas indutivamente. De igual modo é necessário explicitar o conjunto das filas não vazias: $FilaN\tilde{a}oVazia = Fila \setminus \{filaVazia\}$.

As operações têm a seguinte especificação:

$$\begin{array}{rcrcl} \text{cria} & : & \mathbf{1} & \longrightarrow & Fila \\ & & * & \longmapsto & filaVazia \end{array}$$

$$\begin{array}{rcrcl} \text{insere} & : & Fila \times Elementos & \longrightarrow & Fila \\ & & (f, e) & \longmapsto & e : f \end{array}$$

$$\begin{array}{rcrcl} \text{retira} & : & FilaN\tilde{a}oVazia & \longrightarrow & Fila \\ & & f : e & \longmapsto & f \end{array}$$

$$\begin{array}{rcrcl} \text{topo} & : & FilaN\tilde{a}oVazia & \longrightarrow & Elementos \\ & & f : e & \longmapsto & e \end{array}$$

$$\begin{array}{rcrcl} \text{vazia?} & : & Fila & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathscr{V}, & \text{se p = filaVazia} \\ \mathscr{F}, & \text{se p} \neq \text{filaVazia} \end{cases} \end{array}$$

Listas: listas genéricas com acesso livre a uma qualquer posição. Podem ser usadas para modelizar todo o tipo de situações em que se pretende uma lista de elementos todos do mesmo tipo mas em que o número de elementos a considerar varia muito durante o correr do programa.

$$\text{Lista} = (\{listaVazia, elemento\colon lista\}, \{\text{cria, insereN, retiraN, veN, vazia?}\})$$

A exemplo das pilhas e filas são também definidas indutivamente.

As operações têm a seguinte especificação:

$$\begin{array}{rcrcl} \text{cria} & : & \mathbf{1} & \longrightarrow & Lista \\ & & * & \longmapsto & listaVazia \end{array}$$

$$\begin{array}{rcrcl} \text{insereN} & : & Lista \times Elementos \times \mathbb{N} & \longrightarrow & Lista \\ & & (l, e, i) & \longmapsto & l' = l_1 : \ldots : l_{i-1} : e : l_i : \ldots : l_n \end{array}$$

No caso em que o comprimento da lista é menor do que a posição em que se pretende efectuar a inserção vai-se optar por inserir no fim da lista. A outra opção possível é a de considerar que nesse caso o valor da função não está definido.

$$\begin{array}{rcrcl} \text{retiraN} & : & Lista \times \mathbb{N} & \longrightarrow & Lista \\ & & (l, i) & \longmapsto & l' = l_1 : \ldots : l_{i-1} : l_{i+1} : \ldots : l_n \end{array}$$

No caso em que o comprimento da lista é menor do que a posição em que se pretende efectuar a remoção esta não é efectuada. A outra opção possível é a de considerar que nesse caso o valor da função não está definido.

$$\begin{array}{rcccl} \text{veN} & : & \mathit{Lista} \times \mathbb{N} & \longrightarrow & \mathit{Elementos} \\ & & (l,i) & \longmapsto & \left\{ \begin{array}{ll} e, & \text{se } |l| \geq i \\ \bot, & \text{se } |l| < i \end{array} \right. \end{array}$$

$$\begin{array}{rcccl} \text{vazia?} & : & \mathit{Lista} & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \left\{ \begin{array}{ll} \mathscr{V}, & \text{se p = listaVazia} \\ \mathscr{F}, & \text{se p} \neq \text{listaVazia} \end{array} \right. \end{array}$$

Árvores: estrutura hierárquica.

Árvores Binárias: uma raiz e dois sub-árvores. Dado que é possível representar qualquer tipo de árvore como uma árvore binária, este tipo de estrutura de dados é usada para modelizar todas as situações em que se pretende representar uma estrutura hierárquica.

$$\begin{array}{l} \text{AB} = (\{\mathit{Abvazia}, \mathit{ABesq}\!:\!\mathit{elemento}\!:\!\mathit{ABdir}\}, \{\text{criaVazia,criaAB,} \\ \qquad \text{procuraElemento,insereElemento,retiraElemento,vazia?}\}) \end{array}$$

No caso deste tipo de dados há variantes tanto na forma de definir assim como nas operações de base (Main & Savitch, 1997; Semgupta & Korobkin, 1994; Weiss, 1997; Aho *et al.*, 1983).

Uma operação importante para este tipo de dados é a travessia da árvore, isto é, o ≪visitar≫ de todos os nós sem esquecimentos e sem repetições. Podemos ter travessias *em-ordem*, em *pré-ordem* e em *pós-ordem*, consuante a ordem pela qual se visita a raiz em relação aos outros dois elementos, a sub-árvore esquerda e a sub-árvore direita.

De novo temos uma definição indutiva, neste caso com uma dupla indução.

Esta estrutura e a forma como é utilizada é muito mais complexa e variada do que os casos anteriores. Apresenta-se de seguida as definições das operações, é necessário ter em conta que neste caso não há uma uniformidade, estas definições representam uma possível definição para uma utilização genérica.

$$\begin{array}{rcccl} \text{criaVazia} & : & \mathbf{1} & \longrightarrow & AB \\ & & * & \longmapsto & ABVazia \end{array}$$

$$\begin{array}{rcccl} \text{criaAB} & : & AB \times \mathit{Elementos} \times AB & \longrightarrow & AB \\ & & (ab1, e, ab2) & \longmapsto & ab1 : e : ab2 \end{array}$$

Tanto $ab1$ como $ab2$ podem ser árvores binárias vazias. No caso extremo de ambas as sub-árvores serem vazias temos aqui uma operação que transforma um elemento numa árvore binária.

$$\begin{array}{rcccl} \text{procuraElemento} & : & AB \times \mathit{Elementos} & \longrightarrow & \mathbb{B} \\ & & (ab, e) & \longmapsto & \left\{ \begin{array}{ll} \mathscr{V}, & e \in ab \\ \mathscr{F}, & e \notin ab \end{array} \right. \end{array}$$

$$\begin{array}{rcrcl} \text{insereElemento} & : & AB \times Elementos & \longrightarrow & AB \\ & & (ab, e) & \longmapsto & ab' \end{array}$$

A posição de inserção vai estar dependente da forma como os elementos estão organizados na árvore. No caso da inserção ser feito nos extremos (folhas) da árvore ao elemento serão «adicionadas» duas sub-árvores vazias.

$$\begin{array}{rcrcl} \text{retiraElemento} & : & AB \times Elementos & \longrightarrow & AB \\ & & (ab, e) & \longmapsto & ab' \end{array}$$

No caso do retirar de uma das folhas (elementos no extremo da árvore) é só uma questão de colocar no nó da árvore exactamente acima do elemento a retirar a árvore vazia. No caso de se pretender retirar um elemento do meio da árvore, isso vai implicar um re-ordenar da mesma. A forma exacta como isso é feito vai depender da forma como a árvore está organizada.

$$\begin{array}{rcrcl} \text{vazia?} & : & AB & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathscr{V}, & \text{se p} = \text{ABVazia} \\ \mathscr{F}, & \text{se p} \neq \text{ABVazia} \end{cases} \end{array}$$

Grafos: conjunto de nós e de ramos entre eles (ver figura 3.9). É uma estrutura muito importante quando se quer modelizar situações em rede, isto é, situações em que se tem «locais» e «ligações» entre eles.

Para este tipo de estrutura temos, a exemplo das árvores, diferentes representações. Talvez a mais usual é a representação através de um conjunto de nós e um conjunto de arcos (pares de nós: nó origem, nó destino).

As operações passam pela criação do grafo vazio, a inserção de um novo nó, a inserção de um novo arco, as travessias, e as operações de retirar nós e arcos.

Pelas definições destes tipos de dados acima apresentados é possível reparar que a definição dos elemen tos é feita de forma recursiva. Por exemplo para as *Pilhas*:

$$\text{Pilha} = \begin{cases} \text{Vazia,} & \text{Caso de base} \\ \text{Elemento:Pilha} & \text{Caso Recursivo} \end{cases}$$

Este tipo de estrutura é possível de representar em C da seguinte forma:

```
typedef struct no { // nó de um pilha
  int elems;          // o elemento
  struct no* prox;    // o ponteiro para o próximo elemento
} No;

typedef No* Pilha;   // Pilha como o tipo ponteiro para No
```

Temos então a definição de uma pilha como uma lista de nós. A definição recursiva é possível dado que o elemento não se auto-referência, o que temos é que `prox` é do tipo ponteiro para `no` e não um elemento do tipo `no`.

Para a definição de uma árvore binária ter-se-ia algo de muito idêntico.

```
struct ABno {           // Nó árvore binária
  int elems;            // o elemento na «raiz» da árvore
  struct ABno* ABesq; // ponteiro para a AB esquerda
  struct ABno* ABdir; // ponteiro para a AB direita
}
```

Exemplo: Pilha de Caracteres

A título de exemplo apresenta-se uma possível implementação do tipo abstracto de dados *Pilha* em *C*.

Ficheiro `pilhas.h`

```
/*********************************************/
/*  Pilhas Implementação Dinâmica            */
/*********************************************/
#if !defined PILHAS
#define PILHAS

typedef struct no {
  int elems;
  struct no* prox;
} No;

typedef No* Pilha;

void cria(Pilha*);
int push(int, Pilha*);
int top(Pilha);
int pop(Pilha*);
int vazia(Pilha);

#endif
```

Ficheiro `pilhas.c`

```
#include "pilha.h"
#include <stdlib.h>
#include <stdio.h>

#define PILHAVAZIA 1

Pilha afectapilha(void) {
  return (Pilha) malloc(sizeof(No));
}

void cria(Pilha* p){
  (*p) = NULL;
}

int push(int elem,Pilha* p){
```

```
  Pilha novo;

  if (novo = afectapilha()){
    novo->elems = elem;
    novo->prox = (*p);
    (*p) = novo;
    return 0;
  }
  else
    return 1;
}

int pop(Pilha* p){
  Pilha aux;

  if ((*p) == NULL)
    return PILHAVAZIA;
  aux = (*p);
  (*p) = (*p)->prox;
  free(aux);
  return 0;
}

int top(Pilha p){
  return(p->elems);
}

int vazia(Pilha p){
  return(p == NULL);
}
```

Capítulo 4

A Biblioteca Padrão do *C*

Uma linguagem cujo objectivo seja a construção de programas grandes e complexos tem de, obrigatoriamente, suportar a construção modular de tais programas. Por outro existe um conjunto de funcionalidades que são básicas à maior parte dos programas não fazendo sentido rê-implementá-las sempre que necessário.

Podemos definir as bibliotecas padrão como sendo um agregado de módulos que implementam as funcionalidades mais comummente usadas pelos programadores.

A separação entre a implementação da linguagem e a biblioteca padrão, ≪Standard Library≫, está já presente na linguagem *C*.

A biblioteca padrão é algo que todo o implementador de um sistema *C* tem de providenciar e em que todo o programador pode contar para a construção dos seus programas (Stroustrup, 1997, §16).

4.1 A Biblioteca Padrão

A biblioteca padrão *C* providência um conjunto de funções e macros que, não fazendo parte da definição da linguagem, providenciam um conjunto de funcionalidades adicionais.

O acesso às mesmas passa pela inclusão dos diferentes ficheiros de especificação no programa, e dos correspondentes ficheiros de código no momento da compilação.

De seguida descrevem-se as características mais importantes das bibliotecas mais relevantes.

Para mais detalhes consultar (Kernighan & Ritchie, 1988), assim como as seguintes páginas da Rede: a *gnu C library*[1] ou a secção dedicada à biblioteca padrão do *C*[2], na página da *The C++ Resources Network*.

4.1.1 Diagnósticos: `<assert.h>`

As funções nesta biblioteca são usada para processar diagnósticos em programas. Para uma descrição completa da biblioteca `assert.h` consultar (Kernighan & Ritchie, 1988).

4.1.2 Funções Matemáticas: `<math.h>`

Esta biblioteca contêm a implementação das funções matemáticas mais usuais.

[1] `https://www.gnu.org/software/libc/`
[2] `http://www.cplusplus.com/reference/clibrary/`

As funções na tabela 4.1 têm um ou dois argumentos do tipo `double` e devolvem um resultado do tipo `double`. Para todas as funções trigonométricas os ângulos são expressos em radianos.

`sin(x)`	seno de x
`cos(x)`	cosseno de x
`tan(x)`	tangente de x
`asin(x)`	arco seno, $\sin^{-1}(x)$, no intervalo $[-\pi/2, \pi/2]$, com $x \in [-1, 1]$
`acos(x)`	arco cosseno, $\cos^{-1}(x)$, no intervalo $[0, \pi]$, $x \in [-1, 1]$
`atan(x)`	arco tangente, $\tan^{-1}(x)$, no intervalo $[-\pi/2, \pi/2]$
`atan2(y,x)`	arco tangente, $\tan^{-1}(y/x)$, no intervalo $[-\pi, \pi]$
`sinh(x)`	seno hiperbólico de x
`cosh(x)`	coseno hiperbólico de x
`tanh(x)`	tangente hiperbólica de x
`exp(x)`	e^x, função exponencial (base e)
`log(x)`	logaritmo natural (base e) de x $(x > 0)$
`log10(x)`	logaritmo decimal (base 10) de x $(x > 0)$
`pow(x,y)`	x^y, erro se $x = 0$ e $y \leq 0$, ou se $x < 0$ e y não é um número inteiro
`sqrt(x)`	$\sqrt{x}$ $(x \geq 0)$
`fabs(x)`	$\|x\|$

Tabela 4.1: Funções Matemáticas em `math.h`

Existem também duas funções de conversão (arredondamento) de reais para inteiros.

- `ceil(x)`, devolve o menor inteiro, maior de que o real x.
- `floor(x)`, devolve o maior inteiro, menor de que o real x.

Para uma descrição completa da biblioteca `math.h` consultar (Kernighan & Ritchie, 1988).

4.1.3 Casos Especiais para Argumentos de Funções: `<stdarg.h>`

Possibilita definir funções com um número indeterminado de argumentos. Para uma descrição completa da biblioteca `stdarg.h` consultar (Kernighan & Ritchie, 1988).

