Aprenda a Programar

Uma breve introdução ilustrada com a linguagem C

pplware
no comments

Henrique Dias
Revisto por Luís Soares

Aprenda a Programar
Uma Breve Introdução

Henrique Dias

Autor: Henrique Dias, henriquedias.com
Revisto por: Luís Soares, luissoares.com

Versão 2.1.1
Publicado a 6 de março de 2015
Com o apoio de: Pplware, pplware.com

Índice

Agradecimentos

Em primeiro lugar, quero agradecer aos meus pais por estarem sempre presentem e me terem apoiado imenso durante toda a minha vida.

Em segundo lugar, quero agradecer a toda a equipa do Pplware, pelo apoio dado. Agradeço também à comunidade do Pplware, que me ajudou, deu dicas, fez correções, sempre construtivamente.

Finalmente, mas não menos importante, quero agradecer ao Luís Soares, por se ter oferecido a rever este livro e me dado imensas dicas para o aprimorar.

Capítulo 0

Introdução

Este livro pretende dar-lhe a conhecer os fundamentos básicos e essenciais do mundo da programação. Não apenas ensinar-lhe a linguagem de programação x ou y, mas sim ensinar-lhe a **programar**.

Programar não é conhecer uma linguagem de programação. É conhecer os fundamentos, a lógica, que está na base de todas as linguagens de programação. Com isso, será capaz de utilizar **qualquer** linguagem de programação.

0.1 Qual o público-alvo deste livro

O público-alvo deste livro prende-se aos que querem iniciar a sua jornada no mundo da programação: independentemente de estarem a estudar ou serem apenas curiosos por esta área. Neste livro é utilizada a linguagem de programação C de forma a mostrar os fundamentos básicos da programação. Fundamentos estes que são omnipresentes em todo o mundo da programação.

O capítulo 7, sobre apontadores, poderá, eventualmente, ser do interesse de alguém que já conhece algumas linguagens de programação, que já tem alguma habilidade, mas que está com dificuldade nessa parte.

0.2 Como está estruturado o livro

Este livro foi baseado nos artigos da saga Introdução à Programação do Pplware[1]. Não é apenas a compilação de todos os artigos, mas também a sua evolução. Tudo foi remodelado, revisto e melhorado.

A estrutura deste livro foi feita de forma a que a aprendizagem seja gradual e lhe dê uma visão sobre aquilo que irá encontrar em (quase) **todas** as linguagens de programação existentes. Ora veja como está estruturado.

- **Capítulo 1: "Fundamentos"** - No primeiro capítulo serão abordados fundamentos essenciais sobre o mundo da programação: o que é uma linguagem de programação, para que serve, como está estruturada, o que precisará utilizar, etc;
- **Capítulo 2: "Dados"** - Tal como o nome sugere, o tema a abordar são os dados e várias coisas relacionadas com eles. São abordados os vários tipos de dados existentes, tal como a utilização de variáveis e constantes em C;
- **Capítulo 3: "Operadores"** - No mundo da programação os dados devem ser modelados, moldados, alterados. É com os operadores abordados neste capítulo que tudo isso pode ser feito. Os operadores abordados existem na maioria das linguagens de programação;
- **Capítulo 4: "Controlo de Fluxo"** - A ordem lógica das ações a tomar pode ser diferente em diversas situações. Neste capítulo aprenderá a alterar e a modificar a ordem das ações com base em condições;
- **Capítulo 5: "Funções e Procedimentos"** - No quinto capítulo serão abordadas duas

[1] Consultar pplware.com

coisas importantíssimas na programação: as funções e procedimentos. Com elas irá poupar espaço e ainda reutilizar código;
- **Capítulo 6: "Arrays"** - Neste capítulo é abordada uma estrutura de dados extremamente importante, os *arrays*, que estão presentes em quase todas as linguagens de programação;
- **Capítulo 7: "Strings"** - No sétimo capítulo serão abordadas as *strings*, ou seja, sequências de caracteres. São muito importantes, principalmente quando precisa de armazenar texto;
- **Capítulo 8: "Apontadores"** - Neste capítulo, sobre apontadores, irá abordar algo omnipresente, mas que é utilizado massivamente em apenas algumas linguagens, como C. Pode ser um capítulo interessante para quem já tem bases nalguma linguagem de programação, mas que gostava de aprender mais sobre apontadores;
- **Capítulo 9: "Anexos"** - No último capítulo encontram-se os anexos que são mencionados ao longo do livro.