4.1.4 stdlib `<stdlib.h>`

Declara funções para conversão entre tipos, funções para a gestão de memória, assim como outras funções auxiliares.

Vejamos o exemplo das funções para a geração de números (pseudo) aleatórios.

Geração de Números Aleatórios

A biblioteca `stdlib` possuí duas duas funções para gerar números pseudo-aleatórios.

- `int rand(void)`, `rand` (de *random*), devolve um número pseudo-aleatório na gama de `0` a `RAND_MAX`, o qual é igual ou maior do que 32767.

- `void srand(unsigned int semente)`, `srand` usa a `semente` (*seed*), para definir o começo da sequência de pseudo-aleatórios.

Para que um programa tenha sempre um valor inicial da sequência de números pseudo-aleatórios diferente de cada vez que é executado, é usual, utilizar a função `time()` como semente do gerador. Por exemplo;

```
#include <stdio.h>  // printf
#include <stdlib.h> // rand, srand, RAND_MAX
#include <math.h>   // floor
#include <time.h>   // time


int main() {
  int aleatorio;

  // inicializar o gerador
  srand(time(NULL));
  // gerar um inteiro pseudo-aleatório entre 5 e 15
  aleatorio = 5+floor(1+rand()%10);
  // escreve o resultado
  printf("%d\n",aleatorio);
}
```

Note-se a necessidade de incluir as bibliotecas: `stdio` (`printf`); `stdlib` (`rand` e `srand`); `math` (`floor`); `time` (`time`).

Para uma descrição completa da biblioteca `stdlib.h` consultar (Kernighan & Ritchie, 1988).

4.1.5 Teste de Classes de Caracteres: `<ctype.h>`

Esta biblioteca tem um conjunto de funções próprias para testar um carácter e dizer em que classe (dígitos, letras, etc.) este se encontra. Para uma descrição completa da biblioteca `ctype.h` consultar (Kernighan & Ritchie, 1988).

4.1.6 Limites Dependentes do Compilador: `<limits.h>` & float `<float.h>`

Estas duas classes definem os limites, dependentes do compilador usado, para os tipos inteiros e reais, respectivamente. Para uma descrição completa da biblioteca `limits.h` e `float.h` consultar (Kernighan & Ritchie, 1988).

4.1.7 Saídas Alternativas de Funções: `<setjmp.h>`

Providencia formas alternativas à forma normalizada de saída de uma função. Para uma descrição completa da biblioteca `setjmp.h` consultar (Kernighan & Ritchie, 1988).

4.1.8 Definições Padrão: `<stddef.h>`

Define alguns padrões dentro da linguagem C. Para uma descrição completa da biblioteca `stddef.h` consultar (Kernighan & Ritchie, 1988).

4.1.9 Funções para ≪Strings≫: `<string.h>`

Conjunto de funções para lidar com sequências de caracteres. Para uma descrição completa da biblioteca `string.h` consultar (Kernighan & Ritchie, 1988).

4.1.10 Localização: `<locale.h>`

Define um conjunto de declarações e funções relativas à localização (adaptação a uma dada cultura, por exemplo a Portuguesa) de um programa. Para uma descrição completa da biblioteca `locale.h` consultar (Kernighan & Ritchie, 1988).

4.1.11 Sinais: `<signal.h>`

Providencia formas de lidar com condições de excepção. Para uma descrição completa da biblioteca `signal.h` consultar (Kernighan & Ritchie, 1988).

4.1.12 Entradas e Saídas: `<stdio.h>`

As funções relacionadas com as entradas **I**nput e saídas **O**utput.

As leituras/escritas em *C* são feitas usando *fluxos de informação* (*streams*), os quais podem estar associados a diferentes fontes/destinos.

Um fluxo de informação é uma sequência de linhas de de um ou mais caracteres terminada por um '`\n`'.

Aquando do início de um programa são abertos de imediato três canais de informação referentes a três fluxos de informação: `stdin`, para fluxos de entrada (leitura); `stdout`, para fluxos de saída (escrita); `stderr`, para fluxos de saída relacionados com situações de erro.

Os fluxos de informação estão associados a canais de comunicação no sistema operativo. Em *C* temos as seguintes funções `scanf` e `printf` associados ao teclado e ao ecrã (os canais padrão); `fprinf` e `fscanf` associados a ficheiros **file**; e `sprintf` e `sscanf` associados a sequências de caracteres *string*.

Operações com Ficheiros

As funções que a seguir se descrevem lidam com ficheiros. O tipo `size_t` é um valor inteiro, positivo, resultado do operador `sizeof`.

```
FILE *fopen(const char *nome_de_ficheiro, const char *modo)
```

`fopen` abre um canal de comunicação associando-o ao ficheiro cujo nome é o primeiro argumento. Se, por alguma razão o comando falha o valor `NULL` (ponteiro nulo) é devolvido.

Na tabela 4.2 apresentam-se os modos mais usuais.

Modo	
r	(**r**ead), abre um ficheiro para leitura
w	(**w**rite), cria um ficheiro para escrita, apaga qualquer conteúdo prévio
a	(**a**ppend), abre, ou cria, um ficheiro para escrita, adicionando o novo conteúdo ao fim do ficheiro

Tabela 4.2: Modos de Abertura para Ficheiros

Os nomes de ficheiros estão limitados a um comprimento de `FILENAME_MAX` caracteres. Num dado momento somente `FOPEN_MAX` ficheiros podem estar abertos.

```
int fflush(FILE *canal)
```

Dado um canal de saída, `fflush` «limpa» esse canal enviando todos a informação ainda na memória (de escrita) para o meio de saída associado ao canal de saída. Num canal de entrada o seu efeito é indefinido. Devolve `EOF` (***End Of File***) em caso de erro, e zero em caso de sucesso. No caso de se escrever `fflush(NULL)`, todos os canais de saída são «limpos».

```
int fclose(FILE *canal)
```

`fclose` envia toda a informação para o canal de saída, ou ignora a restante informação no canal de leitura, liberta todos as memórias dedicadas ao canal de comunicação e de seguida fecha o canal. Devolve `EOF` se houver erros, zero no caso contrário.

Entradas e Saídas Formatadas

```
int fprintf(FILE *canal, const char *formato, ...)

int printf(const char *formato, ...)

int sprintf(char *s, const char *formato, ...)
```

`fprintf` converte, de acordo com o formato definido, e escreve no canal de saída a informação. O valor de retorno é o número de caracteres escritos. Em caso de erro um valor negativo é devolvido.

Temos que `printf(...)` é equivalente a `fprintf(stdout, ...)` e `sprintf(...)` escreve o resultado numa dada sequência de caracteres (o primeiro argumento).

```
int fscanf(FILE *canal, const char *formato, ...)

int scanf{const char *formato, ...)

int sscanf{char *s, const char *formato, ...)
```

`fscanf` lê do canal de entrada e, tendo em atenção o formato definido, converte as palavras, e atribui os valores convertidos aos argumentos dados na instrução os quais têm de ser do tipo ponteiro. Devolve `EOF` se algum erro ocorreu mesmo antes de fazer a conversão. Em caso de sucesso devolve o número de elementos lidos, convertidos e atribuídos.

Temos que `scanf(...)` é equivalente a `fscanf(stdin, ...)` e `sscanf(...)` obtêm os dados de entrada de uma dada sequência de caracteres (o primeiro argumento).

O formato de conversão, para leitura, pode conter:

- espaços, os quais são ignorados.
- Caracteres (que não '%'), os quais devem coincidir com os caracteres, que não espaços, do canal de entrada.
- Especificações de conversão, as quais consistem de um '%', opcionalmente um carácter de supressão '*', opcionalmente, um número que especifica o comprimento máximo da entrada, um opcional '`h`', '`l`', ou '`L`' indicando a largura do alvo, e um carácter de conversão.

Funções de Leitura/Escrita de Caracteres

Além das entradas e saídas formatadas esta biblioteca também possuí uma série de funções que lidam com caracteres individuais.

```
int fgetc{FILE *canal)
```

`fgetc` devolve o próximo carácter no canal de comunicação como um inteiro sem sinal (a codificação referente ao carácter em questão). Em caso de erro tem-se `EOF`.

```
char *fgets(char *s, int n, FILE *canal)
```

`fgets` lê, quanto muito $n-1$ caracteres para o vector s, parando assim que se chega ao fim da linha; o carácter ≪mudança de linha≫ é incluído no vector de caracteres o que é terminado (automaticamente) por '\0'.

`fgets` devolve s, ou `NULL` em caso de erro.

Por exemplo, para ler um ficheiro, escrevendo o seu conteúdo no canal de saída (ecrã) podíamos escrever o seguinte programa.

```
#include <stdio.h>

int main () {
  FILE *cLeitura; // declarar o canal de leitura
  char aux; // variável auxiliar

  // abrir um canal de leitura a partir de ficheiro
  cLeitura = fopen("texto.txt","r");
  if (cLeitura != NULL) { // Sem erros
    // leitura de caracteres até se atingir o fim de ficheiro
    while ((aux = fgetc(cLeitura)) != EOF) {
      printf("%c",aux);
    }
    return(0); // saída sem erros
  }
  else { // erro na abertura de canal de comunicação
    printf("Não foi possível ler do ficheiro 'texto.txt'\n");
    return(1); // saída com erros
  }
}
```

Além destas funções temos as correspondente referentes à escrita de caracteres.

```
int fputc(int c, FILE *canal)
```

`fputc` escreve um carácter '`c`' (convertido para um `char`) no canal de escrita. Devolve o carácter escrito ou, em caso de erro, `EOF`.

```
int fputs(const char *s, FILE *canal)
```

`fputs` escreve a sequência de caracteres '`s`' (a qual não pode conter um '\n ,) no canal. Devolve um não negativo ou, em caso de erro, `EOF`.

Para uma descrição completa da biblioteca `stdio.h` consultar (Kernighan & Ritchie, 1988).

4.1.13 Tempo e Datas: `<time.h>`

Declara tipos e funções para lidar com as datas e o tempo.

Destaco duas funções interessantes nesta biblioteca.

- `clock_t clock(void)`, esta função devolve o tempo de processamento (CPU) usado desde o começo de funcionamento de um dado programa. O valor `clock()/CLOCKS_PER_SEC` dá-nos o tempo em segundos.
- `time_t time(time_t *tp)`, esta função devolve o tempo corrente. Se o ponteiro `tp` é não nulo o valor de saída é também atribuído a `*tp`.

Um exemplo da utilização da função `time` é dado pela sua utilização para inicializar o gerador de pseudo-aleatórios (ver Secção 4.1.4).

A função `clock` é útil para se obter o tempo de execução de um programa.

```
/* Exemplo de utilização de clock: Frequência de Primos */
#include <stdio.h>       // printf
#include <time.h>        // clock_t, clock, CLOCKS_PER_SEC
#include <math.h>        // sqrt

int frequenciaPrimos (int n) {
  int i,j;
  int freq;

  freq = n-1;
  for (i=2; i<=n; ++i)
    for (j = sqrt(i); j>1; --j)
      if (i%j == 0) {
        --freq; break;
      }
  return(freq);
}

int main () {
  clock_t t;
  int limite,numPrimos;

  printf("Calculando a frequência de Números Primos.\n\n");
  printf("Introduza o limite superior: ");
  scanf("%d",&limite);
  t = clock(); // valor inicial
  numPrimos = frequenciaPrimos(limite);
  printf ("O número de primos menores que %d é: %d\n",limite,numPrimos);
  t = clock() - t; // tempo gasto <- tempo final - tempo inicial
  printf ("CPU %d 'clicks' (%f segundos).\n",t,((float)t)/CLOCKS_PER_SEC);
  return 0;
}
```

A sua execução dar-nos-ia:

```
Calculando a frequência de Números Primos.
```

```
Introduza o limite superior: 10000
O número de primos menores que 10000 é: 1229
Levou-me 6066 'clicks' (0.006066 segundos).
```

Para uma descrição completa da biblioteca `time.h` consultar (Kernighan & Ritchie, 1988).

Capítulo 5

Ordenação & Pesquisa

Dado a possível grande dimensão das estruturas de dados do tipo tabela, uma questão que surge naturalmente em muitas das situações relacionadas com este tipo de estrutura é o de encontrar um dado elemento, e eventualmente a sua posição, num dado vector.

Se o vector não está ordenado a única estratégia possível é a procura exaustiva, começar numa ponta e ir até ao fim. Se o vector está ordenado é possível usar uma estratégia que está próxima da forma como se consulta um dicionário (em cópia física, não virtual), abre-se numa dada página e depois procura-se para trás ou para a frente consoante o resultado desta primeira pesquisa.

5.1 Pesquisa

5.1.1 Pesquisa Exaustiva

Para o caso em que o vector não está ordenado. No caso em que o vector não esteja ordenado o único algoritmo possível é dado pela procura exaustiva desde o início do vector até ao seu fim.

5.1.2 Pesquisa Binária

No caso em que o vector está ordenado podemos usar esse facto para implementar um método eficiente de pesquisa. Método ≪Pesquisa Binária≫.

- Verificar se o elemento é igual à posição média do vector.
- Caso seja menor do que a posição média, repetir o processo na primeira metade do vector;
- Caso seja maior do que a posição média, repetir o processo na segunda metade do vector.

O algoritmo pára assim que encontrar o elemento, ou quando atinge um vector de dimensão nula.

5.2 Ordenação

Dado a enorme diferença, em termos de tempo médio de pesquisa, entre as duas estratégias de procura a ordenação de tabelas uni-dimensionais torna-se importante.

Existem muitos métodos, alguns caracterizados por serem muito simples de descrever e implementar, mas nem sempre muito eficientes, outros mais complexos, mas também mais eficientes.

5.2.1 Borbulhagem

Método de ordenação *Borbulagem* («Bubble Sort»).

- Percorra o vector trocando pares de elementos que estejam fora de ordem.
- Repita o processo até que, numa das passagens anteriores, não se proceda a nenhuma troca.

Para um vector com n componentes são necessárias, no máximo, $n-1$ passagens.

```
/*
 * Recebe um vector, assim como o num. de elementos e ordena-o
 * através do método Borbulhagem («Bubble sort»)
 */
int borbulhagem(int nElem, int v[]) {

  int i,aux;
  int ordenado;

  do {
    ordenado = 1;
    for (i = 0; i < nElem-1 ; i++) { // precorre o vector
      if (v[i] > v[i+1]) { // se necessário troca elementos
        aux = v[i];
        v[i] = v[i+1];
        v[i+1] = aux;
        ordenado = 0;
      }
    }
  } while (!ordenado); // repete até não haver mais trocas

  return 0;
}
```

5.2.2 Selecção Linear

Método de ordenação *Selecção Linear* («Linear Sort»).

- Encontre o maior dos n elementos;
- Coloque esse elemento na sua posição final trocando-o com o enésimo elemento;
- Recursivamente ordene $n-1$ primeiras posições do vector.

5.2.3 Inserção Ordenada

Método de ordenação *Inserção Ordenada* («Insertion Sort»).

- Consideramos o vector como a concatenação de duas sequências: uma ordenada, e a segunda não ordenada. No início a primeira sequência contém apenas um elemento.
- O primeiro elemento da sequência não ordenada é inserido na posição correcta da sequência ordenada (movendo os elementos maiores uma posição para a esquerda).
- Esta operação aumentou a sequência ordenada de um elemento e retira esse elemento à sequência não ordenada.
- Repetimos até todos os elementos estarem na sequência ordenada.

5.2.4 Fusão

Método de ordenação *Fusão* («Merge Sort»).

Dados dois vectores, ordenados, podemos fundi-los de forma ordenada num outro vector.

- Comparam-se os dois elementos iniciais dos dois vectores.
- Escolhe-se o vector que contêm o elemento menor e copiam-se os seus elementos para o vector final até que o elemento que se está a considerar já seja maior do que o elemento inicial do outro vector.
- troca-se de vector e repete-se o processo, até que se chega ao fim de um dos vectores.
- copiam-se os restantes elementos do outro vector para o vector final.

O processo precisa sempre de um vector auxiliar, sobre o qual se vai construir o vector ordenado.

Pode-se ordenar um vector considerando que o mesmo é constituído por dois sub-vectores, isto é, divide-se o vector inicial em dois, fazendo de seguida a fusão ordenada desses dois sub-vectores.

Dado que as metades iniciais não estarão, provavelmente, ordenadas, repete-se (de forma recorrente) o processo de divisão até se atingir vectores individuais, que estão, obviamente, ordenados. O processo de fusão ordenada começa então a partir daí até se atingir o vector final.

5.2.5 Quick Sort

Método de ordenação («Quick Sort»). Dado um vector de n inteiros e um inteiro l existente no vector pretende-se particionar esse vector em duas partes: os elementos inferiores a l e os elementos superiores a l.

- Começando do lado esquerdo do vector procurar um elemento que seja maior ou igual a l.
- Começando do lado direito do vector procurar um elemento que seja menor ou igual a l.

- Trocar estes dois elementos.
- Continuar até as duas procuras se cruzarem.

No fim, posiciona-se o elemento l na sua posição final, entre os que lhe são menores e os que lhe são maiores.

Repete-se, por recorrência, o processo nos dois vectores resultantes dessa divisão entre menores e maiores do que l.

Capítulo 6

Complexidade Algorítmica

Em ciência da computação, o O-grande é usado para classificar algoritmos pela forma como eles respondem (por exemplo, no tempo de processamento ou espaço de trabalho requerido) a mudanças no tamanho da entrada.

A notação O-grande caracteriza funções de acordo com suas taxas de crescimento: funções diferente com as mesmas taxas de crescimento tem a mesma notação O-grande. A letra O é usada porque a taxa de crescimento de uma função também é referenciada como como a ordem de uma função. Uma descrição de uma função em termos de O-grande normalmente provê um limite superior na taxa de crescimento da função.

A notação O-grande é útil na análise da eficiência de algoritmos (espaço e/ou tempo).

Por exemplo, o tempo (ou o número de passos) que ele leva para resolve um problema de tamanho n pode ser considerado $T(n) = 4n^2 - 2n + 2$. Quando n cresce, o termo n^2 vai dominar, de forma que os outros termos podem ser negligenciados – por exemplo quando $n = 500$, o termo $4n^2$ é 1000 vezes maior que o termo $2n$. Ignorando este termo teria um efeito negligenciável sobre o valor da expressão para muitos propósitos. Além disso, os coeficientes se tornam irrelevantes se comparados a qualquer outra ordem de expressão, de forma que uma expressão contendo um termo n^3 ou n^4. Mesmo no caso de $T(n) = 1.000.000n^2$, se $U(n) = n^3$, este último será sempre superior ao primeiro n se torna maior do que $1,000,000$ $(T(1,000,000) = 1,000,000^3 = U(1,000,000))$. Adicionalmente, o número de passos depende dos detalhes do modelo de máquina no qual o algoritmo roda, mas diferentes tipos de máquina normalmente variam de apenas um factor constante no número de passos necessários para executar o algoritmo. Então a notação O-grande cobre o restante: podemos escrever tanto

$$T(n) = O(n^2),$$

quanto

$$T(n) \in O(n^2),$$

e dizer que o algoritmo possui complexidade de tempo na ordem de n^2. Note que "=" não significa "é igual a" como na senso comum matemático, e sim um "é" mais coloquial, de forma que a segunda expressão é tecnicamente precisa (veja a discussão sobre o "Sinal de igualdade" abaixo) enquanto a primeira é considerada por alguns com um abuso de notação.

$O(k)$	constante (k é uma constante natural
$O(\log n)$	logarítmico
$O(n)$	linear
$O(n^c), c > 1$	polinomial
$O(c^n), c > 1$	exponencial
$O(n!)$	factorial

Apêndice A

Documentação

Este capítulo é sobre a importância da documentação do ponto de vista do programador, só de forma breve é que se fala sob a perspectiva do utilizador, e mesmo aí é tendo na ideia uma utilizador/programador. A escrita de manuais para utilizadores não será aqui abordada.

O acto de construir um programa é, em geral, um acto não isolado, nem no tempo, nem na pessoa do programador. Isto é, um programa, mesmo que feito por uma só pessoa, ou por uma só equipa, não é, regra geral, feito num só momento ininterrupto. Por outro a construção de um programa é em geral um trabalho de equipa, sendo que a mesma pode sofrer alterações na sua composição durante a construção do programa.

Por este tipo de razões a documentação, interna, isto é como comentários no próprio código, ou externa, na forma de um relatório externo ao código do programa, é de uma importância extrema.

Neste capítulo dão-se, de forma breve, algumas indicações a seguir na construção de um programa, nomeadamente na documentação que o deve acompanhar.

Como última secção falar-se-á, também de forma breve, numa aproximação, digamos assim, radical ao problema. O programa não é o importante, a documentação é que é o importante. Esta aproximação, conhecida por *programação literária* advoga a mudança de hábitos do programador, este não deve começar por escrever código e mais tarde, eventualmente, escrever a documentação, mas pelo contrário, deve começar pela documentação e só depois é que deve passar para o código.

A.1 Documentação Interna

Através da utilização dos *comentários* a inserir nos ficheiros contendo os programas. A descrição, sucinta, das estruturas de dados (atributos) principais, a descrição em termos de pré e pós condições das diferentes funções implementadas.

Sucinta, mas presente.

No *C* podemos ter ≪**linhas de comentários**≫, tudo o que esteja à direita de um par de barras oblíquas para a direita (≪`//`≫) e até ao fim dessa linha e ≪**blocos de comentários**≫ tudo (várias linhas) o que estiver entre o par de símbolos ≪`/*`≫ e o par ≪`*/`≫.

A.2 Documentação Externa

A descrição do problema, o manual do utilizador, a documentação para o programador: descrição geral da estrutura(s) de dados utilizada(s); descrição geral do(s) algoritmo(s) construídos.

A.3 Programação Literária

Sob a designação de *programação literária* define-se uma aproximação à programação em que se pretende que os programas sejam, obras literárias, isto é que possam ser lidas e compreendidas facilmente por todos, em contra-ponto a programas escritos de tal forma que são de difícil leitura (muitas vezes, mesmo pelo próprio programador).

Na programação literária pretende-se que a documentação (legível por humanos) e código fonte (legível pela máquina) estejam num único arquivo (ficheiro) fonte, de modo a manter uma correspondência próxima entre a documentação e o código fonte. A ordem e a estrutura desse arquivo são especificamente projetadas para auxiliar a compreensão humana: código e documentação juntos são organizados em ordem lógica e/ou hierárquica (tipicamente de acordo com um esquema que acomode explicações detalhadas e comentários como necessárias).

Ferramentas externas pegam neste documento e geram a documentação do programa e/ou extraem o o programa propriamente dito para processamento subsequente por compiladores ou interpretadores.

O primeiro ambiente de programação literária publicado foi o sistema *WEB*, introduzido por Donald Knuth em 1981 (Knuth, 1984). Nesse sistema o programa *Weave* produz, a partir do documento original, a documentação a ser posteriormente formatada através do sistema TeX e o programa *Tangle* produz, a partir do documento original, o programa fonte em *Pascal*. Posteriormente o sistema *CWEB* foi introduzido com o intuito de dar suporte a este tipo de programação mas à linguagem de programação *C*.

Apêndice B

Construção de um Executável

As linguagens tais como *C* são ditas linguagens compiláveis. Quer se com isto dizer que, através da utilização de um programa próprio, o compilador, o programa escrito na linguagem *C* vai ser transformado num programa em código máquina e autónomo. Isto é o compilador vai transformar o programa em *C* num executável.

A forma como essa transformação é feita, e os programas que podem ser usados para nos ajudar nesse processo são o objecto deste capítulo.

B.1 Compilador de C

A transformação de um programa em *C* num programa executável é feito pelo compilador respectivo. É possível individualizar três fases distintas no processo de compilação: o pré-processamento; a fase de análise e de geração do código máquina (programa na linguagem da máquina); e agregação e finalização (algumas vezes designado por ≪linkagem≫, um aportuguesamento do termo inglês *linking* (agregando/juntando)). Vejamos com um pouco mais de detalhes estas diferentes fases.

pré-processamento: o ficheiro contendo o programa é analisado por completo sendo que as directivas de pré-processamento são cumpridas. É também nesta fase do processamento que todos os comentários são descartados. O resultado desta fase é um programa contendo exclusivamente código *C*. Num processo de compilação normal esta fase não produz nenhum ficheiro de saída.

análise: análise léxica (palavras), sintáctica (frases) e semântica (tipos de dados) do código resultante da fase anterior para avaliar se o mesmo está de acordo com a regras da linguagem *C*.

Se não houver erros, o resultado final desta fase é dado pela conversão do código *C* em código máquina. Nem sempre o resultado desta fase resulta na produção de um ficheiro de saída (ver a secção B.1.2).

agregação (*linkagem*): no caso de se tratar do ficheiro contendo a função `main` (≪ponto de partida≫ do programa) o compilador pode proceder à fase final da transformação, agregando todo o código máquina produzido nas fases anteriores, juntamente com as bibliotecas necessárias (incluídas através de directivas de pré-processamento), assim

como o código necessário à execução do código pelo sistema operativo (*runtime routines*), num ≪bloco≫ único que é o programa executável.

O resultado desta fase, para o caso em que não há erros, é um ficheiro contendo o executável respeitante ao nosso programa.

É de notar que actualmente é normal o ficheiro resultante do processo de compilação não ser verdadeiramente auto-contido, isto é, o ficheiro não tem em si tudo o que é necessário para o seu funcionamento.

O facto de cada vez mais se usar bibliotecas externas para complementar os programas faz com que a proporção entre o código próprio do programa e o código externo (bibliotecas) seja em muitos casos muito desproporcionado em favor das bibliotecas. Este facto é ainda mais visível numa linguagem como o *C* dado que a sua modularidade leva a um uso intensivo de bibliotecas padrão (biblioteca padrão do *C*, bibliotecas gráficas, etc.).

A conclusão do que foi dito acima é que, em geral, a fase de agregação vai ≪deixar de fora≫ as bibliotecas (do sistemas) objecto das instruções de inclusão (`#include`), deixando só a indicação que elas são necessárias para o funcionamento do programa. Este processo, designado por agregação dinâmica (*dynamic linking*) tem como grande vantagem gerar programas executáveis com uma dimensão (em ≪bytes≫) que é proporcional ao código *C* escrito pelo programador. Tem como desvantagem o facto de que, na ausência de uma das bibliotecas necessárias ao seu funcionamento, o programa não executará.

É possível explicitar que se pretende a inclusão de tudo o que é necessário para a execução do programa obrigando o compilador a proceder a uma, assim designada, agregação estática (*Static Linking*). A desvantagem é que um qualquer programa, mesmo que pequeno, irá gerar um executável de dimensão considerável (vai depender das bibliotecas que se pretendeu incluir). A vantagem é que neste caso o programa será auto-contido, não necessitando de nenhum recurso adicional para o seu funcionamento (ver a secção B.1.2).

B.1.1 Pré-processamento

As directivas de pré-processamento passam pela definição de ≪macros≫, da declaração dos nomes dos ficheiros a incluir (bibliotecas e/ou programas auxiliares), assim como a compilação condicional (Kernighan & Ritchie, 1988; Stroustrup, 1997).

Qualquer linha iniciada por '`#`' (eventualmente com espaços em branco antes) será pré-processada. O efeitos das directivas é global para todo o ficheiro em que estão contidas. A declaração de um destes comandos ocorre sempre numa única linha, no entanto uma declaração muito longa pode ser dividida em várias linhas do ficheiro, basta para tal colocar '\' no fim de uma linha para que a próxima linha seja formalmente uma continuação da linha anterior, isto é, do ponto de vista da definição as duas linhas passam a contar como um linha única.