0.3 Convenções

Ao longo do livro são apresentados endereços de *websites* para que os possa visitar. Todos estão dotados de hiperligação de forma a que o seu acesso seja fácil. Além disso, as partes "http://" e "www." foram omitidas de todos os endereços, tal como a barra que por vezes aparece no final. O endereço `http://www.wikipedia.org/` será mostrado como `wikipedia.org`.

A linguagem de programação C será a utilizada para ilustrar os fundamentos sobre a programação. Por vezes um conjunto de pessoas reúne-se em redor desta linguagem e lança uma revisão da mesma, ou seja, novas formas de fazer algo, melhorias. Atualmente são utilizadas, maioritariamente, três revisões: a C89, C99 e a C11. Tendo em conta que a C11 é a mais recente e a que permite ter um código mais elegante e legível, será aquela que vai estar na base de todos os exemplos ao longo do livro.

0.4 Sobre o revisor

O Luís Soares[2] estudou Licenciatura e Mestrado em Engenharia Informática e de Computadores no Instituto Superior Técnico. Trabalha, há vários anos, no âmbito de aplicações *web* com as mais diversas tecnologias. Atualmente trabalha na Media Capital numa posição de Desenvolvedor e Arquiteto de *Software*.

0.5 Em caso de dúvidas

Caso tenha alguma dúvida, poderá utilizar algum dos artigos da saga Introdução à Programação no Pplware ou enviar-me um mail para hacdias@gmail.com.

[2]Consultar: luissoares.com

Capítulo 1

Fundamentos

O objetivo deste primeiro capítulo é mostrar-lhe os fundamentos essenciais sobre o mundo da programação. Começaremos por utilizar pseudocódigo (conceito explicado à frente), avançando gradualmente até chegar à linguagem pretendida: a Linguagem C.

1.1 Linguagens de programação

À semelhança de uma linguagem humana, uma linguagem de programação permite-nos comunicar, não com Humanos, mas com computadores ou qualquer outro sistema computorizado. Linguagens de programação são constituídas por uma "gramática" que contém todas as regras sintáticas utilizadas. Através da utilização dessas regras, podemos comunicar instruções a um computador, ou seja, instruir-lhe a fazer algo. É assim que os programas são criados.

> **Definição 1. Regras sintáticas** consistem num conjunto de normas a seguir que indicam como se deve estruturar o código, ou seja, como se deve construir o código.

Existem inúmeras linguagens de programação: algumas de propósito geral, ou seja, sem ter uma finalidade específica. Por outro lado, existem outras criadas para um domínio específico. A linguagem de programação C é um exemplo de uma linguagem de programação de propósito geral. Por outro lado a *Wolfram Language*[1], é uma linguagem de domínio específico, multi-paradigma (abordado mais à frente) e dedicada à Matemática.

1.2 Algoritmos

É importante compreender alguns conceitos básicos que serão fundamentais na sua jornada no mundo da programação. Assim, vai começar por entender o que são **algoritmos** visto que vai estar sempre em contacto com eles.

> **Definição 2. Algoritmos** são quaisquer sequências de instruções finitas e bem definidas que podem ser executadas por computadores, autómatos ou até mesmo humanos.

Na Figura 1.1 pode visualizar todo o processo da confeção de um bolo onde o algoritmo é a receita – uma sequência de instruções bem definida e finita – que é executada pelo(a) "cozinheiro(a)".

Os algoritmos podem ser representados de diversas formas. Aqui são abordadas duas delas: os fluxogramas e o pseudocódigo. Estas representações dos algoritmos são essenciais antes destes serem escritos em código; ir-se-à poupar tempo visto que são reduzidos os possíveis erros durante o desenvolvimento.