Definição de ≪Macros≫

A sintaxe deste tipo de definição é a seguinte:

```
#define  <identificador> <sequênciaPalavrasReservadas>
#define  <identificador> (<listaIdentificadores>) <sequênciaPalavrasReservadas>
```

A definição de «macros» serve para dar um nome a uma expressão (eventualmente) complexa. Os casos mais usuais prendem-se com a definição de valores constantes que, desta forma, ganham um nome, um significado, assim como uma maior facilidade na sua (eventual) modificação. Por exemplo:

```
#define PI 3.14
#define NUMMAXELEM 30
```

No segundo caso podia significar o número de elementos que se estava a considerar para um dado vector. A declaração, leitura, escrita e cálculo, com os seus correspondentes ciclos seriam escritos todos em função deste valor. A sua alteração seria muito fácil. Por exemplo ter-se-ia:

```
int tabela[NUMMAXELEM];
```

O considerar de um valor mais exacto para π seria também uma questão de alterar uma só linha.

Uma nota: é usual escrever os identificadores associados à definição de macros com todas as letras em maiúsculas. Esta é uma convenção informal na comunidade de programadores em *C*.

A definição:

```
#define VALORABS(a,b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

define uma macro que devolve o valor o valor absoluto da diferença entre os seus argumentos.

Inclusão de Ficheiros

As inclusão de bibliotecas e/ou ficheiros auxiliares é definida do seguinte modo:

```
#include  <<nomeFicheiro>>
#include  "<nomeFicheiro>"
```

No primeiro caso trata-se de inclusão de bibliotecas do sistema, a sua localização concreta varia de acordo com o compilador e/ou sistema operativo usado. A sua utilização está dependente de uma correcta instalação da biblioteca no compilador/sistema operativo em causa.

O segundo caso é usualmente usado para a inclusão de programas auxiliares e/ou bibliotecas definidas pelo próprio programador. O nome do ficheiro pode incluir o «caminho» completo até ao ficheiro.

Ou seja, no primeiro caso a referência ao ficheiro é uma referência relativa, a conversão desta referência relativa para uma localização concreta do ficheiro em questão está a cargo do compilador. No segundo caso estamos a explicitar a localização concreta do ficheiro a incluir.

Compilação Condicional

As directivas de pré-processamento servem também para definir que certas componentes do programa podem ser, ou não, compiladas.

`#if`	<expressãoConstante>	início do condicional
`#ifdef`	<identificador>	se está definido
`#ifndef`	<identificador>	se não está definido
`#elif`	<expressãoConstante>	else if
`#else`	<expressãoConstante>	
`#endif`		fim do condicional

As linhas `if`, `elif` e `else` definem um conjunto de opções do qual uma (e só uma) será correcta e como tal processada, sendo os outros casos ignorados.

Uma das utilizações usuais para este tipo de directivas é dado pela inclusão de opções diferenciadas consoante o compilador e/ou sistema computacional em que a compilação está a ser efectuada.

Outra das utilizações é dada pela salvaguarda de uma tentativa (que falharia) de re-declaração de funções e/ou classes aquando da inclusão de um ficheiro auxiliar o qual, pela divisão ≪normal≫ de um programa em componentes, pode ser alvo de inclusão por mais do que um ficheiro e, por essa via, de uma dupla (ou mais) inclusão.

Esta situação é evitada através das seguintes directivas de pré-processamento. Por exemplo para uma dada especificação de Pilhas de Caracteres `pilhaChar.h`.

```
#ifndef PILHAS
#define PILHAS

void pop(pilha p);
char top(pilha p);
(...)

#endif
```

A macro `PILHAS` começa por não estar definida, mas aquando da primeira inclusão `PILHAS` passa a estar definida (o seu valor não é relevante). Numa inclusão posterior toda a definição da classe é ignorada não dado azo a erros de re-definição.

B.1.2 Opções de Compilação

Um qualquer compilador de *C* tem um conjunto extenso de opções que permitem modificar a forma como a compilação se vai processar. No que se segue vai-se apresentar algumas das opções mais usuais para o compilador *GNU* C *Compiler*[1].

[1] `gcc --help`

Opção	Significado
-o <nomeFicheiro>	atribuir o nome ao ficheiro executável.
-c <nomeFicheiro.cpp>	cria somente o código máquina, isto é, não faz os passos de agregação e criação de um executável.
-Wall	(*Warnings all*) Os avisos sobre construções que alguns utilizadores podem achar questionáveis e que são facilmente evitáveis, passam a ser emitidos. Um conjunto alargado de avisos passa a ser emitido.
-I<directório>	Adiciona o directório em questão à lista de directórios a serem pesquisados à procura de ficheiro de cabeçalhos (*header files*, .hpp).
-l<biblioteca>	Pesquisa, para agregação, uma dada biblioteca.
-L<directório>	Adiciona o directório em questão à lista de directórios a serem pesquisados à procura de bibliotecas (ver opção -l).

B.1.3 Utilizar uma Biblioteca

A crescente modularização na construção de programas, de forma quase natural surge a necessidade de construir não um programa, entidade fechada, mas uma biblioteca (ou pelo menos um nó, «livro» , de uma biblioteca). Designa-se por biblioteca (*library*) um, ou mais módulos, que implementam um dado conjunto de funcionalidades e que podem ser agregadas aos nossos programas. Além da biblioteca padrão do *C*, a *Standard Library* existem muitas outras à nossa disposição. Por exemplo:

GMP *The GNU Multiple Precision Arithmetic Library.* A GMP é uma biblioteca livre para a aritmética de precisão e gama de variação arbitrárias. Implementa inteiros com sinal, racionais, e reais. As principais aplicações da biblioteca GMP são: sistemas criptográficos; segurança da Rede; sistemas algébricos, entre outras.

http://gmplib.org/.

Boost A biblioteca *Boost* é um esforço colectivo para providenciar uma biblioteca livre de código aberto e verificada pela própria comunidade de utilizadores. Providencia um conjunto alargado de funcionalidades.

http://www.boost.org/.

NAG *Numerical Algorithms Group Library*, é uma biblioteca comercial com uma vasta gama de algoritmos matemáticos e estatísticos.

www.nag.co.uk/numeric/CL/CLdescription.asp.

GTK+ *the GIMP Toolkit*, é uma biblioteca, multi-plataforma para a criação de interfaces gráficos.

http://www.gtk.org/.

Além destas existem muitas outras com diferentes graus de aplicabilidade, funcionalidade e qualidade.

Como é que podemos então criar, e usar, a nossa própria biblioteca? Como foi dito acima uma biblioteca é algo que providencia um dado conjunto de funcionalidades e que podemos agregar aos nossos programas (a exemplo do que fazemos com a *STL*).

Como usar

Para podermos usar uma biblioteca temos de:

- ter acesso às declarações dos métodos contidos na biblioteca (*header files*);
- ter acesso ao código, na forma de um ficheiro já pré-compilado;
- adicionar as opções '`-l`', '`-L`' e '`-I`' ao comando de compilação. A primeira destas opções indica-nos o nome da biblioteca (sem o prefixo '`lib`'). A segunda opção indica a directoria aonde se encontra a biblioteca (se não estiver integrada no sistema). A última opção indica-nos aonde se encontram as *header files*.

A opção de compilação '`-l`' refere-se ao nome da biblioteca. É no entanto de notar que, por convenção, os nomes das bibliotecas começam todos por '`lib`', por exemplo `libBibMPI.so`, sendo que o prefixo '`lib`' não faz parte do nome da biblioteca, isto é, ter-se-ia `-lBibMPI`.

As bibliotecas podem ser de dois tipos no que diz respeito à sua utilização.

Bibliotecas Dinâmicas consistem de métodos que são pré-compilados e que são carregadas somente aquando da execução do programa. A sua extensão em sistemas Linux é a extensão '`so`', de objectos partilhados (*shared object*).

Bibliotecas Estáticas consistem de métodos que são pré-compilados e agregados ao programa aquando da compilação deste último. A sua extensão usual em sistemas Linux é a extensão '`a`', de arquivo (`archive`).

A grande vantagem das bibliotecas dinâmicas é a sua separação do programa compilado, só na altura da execução é que são chamadas a intervir. As vantagens desta aproximação são múltiplas:

- O ficheiro (executável) do programa não contém o código da biblioteca, em consequência é de menor dimensão do que no caso contrário (bibliotecas estáticas).
- A biblioteca pode sempre ser actualizada sem que isso signifique uma recompilação do programa que a usa.
- A biblioteca pode ser usada por muitos programas, inclusive simultaneamente.

A desvantagem advêm da dependência do programa (para a sua execução) da biblioteca. Num sistema em que a biblioteca dinâmica não esteja instalada os programas que dependem dela não funcionarão.

Isto é, se se quer providenciar uma solução auto-contida, um programa que para funcionar não necessita de mais nada do sistema. Então deve-se construir e usar bibliotecas estáticas. No caso contrário devem-se usar bibliotecas dinâmicas.

B.2 Makefile

O programa `make` é um utilitário presente em todos os sistemas do tipo *Unix* (*Linux, MacOS, etc.*) assim como em outros sistemas em que se instale um compilador de *C*. Este programa permite automatizar o processo de compilação, sendo possível especificar o que se

pretende através da construção de um ficheiro designado `Makefile`. De seguida apresenta-se, de forma sumária, o programa `make` e a forma como construir o ficheiro `Makefile`. Para uma exposição mais completa consulte (Mecklenburg, 2004; Darcano, 2007), ou aceda ao manual do programa[2].

B.2.1 O Programa `make`

O programa `make` é uma maneira muito conveniente de gerir grandes programas ou grupos de programas. Quando se começa a escrever programas cada vez maiores e visível a diferença de tempo necessário para recompilar esses programas em comparação com programas menores. Por outro lado, normalmente trabalha-se apenas em uma pequena parte do programa (tal como uma simples função que se está depurando), e grande parte do resto do programa permanece inalterada.

O programa `make` ajuda na manutenção desses programas observando quais partes do programa foram mudadas desde a última compilação e recompilando apenas essas partes.

Para isso, é necessário que se escreva uma «`Makefile`», que é um arquivo de texto responsável por dizer ao programa make "o que fazer" e contém o relacionamento entre os arquivos fonte, objecto e executáveis.

Outras informações úteis colocadas no `Makefile` são as «`flags`» que precisam ser passados para o compilador e o agregador (*linkador*), como directórios onde encontrar arquivos de cabeçalho (arquivos `.hpp`), com quais bibliotecas o programa deve ser agregado, etc. Isso evita que se precise escrever linhas de comando enormes incluindo essas informações para compilar o programa.

O programa `make` pode ser invocado de duas formas distintas (com '`$>`' a representar a linha de comandos do sistema):

- `$> make`, neste caso o primeiro dos objectivos contidos no ficheiro `Makefile` é executado.

- `$> make <objectivo>`, neste caso especifica-se qual o objectivo que se pretende executar.

Vejamos então como construir um ficheiro `Makefile`.

B.2.2 O Ficheiro `Makefile`

Uma `Makefile` contém essencialmente comentários, «macros» (regras de substituição) e objectivos (*targets*).

Comentários: Os comentários são delimitados pelo carácter "#" e pelo fim-de-linha respectivo. Por exemplo

```
# Análise sintáctica dos ficheiros de resultados
```

[2]`man make`; `http://www.gnu.org/software/make/manual/make.html`

Macros As macros são um simples mecanismo de substituição sintáctica. Permitem ajustar de uma forma simples o processo de compilação a diferentes ambientes de compilação. Permitem também a construção de um ficheiro mais fácil de ler e compreender. Por exemplo.

```
FLAGS   = -lfl -lm
CC      = gcc
OBJS    = lex.yy.c nGeometricSteps.tab.c
```

As macros são especificadas da seguinte forma:

```
<nome> = <valor>
```

sendo que por convenção (usualmente aceite) os nomes das macros são escritos utilizando somente maiúsculas.

Na `Makefile`, expressões da forma `$(<nome>)` ou `${<nome>}` são automaticamente substituídas pelo valor correspondente.

Objectivos Os objectivos determinam a acção a ser efectuada quando se executa o programa `make`. Como foi dito acima se o «chamar» do programa for feito sem argumentos o primeiro dos objectivos é aquele que vai ser processado, caso contrário só o objectivo invocado é que é processado. No caso do objectivo invocado não estar especificado na `Makefile` ocorre uma situação de erro e nada é processado. Por exemplo:

```
all:    nGeometricSteps

clean:
        -rm nGeometricSteps lex.yy.c nGeometricSteps.tab.*
```

o carácter '-' antes do comando `rm` (*remove*, apagar ficheiros) tem como finalidade o forçar a continuação do comando mesmo na eventualidade da ocorrência de um erro, por exemplo um dos ficheiros que se pretende apagar não existir.

Os objectivos são especificados da seguinte forma:

```
<objectivo1> <objectivo2> ... : <dependência1> <dependência2> ...
<espaço_tabular> <comando1>
<espaço_tabular> <comando2>
 ...
```

é de notar que o espaço tabular (tecla *Tab*) é obrigatório.

Caso uma das linhas respeitantes a um dos comandos seja muito comprida pode-se continuar a mesma (para efeitos de facilitar a sua leitura) por tantas linhas quanto o necessário. Neste casos é obrigatório colocar o carácter '\' (linha de continuação) entre no fim das linhas que tenham continuação.

Um tipo de objectivo especial é o assim designado, falsos objectivos (*phony target*). Este tipo de objectivos não está associado a um nome de ficheiro e é usualmente usado para evitar eventuais confusões entre nomes de ficheiros e nomes de objectivos. Por exemplo é usual ter-se:

```
.PHONY : clean all
```

Macros Especiais O programa `make` tem um conjunto muito grande de macros pré-definidas[3] das quais convêm destacar as seguintes:

- `$@`, é o nome do ficheiro a ser produzido (nome do objectivo);
- `$?`, são os nomes dos dependências que foram alteradas;
- `$<`, é o nome do ficheiro que causou a acção;
- `$*`, é o prefixo comum aos ficheiros resultantes e suas dependências.

Exemplos de `Makefiles` Um primeiro exemplo englobando os exemplos usados anteriormente:

```
FLAGS   = -lfl -lm
CC      = gcc
OBJS    = lex.yy.c nGeometricSteps.tab.c
LEX     = flex
YACC    = bison -d


all:      nGeometricSteps

clean:
        -rm nGeometricSteps lex.yy.c nGeometricSteps.tab.*

nGeometricSteps: nGeometricSteps.l nGeometricSteps.y
          bison -d $@.y
          flex $@.l
          ${CC} ${OBJS} ${FLAGS} -o $@
```

esta `Makefile` refere-se à automatização do processo de construção de um reconhecedor sintáctico através da utilização dos programas *flex* e *bison*. Neste caso temos uma `Makefile` com, basicamente, um só objectivo.

Um segundo exemplo pode vê-lo como uma `Makefile` para automatizar a compilação de todos os exercícios, acrescentando objectivos à medida que resolvemos novos exercícios.