Nem sempre são precisos os dois tipos de representação de algoritmos. Por vezes basta um. Isso é algo que depende do fluxo e métodos de trabalho de cada pessoa. Os fluxogramas e pseudocódigo são algo universal, que não se compromete com uma determinada linguagem de programação.

[1] Consultar wolfram.com/language

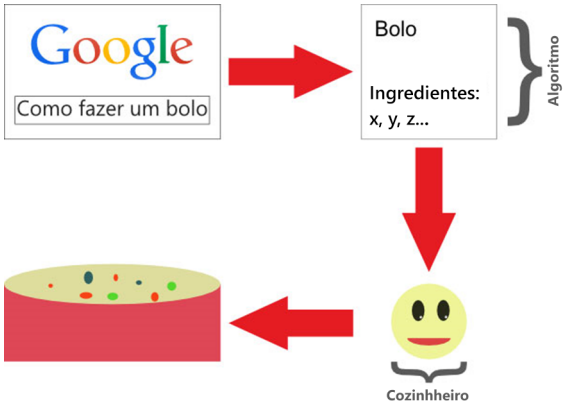

Figura 1.1: Confeção de um bolo

1.2.1 Fluxogramas

Vejamos então a primeira forma de representar algoritmos, os fluxogramas.

> **Definição 3.** Um **fluxograma** é uma representação gráfica de um algoritmo que utiliza símbolos de forma a demonstrar os processos neste realizado.

Existem várias vantagens na criação de fluxogramas como, por exemplo, a sua facilidade de criar, a facilidade na partilha e ajuda a criar modelos mentais.

Um fluxograma pode fazer uso de muitos símbolos. No entanto, apenas iremos necessitar dos básicos para ter uma boa compreensão de como funcionam os fluxogramas. Pode visualizar estes símbolos na Figura 1.2.

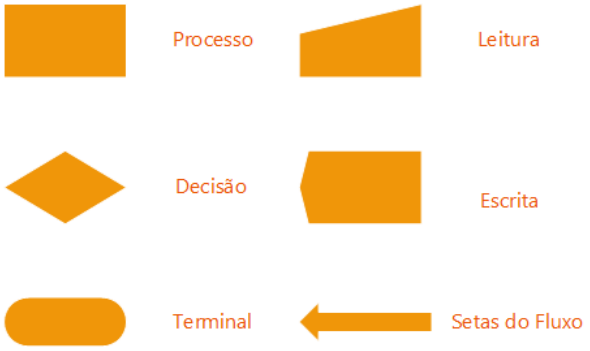

Figura 1.2: Símbolos primários dos fluxogramas

Na Figura 1.3 pode visualizar um fluxograma, baseado no processo de confeção de um bolo.

1.2.2 Pseudocódigo

Uma forma mais aproximada do código final é a utilização de pseudocódigo. Sendo assim, sucede a criação dos fluxogramas.

> **Definição 4. Pseudocódigo** é uma forma de representação de algoritmos que se assemelha a linguagens de programação mas que utiliza a língua nativa do utilizador de forma a ser facilmente entendida por quem não tem quaisquer conhecimentos da sintaxe de uma linguagem de programação.

Os programadores cuja língua nativa é português, costumam referir-se ao pseudocódigo como *Portugol*, também conhecido por "Português Estruturado". O seguinte trecho de Portugol representa o algoritmo anteriormente representado com um fluxograma, a confeção de uma receita:

```
1  inicio
2     variavel caracter receita ← "A minha receita"
3
4    se tenhoIngredientes(receita) == verdadeiro entao
5        fazerBolo()
6    senao
7        comprarIngredientes()
8        fazerBolo()
9   fimse
10 fim
```

Como pode verificar, o algoritmo acima, em Portugol, é extremamente simples de compreender. De momento, peço-lhe que entenda as expressões que terminam com `()` como se fossem comandos.

> **Definição 5.** Um **comando** é uma ordem ou instrução dada a um computador ou qualquer outra máquina automatizada.