```
CC      = gcc

.PHONY : clean all

all:      exer15 exer19

clean:
        -rm *.o

exer15:  exer15.o
         ${CC} -o $@ exer15.o

exer19:  exer19.o
         ${CC} -o $@ exer19.o
```

[3] Para saber quais fazer `prompt make -p`

B.3 Ambientes Integrados de Desenvolvimento

Ambientes Integrados de Desenvolvimento, IDE, sigla formada pelas iniciais de *Integrated Development Environment* são sistemas que procuram responder a algumas das questões importantes que se colocam aquando do desenvolvimento de programas:

- a escrita dos programas através de um editor de textos dedicado;
- a compilação do programa, eventualmente envolvendo mais (muito mais) do que um ficheiro;
- a depuração dos erros;
- a documentação.

Vejamos isso ponto a ponto.

B.3.1 Editor de Textos Dedicado

Antes de mais é de esclarecer que estamos a falar de um editor de textos e não de um processadores de texto. Estes últimos manipulam o texto acrescentando muita informação necessária ao processamento do texto, nomeadamente todas as questões relacionadas com a formatação, o que é totalmente inapropriado quando se está a querer construir algo que vai ser submetido ao tratamento automático de um compilador.

Como primeira conclusão: não tente usar um processador de textos para a escrita de um qualquer programa. É necessário usar um editor de textos.

Um editor de texto dedicado é um editor (não acrescenta nada ao ficheiro contendo o código do programa) mas que conhece a linguagem que se está a usar. Isto é um editor deste tipo tem um conhecimento da linguagem que passa por conhecer a sua estrutura léxica e gramatical, o que leva a que possa:

- fazer a verificação sintáctica à medida que se escreve. Em geral isso reflecte-se num código de cores, isto é, a atribuição diferentes cores aos tipos de dados, aos identificadores da linguagem, aos comentários, às sequências de caracteres, etc;
- fazer a verificação gramatical à medida que se escreve, o que se reflecte na forma como a indentação (os diferentes níveis de alinhamento do texto) é feita;

Pode parecer pouco mas, dado que um programa não é mais do que um texto numa dada linguagem, texto esse que vai ser processada automaticamente por um programa, o qual é muito pouco, ou mesmo nada, tolerante aos erros sintácticos e gramaticais, é fácil de perceber que toda a ajuda é muito importante.

Quando algo não está na cor que é suposto estar. . . algo está sintacticamente errado, por exemplo um identificador da linguagem que está escrito de forma errada.

Quando algo não se ajusta (indenta) para a forma correcta. . . algo está gramaticalmente errado, por exemplo um esquecimento de um ";" no fim de uma instrução.

B.3.2 Compilação & Makefiles

Desde a compilação simples através da simples escolha de uma opção, até à construção e utilização, com diferentes graus de automatismo, de "makefiles" para a compilação de programas multi-ficheiros.

B.3.3 Depuração de Erros

Desde a indicação e posicionamento automático, da linha aonde os erros ocorrem, até à ligação com programas específicos para o depuramento de erros.

B.3.4 Documentação

No caso do *C* uma ajuda importante é o de mostrar a estrutura do programa com as suas diferentes funções. Em alguns casos (e em certas linguagens), é mesmo possível gerar parte da documentação de forma automática.

B.3.5 Alguns IDEs para o *C*

É importante que o ambiente de programação escolhido seja:

- Apropriado (talvez mesmo especializado) para o ambiente de trabalho (leia-se empresa) em que se está.
- Multi-plataforma, isto é, esteja disponível em diferentes plataformas computacionais computador/sistema operativo.
- Multi-linguagem, isto é, que seja capaz de se adaptar aos requisitos das diferentes linguagens de programação.

Obviamente que estes dois requisitos aplicam-se a duas situações bem distintas. O primeiro quando se está a trabalhar numa equipa com hábitos e objectivos bem definidos, aí é importante que, para uma maior eficiência, todos utilizem a mesma plataforma de desenvolvimento.

Quando se tem uma situação mais difusa é importante que utilizemos uma plataforma que melhor se adapte às mudanças de plataforma computacional usada e/ou de linguagem de programação usada.

Três aproximações que se adaptam a estes requisitos são:

Emacs neste caso estamos perante um editor de textos não exactamente um IDE completo. Dado ser programável (em EmacsLisp) é altamente configurável, o seu modo específico para o *C* é muito bom. Dado não ser um IDE as tarefas de compilação e de depuração de erros não estão integradas. De aprendizagem algo difícil é no entanto um dos (se não o) editor de texto mais flexível e poderoso existente. Multi-plataforma, multi-linguagens de programação, código aberto. `http://www.gnu.org/software/emacs/`.

Geany é um IDE bastante completo, o seu editor, embora não tão poderoso como o (X)emacs é mesmo assim muito bom. A construção de Makefiles não é automática, a ligação a um depurador de erros é possível embora necessite configuração. Muito fácil de usar. Multi-plataforma, multi-linguagens de programação, código aberto. `http://www.geany.org/`.

CodeBlocks mais poderoso do que os anteriormente referidos, incorpora a noção de projecto com a automatização da compilação (que não uma Makefile). É um pouco mais complexo que o anterior. Multi-plataforma, específico para a linguagem C, código aberto. `http://www.codeblocks.org/`

B.4 Comandos para a Consola

Na compilação e execução de programas é algumas vezes necessário recorrer à *consola*, local aonde é possível executar programas sem recurso a um interface gráfico. Eis alguns dos comandos necessários para "navegar" no sistema de ficheiros e colocar a executar os programas.[4]

ls — lista o conteúdo de um directório (*listing*).

ls -la — lista o conteúdo de um directório com um nível de detalhe mais elevado ('l') e mostrando os ficheiros escondidos ('a').

cd *dir* — muda para o directório *dir* (*change directory*).

cd — muda para o directório de base (*home*).

cd .. — muda para o directório acima do actual.

pwd — mostra o directório actual (*print working directory*).

more *ficheiro* — mostra o conteúdo de um ficheiro (de texto).

./*executável* — executa (coloca a funcionar) o *executável*.

[4]Os comandos descritos referem-se à consola *bash* do sistema operativo *Linux*. Comandos semelhantes encontram-se disponíveis noutras consolas/sistemas operativos.

Apêndice C

Exemplos

Nesta apêndice apresenta-se o código referente aos vários exemplos descritos nos diferentes capítulos destes apontamentos.

C.1 Exemplos Referentes ao Capítulo 3

A classe `<limits>` dá-nos acesso aos limites númericos para cada um dos tipos númericos disponíveis. O programa a seguir apresentado dá-nos um exemplo de como obter os limites para a versão do compilador e sistema operativo que se está a usar.

```
/*
 * Tamanho número de bytes) e gama de valores para
 * alguns tipos básicos
 */

#include <stdio.h>
#include <limits.h>
#include <float.h>

int main () {
 printf("Número de 'bytes' por 'char': %d\n",sizeof(char));
 printf("Número de 'bytes' por 'short': %d\n",sizeof(short));
 printf("Gama de variação: %d a %d\n", SHRT_MIN,SHRT_MAX);
 printf("Número de 'bytes' por 'int': %d\n",sizeof(int));
 printf("Gama de variação: %d a %d\n", INT_MIN,INT_MAX);
 printf("Número de 'bytes' por 'long int': %d\n",sizeof(long));
 printf("Gama de variação: %ld a %ld\n", LONG_MIN,LONG_MAX);
 printf("Número de 'bytes' por 'float': %d\n",sizeof(float));
 printf("Gama de variação: %e a %e\n", FLT_MIN,FLT_MAX);
 printf("Número de 'bytes' por 'double': %d\n",sizeof(double));
 printf("Gama de variação: %e a %e\n", DBL_MIN,DBL_MAX);
 printf("Número de 'bytes' por 'long double': %d\n",sizeof(long double));
 printf("Gama de variação: %Le a %Le\n", LDBL_MIN,LDBL_MAX);

 return(0);
}
```

Para obter a versão do compilador que se está a usar basta usar a opção `-v`. Por exemplo:

```
$> gcc -v
gcc version 5.2.1 20151028 (Debian 5.2.1-23)
```

Apêndice D

Forma Normal Estendida de Backus-Naur

A forma normal Estendida de Backus-Naur (**E**xtended **B**ackus-**N**aur **F**orm) é um simbolismo muito usado na área da descrição das linguagens, nomeadamente na descrição da sintaxe das linguagens computacionais (Pattis, 1980).

As regras de derivação têm um lado direito e um lado esquerdo, separadas pelo símbolo de derivação (::= ou ⇐), por exemplo:

função ::= <tipo> <identificador> `(` <listaArgumentos> `)` { <corpoDaFunção> }

Esta regra que define a sintaxe da definição de uma função em *C* lê-se como *função* é definida por ... o lado direito da regra de derivação.

Do lado esquerdo aparece sempre um só identificador o qual dá nome à regra e, como já se disse, é o objectivo da definição.

No lado direito, o qual define associada à regra de derivação é uma combinação das formas de controle especificadas na tabela D.1.

Sequência	os elementos aparecem da esquerda para a direita, sendo que a ordem relativa é importante
Alternativa	os elementos são separados por uma barra vertical |; um dos elementos é escolhida dessa lista de opções, a ordem relativa não é importante
Opcional	um elemento opcional é colocado entre parêntesis rectos («[» e «]»), o elemento pode ser considerado ou simplesmente ignorado
Repetição	um elemento é colocado entre chavetas («{», «}») significando que pode ser repetido zero ou mais vezes

Tabela D.1: Formas de Controlo do Lado Direito das Regras EBNF

Sempre que os elementos do lado direito ainda carecem de definição, colocam-se entre os símbolo relacionais de menor e de maior (<, >), a sua definição terá de ser encontrada numa outra regra de derivação. Os *literais*, isto é, os elementos a serem interpretado de forma literal, são escritos utilizado o tipo de letra «máquina de escrever» («typewriter», tipo de letra mono-espaço) e devem ser escritos exactamente como está na regra.

A forma de aplicar este formalismo na validação da escrita de programas é feita através da associação de padrões, entre o padrão definido por uma definição concreta em, neste caso, da linguagem *C* e a regra de derivação associada ao elemento que se está a definir.

Por exemplo, será que a seguinte definição de função em *C* está correcta?

```
void troca(int *a,int *b) {
  *a = *a*(*b);
  *b = *a/(*b);
  *a = *a/(*b);
}
```

Analisando-o à luz regra definida acima temos

Elemento *C*	Elemento EBNF	Associação
`void`	<tipo>	sim, `void` é um tipo em *C*
`troca`	<identificador>	sim, `troca` é um identificador possível em *C*
`(`	`(`	sim, é um literal
`int *a,int *b`	<listaArgumentos>	sim, ver regra específica
`)`	`)`	sim, é um literal
{	{	sim, é um literal
*a = *a*(*b); ...	<corpoDaFunção>	sim, ver regra específica
}	}	sim, é um literal

Tendo havido um «encaixe» perfeito entre a regra de derivação e o fragmento de *C*, podemos afirmar que a definição da função `troca` está correcta, do ponto de vista da sintaxe do *C*.

Apêndice E

Exercícios Práticos

E.1 Leitura e Escrita

```
int printf(char *format, arg1, arg2, ...)
int scanf(char *format, ...)
```

uma sequência de caracteres que especifica o formato de saída (entrada) e depois uma lista de argumentos.

Cada especificação de conversão (ver Tabela E.1) começa com % e termina com um carácter de conversão. Entre o % e o carácter de conversão pode-se incluir, em ordem:

- Um sinal de menos, que especifica o ajuste à esquerda do argumento convertido.
- Um número que especifica a largura mínima do campo. O argumento será impresso em um campo, pelo menos, com essa amplitude. Se necessário será preenchido à esquerda (ou à direita, se o ajuste esquerdo for chamado) para compensar a largura do campo.
- Um ponto final '.', que separa a largura do campo da precisão.
- Um número, a precisão, que especifica o número máximo de caracteres a serem impressos a partir de uma ≪string≫, ou o número de dígitos após o ponto decimal de um valor de vírgula flutuante, ou o mínimo número de dígitos para um número inteiro.
- Um 'h' se o número inteiro for impresso como um ≪short≫, ou 'l' (éle) se for um ≪long≫.

1 Edite, compile e execute os seguintes programas:

1. Escrever a frase (sequência de caracteres) "Olá Mundo".

```
// Olá mundo
#include <stdio.h>

int main() {
  printf("Olá Mundo!\n");
  return 0;
}
```

2. Escrever o inteiro "23" e o resultado de uma expressão com inteiros, alinhados à direita.

Carácter	**Tipo de Argumento**	**Escrito como**
d,i	`int`	número decimal
o	`int`	número octal sem sinal (sem zeros à esquerda).
x, X	`int`	número hexadecimal sem sinal (sem '0x' ou '0X' à esquerda), utiliza-se 'abcdef' ou 'ABCDEF' para 10,. . . , 15
u	`int`	número decimal sem sinal
c	`int`	um carácter
s	`char *`	escreve os caracteres de uma sequência de caracters (*string*) até o carácter '\0' ou até ao números de caracteres dado pela precisão
f	`double`	[-]m.dddddd, aonde o número de 'ds' é dado pela precisão (por omissão, 6)
e,E	`double`	[-]m.dddddde±xx ou [-]m.ddddddE±XX, aonde o número de 'ds' é dado pela precisão (por omissão, 6)
g,G	`double`	usar %e ou %E se o expoente é menor do que -4, ou maior ou igual do que a precisão; em caso contrário usar %f. Os zeros, assim como o ponto decimal, são omitidos quando no fim do número
p	`void *`	ponteiro (representação dependente da implementação)
%		sem argumento, escrever um %.
lf	double	**leitura** de um real do tipo `double`.
Lf	long double	**leitura** de um real do tipo `long double`.

Tabela E.1: Especificadores de Formato

```
// Escrita de inteiros (alinhados à direita)
#include <stdio.h>

int main() {
  printf("%5d\n",23);
  printf("%5d\n",2+3);
  return 0;
}
```

3. Escrever o inteiro "23" e o resultado de uma expressão com inteiros, alinhados à esquerda.
4. Escrever o real "23,5" e o resultado de uma expressão com reais, alinhados à esquerda.
5. Escrever o real "23,5" e o resultado de uma expressão com reais, alinhados à esquerda. Reais em formato cientifico.
6. Escrever o resultado de uma expressão com inteiros formatando a saída de forma a mostrar a expressão, e.g. "$2 + 3 = 5$".
7. Ler dois argumentos do tipo inteiro e escrever o resultado da sua soma.
8. Escrever uma mensagem a pedir a dois argumentos do tipo inteiro e escrever o resultado, formatado, da sua soma.
9. Leia três caracteres e escreva-os por ordem inversa.
10. Leia um inteiro, um carácter, um real e de seguida escreva-os pela mesma ordem.

E.2 Instruções de Atribuição

2 Edite, compile e execute os seguintes programas:

1. Ler dois valores inteiros, calcular (e guardar) a sua soma e escrever a soma.
2. Escreva um programa para calcular a área de um quadrado
3. Altere o programa anterior de modo a calcular a área de :
   - um rectângulo;
   - um triângulo.
4. Escreva um programa para calcular a hipotenusa de um triângulo rectângulo, dados os seus dois catetos.
5. Altere o programa anterior de modo a:
   - escrever o comprimento da hipotenusa num campo de comprimento 7;
   - escrever o comprimento da hipotenusa em notação normal com três casas decimais;
   - escrever o comprimento da hipotenusa em notação científica com três casas decimais.

E.3 Tipos de Dados Simples, Inteiros e Reais

`int`, `float`, `double`. Algumas das operações requerem o uso da biblioteca `math.h`.

`http://www.gnu.org/software/libc/manual/html_node/`

Algumas das funções contidas em `math.h`. Têm um ou dois argumentos do tipo `double` e devolvem um resultado do tipo `double`

`sin(x)`	seno de x, com x em radianos
`cos(x)`	coseno de x, com x em radianos
`atan2(y,x)`	arco-tangente de y/x, em radianos
`exp(x)`	função exponential (base e)
`log(x)`	logaritmo natural (base e) de x ($x > 0$)
`log10(x)`	logaritmo decimal (base 10) de x ($x > 0$)
`pow(x,y)`	x^y
`sqrt(x)`	$\sqrt{x}$ ($x \geq 0$)
`fabs(x)`	$\lvert x \rvert$

3 Diga o tipo e calcule o valor de cada uma das expressões seguintes:

1. 2 + 3 ∗ 2 + 3 ∗ 4
2. pow(3.0,2.0)
3. pow(2.0,2.0)
4. ceil(-99.9)
5. floor(99.9)
6. ceil(99.9)
7. trunc(99.9)
8. 10 / 3
9. 128 / 3 % 5
10. 128 % 5 / 3
11. 193 % 19 / 3 ∗ 127
12. (8 / 5) / (8 % 5)

Escreva programas que lhe permitam verificar a correcção das suas respostas.

4 Tendo em conta a tabela:

variável	tipo	valor
a	double	5.7
b	double	8.2
c	int	7
d	int	4

determine o tipo e o valor de cada uma das expressões:

1. sqrt(a+b) / (c+d)
2. c ∗ (c % d)
3. (c % d) / 2
4. trunc(a-b)
5. trunc(a-b) / c
6. exp(2 ∗ log(c))
7. (c / d) / (-a)
8. trunc(fabs(cos(x))) % 1
9. 2 ∗ (b - a)

Escreva programas em que lhe seja possível verificar a correcção das suas respostas.

5 Considere a seguinte expressão:

$$16.4 - 51.0 \,/\, 4.8 * 0.6 + 74.92$$

Coloque o número mínimo de parentesis na expressão de modo que as operaçoes sejam efectuadas pela ordem indicada em cada uma das alíneas:

1. $- + * /$
2. $* + / -$
3. $/ - + *$

6 Proceda de modo análogo ao do problema anterior relativamente à expressão

$$22 + 16 * 100 \ \% \ 12 - 9 \ / \ 4$$

e de modo que as operaçoes sejam efectuadas pelas seguintes ordens:

1. + % * / -
2. / - % * +
3. - % / * +

7 Supondo que as seguintes declarações tenham sido feitas, diga se cada uma das atribuições é ou não válida, e nesse caso, proponha uma modificação necessária para que a referida instrução se torne válida :

```
int m, n;
float a, b;
```

1. m = trunc(b);
2. m = trunc(b) + a;
3. p = m + n;
4. m = n % a;
5. b = 2.99 * 10;
6. b = 2.99 * exp(10,9);
7. n = a - trunc(a);
8. maElec = 0.91095E-27;
9. 2ab = 2 * a * b;
10. a := (.54E+2-1);
11. a := m / n;
12. b := 4 + 2a;

Escreva programas em que lhe seja possível verificar a correcção das suas respostas.

8 Escreva uma instrução de atribuição para cada uma das seguintes acções:

1. O contador I é incrementado de uma unidade;
2. M é uma cópia do valor de N;
3. Q é o valor da divisão inteira dos inteiros I e J;
4. X é o valor da divisão real dos inteiros I e J;
5. I é o valor arredondado do real X;
6. I é o maior inteiro inferior ou igual a X, positivo;
7. M é o inteiro mais próximo da média dos reais A e B;
8. A variável t20 toma o valor da tangente de 20 graus;
9. Dado n, inteiro não nulo, a variável inteira SINAL toma o valor 1 se n for positivo e -1 se n for negativo;
10. U toma o valor do algarismo das unidades do real X;
11. R toma o valor de $\sqrt{x}$ se x for não negativo, ou de $\sqrt{-x}$, caso contrário;
12. Z toma o valor de módulo de Y elevado a X;

13. ALFA é o ângulo (em graus) cuja tangente é x;
14. $y \leftarrow y + 4x + 3x^2 + 2x^3 + x^4$ (procure minimizar o numero de operações);
15. $y \leftarrow \frac{\sin(ax)}{2a\cos^2(ax)}$
16. A área de um polígono regular de n lados de comprimento b é dada por:

$$area \leftarrow \frac{1}{4}nb^2\text{cotg}\frac{\pi}{n}$$

17. $raiz5de3 \leftarrow \sqrt[5]{x^3}$
18. $z \leftarrow \log_6(3x^2 + 6)$
19. $y \leftarrow \frac{\tan(2x)x^2}{2\sqrt{|x+5|}}$
20. $fraccao \leftarrow a + \frac{d}{e+\frac{d}{e+\frac{f}{g}}}$
21. $x \leftarrow \frac{(\sqrt{a^2+b^2}-a)^4}{\sqrt[4]{a^2+b^2}}$

Escreva programas que lhe permitam verificar a correcção das suas respostas.

E.4 Tipos de Dados Simples, Caracteres ("char") e Lógicos

As variáveis e constantes do tipo `char`, por exemplo `'x'`, contêm um só elemento na codificação do alfabeto de base da sistema computacional em que se está (*utf8*, *iso-8859-1*, etc.). A representação interna do tipo `char` é um tipo inteiro com 8 bits de comprimento.

O tipo lógico não existem em C. O tipo inteiro é usado tendo em conta a convenção, $0 \leftrightarrow$ false, $\neq 0 \leftrightarrow$ true.

9 Diga o tipo e calcule o valor de cada uma das expressões seguintes, supondo:

```
int k;
int p,q,r,s;
float x;
```

1. ((4-1)-(2+1)) % 4 + sqrt(4) = 8
2. !(p —— q) = !p && !q
3. !(p && q) = !(!p —— !q)
4. (p && (q && !q)) —— !(r —— (s —— !s))
5. !((3 % 2 = 1) && (7 < 5))
6. (ceil(-65.3) < trunc(-65.3)) && p
7. (ceil(sin(x))=0) —— (fabs(ceil(sin(x)))=1)
8. 'E'+1
9. 'D'-'A'
10. '8'-'0'
11. 'A'+2

Escreva programas que lhe permitam verificar a correcção das suas respostas.

10 Supondo que as seguintes declarações tenham sido feitas, diga se cada uma das instruções é ou não válida e, neste caso, proponha uma modificação necessária para que a referida instrução se torne válida:

```
const char espaco = ' ';
int  m, n;
float a, b;
int p, q; // boolean
char c1,c2;
```

1. c1 = espaco;
2. c1 = 'a';
3. c1 = c2;
4. c1 = 'c2';
5. c = c2;
6. c2 = 'a';
7. m = m-'0';
8. p = (m + n) > 0;
9. 'a' = 'b'+1;
10. b = c1+c2;
11. m = c1+1;
12. n = c1+(5+1);
13. p = q && ((c1+1) <> 'a');
14. p = q && ('a' >= 40);

Escreva programas que lhe permitam verificar a correcção das suas respostas.

11 Escreva uma instrução de atribuição para cada uma das seguintes acções:

1. A variável lógica L é verdadeira se e só se L1 e L2 forem ambas falsas;
2. A variável lógica MAIOR é verdadeira se e só se X é maior que Y e é falsa no caso contrário;
3. A variável lógica L é verdadeira se e só se L1 é verdadeira mas não L2;
4. LOGIC é verdadeira se e só se M for o dobro de N;
5. BOOL é verdadeira se e só se os inteiros K e M forem iguais, em valor absoluto;
6. A variável CONSOANTE é verdadeira se e só se a variável carácter LETRA for uma consoante minúscula;
7. A variável XOR é verdadeira se e só se apenas uma das variáveis B1 ou B2 for verdadeira;
8. A variável lógica PAR é verdadeira se e só se o inteiro N é par;
9. A variável lógica BISSEXTO é verdadeira se e só se a variável inteira ANO for divisível por 4 mas não por 100 ou então for divisível por 400;
10. A variável lógica MULT é verdadeira se e só se I for múltiplo de J (ambos inteiros);
11. A variável lógica VOG é verdadeira se e só se o carácter C for uma vogal;
12. COMPLEX é verdadeiro se e só se a equação $ax^2 + bx + c = 0$ tem raízes complexas;
13. Sendo N uma variável do tipo carácter, o booleano DIGITO é verdadeiro se e só se N representar um algarismo decimal;
14. A variável inteira PAR vale 1 se n for par e 2 se n for ímpar;
15. A variável inteira ALTERNADA toma o valor de $(-1)^n$ com n inteiro;

16. Sendo c um carácter que representa um dígito, atribuir a N o valor numérico desse dígito (N pertencente a $\{0, \ldots, 9\}$);

Escreva programas que lhe permitam verificar a correcção das suas respostas.

12 Elabore os seguintes programas (separando as várias partes em secções de texto distintas separadas por curtos comentários, e.g.
{leitura} ...{cálculo} ...{escrita} ...):

1. calcule o lucro obtido na venda de um produto: leia o nome do produto, o número de unidades vendidas, o preço de custo (preco_custo), o preço de venda (preco_venda) (ambos os preços são unitários) e o IVA a pagar (em percentagem - inteiro), cacule o lucro total apurado e escreva o nome do produto e o respectivo lucro na forma ...... $ .. (O IVA que o comerciante tem de pagar é calculado sobre a diferença entre o preço de custo e o de venda.)
2. calcule a área de um polígono de n lados de comprimento b, sabendo que $A = \frac{1}{4}nb^2\text{cotg}(\frac{\pi}{n})$.
3. calcule o valor das coordenadas polares RO e $teta$ correspondentes às coordenadas cartesianas x e y sabendo que $RO = \sqrt{x^2 + y^2}$ e $teta = \text{arctg}(\frac{y}{x})$
4. um terreno com 80m por 100m custou 1000 contos. Calcule e escreva o preço por metro quadrado de terreno.

E.5 A instrução condicional "if (*P*) *I1* else *I2*"

13 Preveja a saída do seguinte programa e introduza-o no computador para verificar se a saída é igual à que previu:

```
#include <stdio.h>

int main() {
  char letra;
  int n;

  printf("Escreva uma letra: ");
  scanf("%c",&letra);
  printf("O carácter ASCII seguinte à letra '%c' é '%c' e o anterior");
  printf(" é '%c'\n", letra, letra+1,letra -1);
  n = letra; // o código ASCII da letra
  if ((n >= 32) && (n <= 126)){ // a-zA-Z
    printf("Caracteres e inteiros são intermutáveis (um só byte)\n");
    printf("Exemplo: ");
    printf("%d - %d\n",letra,n);
    printf("%c - %c\n",n,letra);
  }
  else {
    printf("O valor dado não pertence ao intervalo indicado.\n");
  }
```