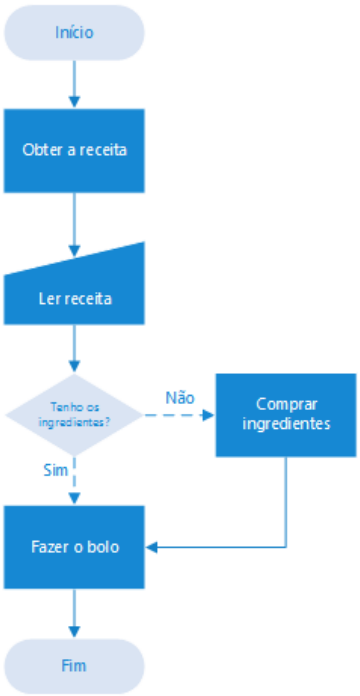

Figura 1.3: Fluxograma "Confeção de um bolo"

1.3 Constantes e variáveis

Na quarta linha do trecho de código que visualizou em **1.2.2**, pôde encontrar o seguinte código:

```
1 variavel caracter receita ← "A minha receita"
```

Esse trecho de código declara uma variável do tipo **caracter**, que se chama `receita`, com o valor "A minha receita".

> **Definição 6.** A **declaração de variáveis**, ou constantes, consiste no processo em que o compilador é "avisado"da sua existência para que, de cada vez se menciona o nome da variável/constante, esta seja utilizada.

Por exemplo, quando declaramos a variável `receita`, estamos a reservar um endereço da memória RAM para alocar um valor. Neste caso, o valor foi atribuído no momento da sua declaração utilizando o operador "←" (sem aspas).

A partir do momento declaramos uma variável ou constante, o endereço da memória RAM que foi reservado estará disponível através do seu nome. Sempre que for referenciado `receita` no código, o valor da variável será retornado.

1.3.1 Constantes

Comecemos por abordar as constantes.

> **Definição 7. Constantes** permitem armazenar valores imutáveis, ou seja, que não podem ser alterados ao longo da execução de um programa.

Ora veja o seguinte exemplo:

```
1 constante caracter EMPRESA ← "Pplware"
2 EMPRESA ← "A Minha Empresa"
```

Na primeira linha, a constante `EMPRESA`, do tipo carácter, é declarada e é-lhe atribuído o valor `Pplware`. De seguida, na segunda linha, existe uma tentativa de alterar o seu valor da constante. Esta tentativa irá falhar, causando um erro visto que o valor de uma constante não pode ser alterado após a sua atribuição.

> **Nomenclatura**
>
> Convencionalmente, o nome das constantes é escrito com letras maiúsculas para haver uma melhor distinção no código-fonte. Esta convenção não é obrigatória e não irá causar quaisquer erros durante a execução de um programa. Seguir esta convenção apenas torna mais clara a distinção entre variáveis e constantes dentro do código-fonte facilitando tanto a si, como programador, como a outros programadores que vejam o código do seu programa.

1.3.2 Variáveis

Por outro lado, existem as variáveis.

> **Definição 8. Variáveis**, ao contrário das constantes, permitem o armazenamento de valores que podem ser alterados durante a execução de um programa. Geralmente são utilizadas para manter estados de algo e são fundamentais na programação.

Abaixo encontra um exemplo:

```
1 variavel caracter temaDaSecao ← "Constantes"
2 temaDaSecao ← "Variáveis"
```

Na primeira linha é declarada uma variável do tipo carácter, com nome `temaDaSecao` e valor `Constantes`. Seguidamente, o seu valor é alterado para `Variáveis` não causando nenhum erro, pois o valor das variáveis pode ser alterado ao longo da execução de um programa.

Regras de nomeação

Existem diversas regras que têm de ser seguidas no momento da declaração de uma variável para que não seja causado nenhum erro.

O nome das variáveis e constantes:

- Não pode começar com números (ex.: `9comida` não é permitido, `comida9` é válido);
- Não pode ser igual a uma palavra reservada (ex.: `if` não é permitido, mas `maria` é permitido);
- Não pode conter espaços (ex.: `a minha var` não é permitido, porém `aMinhaVar` é válido);
- Não pode conter caracteres especiais (existem exceções em diversas linguagens).