```
  return(0);
}
```

14 Escreva um programa que: leia dois números reais; escreva o valor do maior de entre eles.

15 Escreva um programa que: leia um número inteiro; determine se o número é par; escreva o número indicando se é par ou ímpar.

16 Escreva um programa que:

- leia um valor real, x
- calcule e escreva o valor da função:

$$f(x) = \begin{cases} \frac{1}{x}e^x & \text{se } x > 0 \\ e^{|x|} & \text{se } x \leq 0 \end{cases}$$

17 Escreva um programa que, dado x real, calcule $f(x)$ definida por:

$$f(x) = \begin{cases} e - e^{\cos x} & \text{se } x \in [0, 2\pi[\\ \log\cos x & \text{se } x \in [-2\pi, -\frac{3}{2}\pi[\bigcup]-\frac{\pi}{2}, 0[\end{cases}$$

18 Escreva um programa que, dado x real, calcule $f(x)$ definida por:

$$f(x) = \begin{cases} x(\cosh x)^{(2/x)} & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ x^2\log(\frac{1-x}{x})^2 & \text{se } x < 0 \end{cases}$$

Nota: $\cosh x = \frac{e^x+e^{-x}}{2}$

19 Elabore um programa que calcule as raízes da equação quadrática de coeficientes inteiros:

$$ax^2 + bx + c = 0$$

20 Considere a equação da velocidade no instante t ($t \geq 0$):

$$v(t) = \frac{\log|2 - 2t + t^2| - e}{\cos^2[3(t-1)] + e^{-t+1}}$$

Elabore um programa que indique se, num instante $t0$ dado, o objecto se desloca sobre a respectiva trajectória no sentido positivo ou negativo, ou se muda de sentido.

Nota: quando $v > 0$, o objecto desloca-se no sentido positivo; quando $v < 0$, o objecto desloca-se no sentido negativo; quando $v = 0$, o objecto muda de sentido.

21 Considere a seguinte definição de uma função real de duas variáveis reais:

$$f(x,y) = \begin{cases} e^{\sin x} + e^{\cos y} & \text{se } x \geq 0 \text{ e } y \geq 0 \\ e^{-\sin x} - e^{-\cos y} & \text{se } x < 0 \text{ e } y \geq 0 \\ \log_e |\sin x + \cos y| & \text{se } x < 0 \text{ e } y < 0 \\ \log_e |\cos x - \sin y| & \text{se } x \geq 0 \text{ e } y < 0 \end{cases}$$

Elabore um programa que devolva o valor da função para valores x e y, expressos em *graus*, fornecidos pelo utilizador.

22 Pretende-se um programa que converta entre quilómetros por hora, milhas por hora e nós. O programa deve perguntar ao utilizador pelo valor e pelas unidades. Por exemplo:

```
Qual o valor: 13.7894
Quais as unidades
     (k = kilómetros por hora, m = milhas por hora, n = nós): m
```

Ao que o programa deve responder com o resultado:

```
kilómetros por hora = 22.19189
nós = 11.98266
```

Recorde-se que: 1 milha = 1.609344 Km e 1 nó = 1.852000 Km por hora.

23 Pretende-se um programa que faça conversões entre graus e radianos. O programa deve perguntar ao utilizador qual o valor e as unidades do ângulo. Por exemplo:

```
Qual o valor do ângulo: 37.894
Quais as unidades (g = graus, r = radianos): g
```

Ao que o programa deve responder com o resultado:

```
Radianos = 0.6613751
```

Se o utilizador der um carácter não válido quando o programa pedir as unidades, deverá ser escrita uma mensagem de erro. Por exemplo:

```
Qual o valor do ângulo: 37.894
Quais as unidades (g = graus, r = radianos): w
ERRO: entrada não válida para unidades: w
```

Recorde-se que: π radianos = 180 graus. $\text{tg}(\pi/4) = 1$.

24 Dado um par de valores (número de bilhetes, tipo), referente ao número de bilhetes pretendido e o seu tipo. pretende-se saber qual o valor a pagar tendo em conta a seguinte tabela de preços dos bilhetes:

Tipo	**Preço**
1	100€
2	120€
3,4	200€
5	50€

E.6 Instruções de Repetição (Ciclos)

`while(`*P*`)` *I* — `do` *I1* `...` `while(`*P*`)` — `for(`*inicial*`;`*final*`;`*incremento*`)` *I*

25 Leia o seguinte programa e preveja a sua saída:

```
#include <stdio.h>

int main() {
  int i, p;

  // condições iniciais
  i = 1;
  p = 2;
  // ciclo
  while (p < 1000) {
    printf("%6d %10d\n",i, p);
    i = i + 1;
    p = p * 2;
  }

  return(0);
}
```

26 O algoritmo de Euclides permite calcular o maior divisor comum de dois números inteiros positivos, baseando-se na seguinte propriedade:

$$\text{mdc}(a, b) = \begin{cases} a & \text{se } b = 0 \\ \text{mdc}(b, a \bmod b) & \text{se } b \neq 0 \end{cases}$$

Elabore um programa seguindo os seguintes passos:

- peça ao utilizador dois números inteiros positivos e leia esses inteiros (a e b)
- enquanto $b \neq 0$
  - atribuir a *resto* o valor de a mod b
  - faça a tomar o valor de b
  - faça b tomar o valor de *resto*
- escreva os valores dados e o respectivo máximo divisor comum.

27 Elabore um programa que leia um número inteiro e verifique se é uma capicua, através dos seguintes passos:

- Peça ao utilizador que forneça um número inteiro
- Leia esse número inteiro, n
- Atribua a *ncópia* o valor de n

- Inicialize a variável *inverso* a zero
- Enquanto *ncópia* diferente de zero
  - atribua a *dígito* o valor do algarismo das unidades de *ncópia*
  - atribua a *inverso* o seu valor anterior multiplicado por 10 mais o valor de *dígito*
  - atribua a *ncópia* o valor da sua divisão inteira por 10
- Se o número dado for igual a *inverso*
  então escreve "é capicua"
  senão escreve "não é capicua"

28 Elabore programas capazes de receber do teclado uma sucessão de números inteiros $(x_i, i \in \mathbb{N})$. O último elemento da sucessão não é utilizado nos cálculos, servindo para indicar o fim da sucessão (elementos deste tipo denominam-se "sentinelas"). Para cada alínea estabeleça o valor da sentinela mais adequado e escreva o respectivo programa.

1. Contar o número de elementos de uma sucessão de termos positivos;
2. Calcular o produto dos elementos de uma sucessão;
3. Determinar o maior e o menor de entre os elementos de uma sucessão de termos ímpares;
4. Determinar o maior elemento e o número de vezes que ocorre numa sucessão de termos negativos;
5. Calcular a média dos elementos positivos e a média dos elementos negativos numa sucessão de termos ímpares.

29 Determine a média $\bar{x}$ dos i elementos da sucessão utilizando a seguinte fórmula:

$$\bar{x}_i = \begin{cases} 0 & \text{se } i = 0 \\ \bar{x}_{i-1} + \frac{x_i - \bar{x}_{i-1}}{i} & \text{se } i \geq 1 \end{cases}$$

30 Elabore um programa para calcular o valor da função cos, por desenvolvimento em série, desprezando termos de valor absoluto inferior a 10^{-8}, sabendo que:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \dots$$

31 O valor de π pode ser calculado através do chamado produto de Wallis:

$$\frac{\pi}{2} = \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \dots$$

Elabore um programa para o efeito, que irá construindo e multiplicando cada um dos factores, até que a diferença entre dois valores consecutivos seja, em módulo, inferior a 10^{-4}.

E.7 Funções: <tipo> <nome> (<lista argumentos>) {*I*}

32 São dados os seguintes tipos e declaração:

```
typedef struct planeta {
  char  nome[8];
  int   visivel;
  float raioOrbital;
} Planeta;

Planeta sistema[9];
```

que descrevem o Sistema Solar. Escreva um sub-programa que leia o seguinte ficheiro:

Nome	Visibilidade	Raio Orbital
Mercúrio	s	0.39
Vénus	s	0.72
Terra	s	1.0
Marte	s	1.5
Júpiter	s	5.2
Saturno	s	9.5
Úrano	n	19.2
Neptuno	n	30.1
Plutão	n	39.5

e que escreva o nome e o raio orbital dos planetas visíveis, da Terra, a olho nu.

33 Analise os seguintes programas e preveja a sua saída:

1.
```
#include <stdio.h>

int dobro(int a) {
  return(2*a);
}

int main() {
  int n;
  printf("Introduza um inteiro: ");
  scanf("%d",&n);
  printf("O dobro de %2d é %d\n", n,dobro(n));

  return(0);
}
```

2.
```
#include <stdio.h>

int trocar(float* a,float* b) {
  float aux;

  aux = *a;
  *a = *b;
  *b = aux;
```

```
}

int main() {
  float x = 2.5, y= 0.5;
  printf("x = %4.1f y = %4.1f\n", x,y);
  trocar(&x,&y);
  printf("x = %4.1f y = %4.1f\n", x,y);

  return(0);
}
```

3.
```
#include <stdio.h>
#include <math.h>

void lerDados(float* base, int* expoente) {
  printf("Escreva a base: ");
  scanf("%f",base);
  printf("Escreva o expoente: ");
  scanf("%d",expoente);
}

float potenciaInteira(float b, int e) {
  return(pow(b,e));
}

void escreveResultado(int e, float b, float r) {
  printf("Resultado: %4.2f^%d = %4.2f\n",b,e,r);
}

int main () {
  float b,r;
  int e;

  lerDados(&b, &e);
  r = potenciaInteira(b, e);
  escreveResultado(e,b,r);
}
```

34 Escreva um procedimento que troque, entre si, o valor de duas variávei, mas sem usar uma variável auxiliar (só com as operações aritméticas elementares).

35 Implemente as seguintes funções:

1. $$\begin{array}{rcl} \log : \mathbb{R}^+ \times A & \to & \mathbb{R} \\ (x,a) & \mapsto & \log_a(x) = \frac{\ln(x)}{\ln(a)} \end{array}$$
   com $A = \{x \in \mathbb{R} : x > 0 \text{e } x \neq 1\}$

2. $$\begin{array}{rcl} \text{senh} : \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & \text{senh}(x) = \frac{e^x - e^{-x}}{2} \end{array}$$

3. $$\begin{array}{rcl} f : \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & \begin{cases} \sqrt{x} & \text{se } x \geq 0 \\ \sqrt{-x} & \text{se } x < 0 \end{cases} \end{array}$$

4. $$\begin{array}{rcl} \text{argsech} :]0,1] & \to & \mathbb{R} \\ x & \mapsto & \text{argsech}(x) = \log(\frac{1+\sqrt{1-x^2}}{x}) \end{array}$$

36

1. Dados dois números inteiros positivos escreva um sub-programa que verifique se os dois números são ou não amigos, isto é, se cada um deles é igual à soma dos divisores próprios do outro.

2. Usando a alínea anterior faça um programa que escreva todos os pares de números amigos existentes entre 1 e 1000. Tente optimizar este programa.

E.8 Recursão

37 Elabore um programa que calcule os k primeiros termos da série harmónica:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} + \dots$$

38 Escreva um sub-programa que calcule o factorial de um número. Usando esse sub-programa escreva um programa que calcule o número de combinações de m, n a n sabendo que:

$$C_n^m = \frac{m!}{n!(m-n)!}$$

39 Escreva um programa que calcule o m.m.c de dois números inteiros positivos, baseando-se na seguinte propriedade:

$$\text{mmc}(a,b) = \frac{a \times b}{\text{mdc}(a,b)} \text{ e sabendo que: } \text{mdc}(a,b) = \begin{cases} a & \text{se } b = 0 \\ \text{mdc}(b, a \bmod b) & \text{se } b \neq 0 \end{cases}$$

40 O sub-factorial ($!n$) de um número inteiro não negativo é definido por:

$$!n = \begin{cases} 1 & \text{se } \ n = 0 \\ !(n-1).n + (-1)^n & \text{se } \ n > 0 \end{cases}$$

1. Escreva uma função recursiva para calcular $!n$.

2. Escreva a correspondente versão iterativa.

41 Implemente uma função recursiva que permita calcular a potência:

$$x^n = \begin{cases} 1 & \text{se} \quad n = 0 \\ x^{n-1} * x & \text{se} \quad n > 0 \\ x^{n+1}/x & \text{se} \quad n < 0 \end{cases}$$

para n inteiro e x real.

42 Seja Comissões(m, n) o número de diferentes comissões de n pessoas que se podem formar por eleição entre m pessoas. Por exemplo, Comissões$(4, 3) = 4$, porque dadas quatro pessoas, **A, B, C** e **D**, há quatro possíveis comissões de três membros, **ABC, ABD, ACD** e **BCD**. É sabido que:

- Comissões$(m, n) = 1$ quando $m = n$;
- Comissões$(m, n) = m$ quando $n = 1$;
- Comissões(m, n) =Comissões$(m - 1, n - 1)$+ Comissões$(m - 1, n)$ quando $m > n > 1$.

Escreva uma função recursiva que calcule Comissões (m, n) para $m \geq n \geq 1$.

43 Implemente uma função recursiva que permita escrever, por ordem inversa, os dígitos de um número inteiro.

44 Para uso dos alunos do Ensino Primário, pretende-se imprimir uma tabuada de multiplicar. Por exemplo, para $n = 5$, tal tabuada terá o seguinte formato:

```
1
2  4
3  6   9
4  8   12  16
5  10  15  20  25
```

Elabore um procedimento recursivo para imprimir uma tabuada de tamanho n, de cabeçalho:

```
    void function imprimirTabuada(int n);
```

45 Considere a seguinte função, definida em $\mathbb{N}$:

$$P(n) = \prod_{i=1}^{n} (2i - 1)$$

1. Elabore uma função recursiva para o seu cálculo.
2. Construa uma versão iterativa da mesma função.

E.9 Tipo de Dados Compostos: Tabelas (“Array”)

46 Preveja a saída do seguinte programa, de seguida introduza-o no computador para verificar se a saída é igual à que previu:

```
#include <stdio.h>

void lerNumero(int *num) {
  int lido;

  printf("Indique um numero: ");
  scanf("%d%",&lido);
  while (lido < 0) {
    printf("O numero nao pode ser negativo.\n");
    printf("Indique um numero: ");
    scanf("%d",&lido);
  }
  *num = lido;
}

int capicua(int num) {
  int indice, inicial, final;
  int a[10];

  indice = 1;
  while (num > 0) {
    a[indice] = num % 10;
    num = num / 10;
    indice = indice + 1;
  }
  inicial = 1;
  final = indice - 1;
  while ((a[inicial] == a[final]) && (inicial < final)) {
    inicial = inicial + 1;
    final = final - 1;
  }
  return(inicial >= final);
}

int main() {
  int numero;

  lerNumero(&numero);
  if (capicua(numero))
    printf("O numero %d é uma capicua\n",numero);
  else
    printf("O numero %d não é uma capicua\n",numero);
}
```

Nota: Todos os programas seguintes devem ser convenientemente modularizados através de funções.

47 Escreva um sub-programa que, dado um vector de n inteiros, calcule a média dos pares e a média dos impares.

48 Escreva um sub-programa que, dados um inteiro positivo k e um vector de n inteiros positivos, calcule a média dos múltiplos de k e a média dos submúltiplos de k, existentes no vector.

49 São dados os n elementos inteiros de um vector x (n constante e igual a 100) e ainda um valor inteiro k. Escreva um programa para imprimir todos os pares de números x_i, x_j, tais que $x_i + x_j = k$.

50 Escreva um programa para:

1. ler o inteiro n;
2. ler os $n \times n$ elementos inteiros de uma matriz A;
3. contar quantos elementos nulos existem acima da diagonal principal
4. se o número de elementos nulos for par, escrever a matriz, caso contrário escrever a sua transposta.

51 Faça sub-programas para:

1. Somar duas matrizes $A(n,m)$ e $B(n,m)$.
2. Multiplicar duas matrizes $A(n,m)$ e $B(m,k)$.
3. Dada uma matriz $A(n,m)$ de elementos reais, determinar a linha cuja soma dos seus elementos é máxima.
4. Calcular o determinante de uma matriz triangular $A(n,n)$ de elementos reais.

52 Escreva um programa que, dada uma matriz A de $m \times n$ elementos reais, determine quantos são superiores ao valor da média de todos os elementos da matriz.

53 Um treinador de atletismo treina 5 atletas e faz 12 sessões de treino por semana. Em cada sessão, cada atleta percorre uma distância que é cronometrada. Os valores dos tempos, em segundos, são registados sob a forma de uma matriz $T(5 \times 12)$, onde cada linha diz respeito a um atleta e cada coluna a uma sessão de treino. Supondo já feita a leitura da matriz, escreva uma secção de programa para:

1. calcular e escrever a média dos tempos realizados em cada sessão de treinos;
2. determinar e escrever o melhor tempo realizado por cada um dos atletas nas 12 sessões.

54 Escreva um programa que, dada uma matriz quadrada de ordem n, $(0 < n \leq 100)$ de elementos inteiros, e dados dois inteiros k e l, devolva a matriz após troca entre si das colunas k e l.

E.10 Estruturas não homogéneas: Registos ("struct")

55 Escreva sub-programas para operar com complexos, declarando *complexo* como uma estrutura do tipo `struct complexo`;

56 Como se sabe uma matriz quadrada de elementos complexos diz-se hermítica se for igual à sua associada, isto é se:

$$A = A^*$$

em que A^* é a matriz transposta da matriz conjugada de A.

1. Considere a seguinte definição:

   > ≪Dois números reais dizem-se iguais se a distância entre ambos for inferior a 10^{-30}≫

   Elabore uma função para verificar a igualdade entre dois números reais de cabeçalho:

   ```
   int iguais(real x, real y);
   ```

2. Defina o tipo complexo utilizando uma estrutura de `struct complexo`;

3. Considerando as declarações:

   ```
   complexo matriz[20][20];
   ```

   elabore uma função para verificar se uma dada matriz é ou não hermítica, utilizando o conceito de igualdade entre reais definido acima. A função terá como cabeçalho:

   ```
   int hermitica (complexo a[][20], int n);
   ```

   Por razões de eficiência o algoritmo deverá parar logo que encontre um par de elementos correspondentes que não sejam conjugados.

57 Operações com fracções.

1. Escreva sub-programas para operar com fracções (ler, escrever, simplificar, somar, multiplicar, dividir, subtrair, calcular potências de fracções), declarando as fracções como estruturas do tipo `struct` cujos campos são do tipo `integer`;

2. Elabore um programa para escrever os primeiros n termos de uma sucessão associada à *série harmónica*:

   $$H = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

   sob a forma de fracção. Por exemplo, para $n = 4$, a saída do programa deverá ser:

   ```
   1
   3/2
   11/6
   25/12
   ```

58 Sendo dados os seguintes tipos:

```
typedef struct peca {
   char  nome[20];
   int   disponivel;
   float precoUnitario;
} Peca;
```

```
typedef struct tipo{
  int  Codigo;
  peca Peca;
  peca Armazem[int];
} Tipo;
```

que descrevem uma armazém de peças. Escreva um sub-programa de facturação que, recebendo os códigos e quantidades das peças encomendadas escreva, em papel, uma factura cujas *linhas de detalhe*, organizadas por coluna, contêm:

1. Para cada peça *encomendada e disponível*, o código, nome, preço unitário, quantidade encomendada e preço total da quantidade encomendada;

2. Para cada peça *encomendada e não disponível*, o código, nome, preço unitário, quantidade encomendada e a mensagem "não disponível" na coluna correspondente ao preço total;

e cuja *linha de total*, posicionada após todas as *linhas de detalhe*, contêm, na coluna correspondente ao preço total, a soma dos preços totais das quantidades encomendadas de todas as peças disponíveis. Declare os tipos e variáveis de que necessitar para elaborar o sub-programa pedido. Indique como invocaria o dito sub-programa.

E.11 Ficheiros

59 São dados os seguintes tipos e declaração:

```
typedef struct planeta {
    char nome[8];
    int visivel;
    float raioOrbital;
} Planeta;

Planeta sistema[9];
```

que descrevem o Sistema Solar. Escreva um sub-programa que leia os seguintes dados de um ficheiro ≪`sistemaSolar.txt`≫.

Nome	Visibilidade	Raio Orbital
Mercúrio	s	0.39
Vénus	s	0.72
Terra	s	1.0
Marte	s	1.5
Júpiter	s	5.2
Saturno	s	9.5
Úrano	n	19.2
Neptuno	n	30.1
Plutão	n	39.5

e que escreva o nome e o raio orbital dos planetas visíveis, da Terra, a olho nu.

60 Pretende-se implementar um programa capaz de gerir uma lista de amigos.
Considerando o seguinte registo:

```
typedef struct amigo {
 char nome[120];
 char sexo;
 int idade;
 char telefone[9];
 char correioE[100];
} Amigo;

Amigo listaAmigos[1000];

FILE *ficheiroListaAmigos;
```

Construa sub-programas para:

- ler a lista de amigos de um ficheiro.
- Criar a lista de amigos (ficheiro).
- Actualizar a lista de amigos.
- Procurar, por nome, um amigo.

Construa um programa que permita, em ciclo, fazer várias operações na lista de amigos, até que o utilizador escolha terminar o programa.

61 Implemente a Cifra de Deslocamento Simples.

$$\begin{aligned} E_c(x) &= (x+c) \mod |\mathscr{A}| \\ D_c(y) &= (y-c) \mod |\mathscr{A}| \end{aligned}$$

- Alfabeto Português incompleto $\mathscr{A} = \{\text{a-z}\}$.
- Cifração carácter a carácter.
- Interface Entrada/Saída: ficheiros e linha de comando.
- Implementação (cifrar/decifrar) em C.

62 Implemente a Cifra de Deslocamento Linear.

$$\begin{aligned} E_{(a,b)}(x) &= (ax+b) \mod |\mathscr{A}| \\ D_{(a,b)}(y) &= a^{-1}(y-b) \mod |\mathscr{A}| \end{aligned}$$

- Alfabeto Português incompleto $\mathscr{A} = \{\text{a-z}\}$.
- Cifração carácter a carácter.
- Interface Entrada/Saída: ficheiros e linha de comando.
- Implementação (cifrar/decifrar) em C.

E.12 Ordenação e Pesquisa

Pesquisa

63 Implemente o método de ≪Pesquisa Exaustiva≫.

- Caso o vector não esteja ordenado o único algoritmo possível é dado pela procura exaustiva desde o início do vector até ao seu fim.

64 Implemente o método ≪Pesquisa Binária≫.

- Verificar se o elemento é igual à posição média do vector.
- Caso seja menor do que a posição média, repetir o processo na primeira metade do vector;
- Caso seja maior do que a posição média, repetir o processo na segunda metade do vector.

O algoritmo pára assim que encontrar o elemento, ou quando atinge um vector de dimensão nula.

Ordenação

65 Implemente o método de ordenação *Borbulagem* (≪Bubble Sort≫).

- Percorra o vector trocando pares de elementos que estejam fora de ordem.
- Repita o processo até que, numa das passagens anteriores, não se proceda a nenhuma troca.

66 Implemente o método de ordenação *Selecção Linear* (≪Linear Sort≫).

- Encontre o maior dos n elementos;
- Coloque esse elemento na sua posição final trocando-o com o enésimo elemento;
- Recursivamente ordene $n - 1$ primeiras posições do vector.

67 Implemente o método de ordenação *Inserção Ordenada* (≪Insertion Sort≫).

- Consideramos o vector como a concatenação de duas sequências: uma ordenada, e a segunda não ordenada. No início a primeira sequência contém apenas um elemento.
- O primeiro elemento da sequência não ordenada é inserido na posição correcta da sequência ordenada (movendo os elementos maiores uma posição para a esquerda).

- Esta operação aumentou a sequência ordenada de um elemento e retira esse elemento à sequência não ordenada.
- Repetimos até todos os elementos estarem na sequência ordenada.

68 Implemente o método de ordenação *Fusão* (≪Merge Sort≫).

Considerem-se dois vectores ordenados. A sua fusão num só vector ordenado é possível seguindo os seguintes passos:

- Comparam-se os dois elementos iniciais dos dois vectores.
- Escolhe-se o vector que contêm o elemento menor e copiam-se os seus elementos para o vector final até que o elemento que se está a considerar já seja maior do que o elemento inicial do outro vector.
- troca-se de vector e repete-se o processo, até que se chega ao fim de um dos vectores.
- copiam-se os restantes elementos do outro vector para o vector final.

O processo precisa sempre de um vector auxiliar, sobre o qual se vai construir o vector ordenado.

Pode-se aplicar este método para ordenar um vector genérico. Basta cnosiderar a divisão recursiva em metades de um dado vector, com o caso de base dos vectores singulares, e tendo em atenção que um vector singular está ordenado.

69 Implemente o método de ordenação (≪Quick Sort≫).

Dado um vector de n inteiros e um inteiro l existente no vector pretende-se particionar esse vector em duas partes: os elementos inferiores a l e os elementos superiores a l.

- Começando do lado esquerdo do vector procurar um elemento que seja maior ou igual a l.
- Começando do lado direito do vector procurar um elemento que seja menor ou igual a l.
- Trocar estes dois elementos.
- Continuar até as duas procuras se cruzarem.

Referências

ADAMS, JEANNE C., BRAINERD, WALTER S., MARTIN, JEANNE T., SMITH, BRIAN T., & WAGENER, JERROLD L. 1997. *FORTRAN 95 Handbook.* The MIT Press.

AHO, ALFRED, HOPCROFT, JOHN, & ULLMAN, JEFFREY. 1983. *Data Structures and Algorithms.* Addison-Wesley Publishing Company.

ALAGIČ, SUAD, & ARBIB, MICHAEL A. 1978. *The Design of Well-Structured and Correct Programs.* New York: Springer-Verlag.

DARCANO. 2007. *Tutorial: Aprenda a criar seu próprio Makefile.* Forum Ubuntu Linux - PT. `http://ubuntuforum-pt.org/index.php/topic,21155.0.html`.

DE SÁ, JOAQUIM P. MARQUES. 2004. *Fundamentos de Programação Usando C.* Tecnologias de Informação. FCA, Editora de Informática LDA. 68N/SA.

GOTTFIRED, BYRON. 1994. *Programação em Pascal.* 2nd edn. Lisboa: McGraw-Hill.

KERNIGHAN, BRIAN, & RITCHIE, DENNIS. 1988. *The C Programming Language.* 2nd edn. Prentice Hall.

KNUTH, DONALD E. 1973. *The Art of Computer Programming.* 2nd edn. Vol. 1, Fundamental Algorithms. Reading, USA: Addisson-Wesley Publishing Company.

KNUTH, DONALD E. 1981. *The Art of Computer Programming.* 2nd edn. Vol. 2, Seminumerical algorithms. Reading, USA: Addisson-Wesley Publishing Company.

KNUTH, DONALD E. 1984. Literate Programming. *The Computer Journal,* **27 (2)**, 97–111.

MAIN, MICHAEL, & SAVITCH, WALTER. 1997. *Data Structures and other Objects Using C++.* Addison-Wesley. DMAT 68P/MAI.

MECKLENBURG, ROBERT. 2004. *Managing Projects with GNU Make.* 3rd edn. O'Reilly Media.

MENDELSON, ELLIOTT. 1968. *Introduction to Mathematical Logic.* Princeton: D. Van Nostrand Company, Inc.

PATTIS, RICHARD E. 1980. *EBNF: A Notation to Describe Syntax.* "While developing a manuscript for a textbook on the Ada programming language in the late 1980s, I wrote a chapter on EBNF".

SEMGUPTA, SAUMYENDRA, & KOROBKIN, CARL PHILIP. 1994. *C++, Object-Oriented Data Structures*. New-York: Springer-Verlag. DMAT 68N/SEN.

STROUSTRUP, BJARNE. 1997. *The C++ Programming Language*. Addison Wesley Longman, Inc.

WEISS, MARK ALLEN. 1997. *Data Structures and Algorithm Analysis in C*. Menlo Park, California: Addison-Wesley.

WELSH, JIM, & ELDER, JOHN. 1979. *Introduction to Pascal*. 2nd edition edn. Computer Science. London: Prentice-Hall International, Inc.

WIRTH, NIKLAUS. 1976. *Algorithms + Data Structures = Programs*. Automatic Computation. Englewood Clifs, New Jersey: Prentice-Hall, Inc.

WIRTH, NIKLAUS. 1986. *Algorithms & Data Structures*. Englewood Cliffs, New Jersey, USA: Prentice-Hall.

WIRTH, NIKLAUS. 1989. *Algoritmos e Estruturas de Dados*. Rio de Janeiro: Prentice-Hall do Brasil.