> **Definição 9. Palavras reservadas** são aquelas que constam na gramática da linguagem de programação. Tendo em conta que a palavra "se" é um comando do Portugol, não podemos declarar

nenhuma variável ou constante com esse nome. Se isso for feito, será gerado um erro.

As variáveis e constantes podem ser de diversos tipos. Os tipos de dados existentes variam de linguagem para linguagem. Mais à frente irão ser abordados os tipos de dados na linguagem C.

Nomenclatura

As variáveis, ao contrário das constantes, não são totalmente escritas em letras maiúsculas. Convencionalmente, a variação *lowerCamelCase* (pertencente ao padrão *CamelCase*) é seguida na maioria das linguagens de programação. Este padrão será utilizado visto que a maioria das linguagens de programação adotam-no, tal como a linguagem C.

1.4 Paradigmas de programação

Todas as linguagens de programação têm características que as distinguem de outras. O(s) paradigma(s) de programação que ela segue são fundamentais.

Definição 10. Paradigmas de programação são modelos ou padrões da forma de estruturar o nosso código. Existem muitos.

Nesta secção apenas são abordados 6 paradigmas de programação, os mais conhecidos e utilizados. As que adotam mais do que um paradigma chamam-se **multi-paradigma**.

1.4.1 Paradigma imperativo

O primeiro paradigma abordado é o paradigma imperativo. Este concentra-se num **estado** (que são as variáveis) e em **ações** (comandos) que **modelam** (alteram) esse estado.

Este paradigma pode ser comparado ao modo imperativo da linguagem humana visto que é criado para ordenar a realização de ações (como por exemplo, fazer algo, recortar, analisar, apagar...).

Alguns **exemplos** de linguagens de programação imperativas são, por exemplo: C, Java, C#, Pascal.

1.4.2 Paradigma procedimental

Com o paradigma procedimental, trechos de código podem ser reutilizados sem a necessidade de o copiar para diversos locais através da utilização de funções e procedimentos (que serão abordados mais a frente).

A maioria das linguagens de programação adotam este paradigma.

1.4.3 Paradigma estruturado

Nas linguagens de programação em que o paradigma estruturado é seguido, o código-fonte de uma aplicação pode ser reduzido em apenas três estruturas: sequência, decisão e iteração (repetição).

Sequência

Nesta primeira estrutura, as tarefas são executadas de forma linear, ou seja, uma após a outra. Abaixo encontra um pequeno exemplo.

```
1 Acordar;
2 Vestir;
3 Tomar o pequeno-almoco;
4 Ir trabalhar;
```

Este é um exemplo de uma sequência onde são realizadas as ações normais do dia-a-dia de um indivíduo que está empregado.

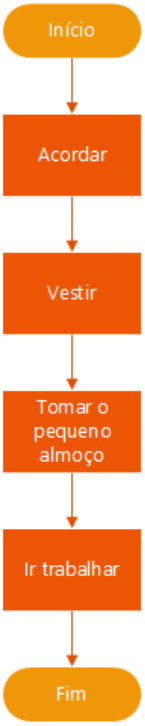

Figura 1.4: Fluxograma de uma sequência

Na maioria das linguagens de programação, os comandos/ações terminam com ponto e vírgula pois estas permitem que os comandos sejam colocados em linha da seguinte forma:

```
Acordar; Vestir; Tomar o pequeno-almoco; Ir trabalhar;
```

A utilização do ponto e vírgula é normalmente obrigatória quando existe mais do que uma instrução numa só linha. Existem linguagens de programação que só obrigam a utilização de ponto e vírgula nestes casos (JavaScript, por exemplo). Porém, existem outras, como C, Java e C# que obrigam a utilização de ponto e vírgula no final de **todas** as instruções.

Decisão

Neste tipo de estrutura, existe um trecho de código que é executado ou não dependendo do resultado de um teste lógico, também conhecido por predicado. Abaixo encontra diversos exemplos para esta estrutura.

O exemplo abaixo descreve a condição/decisão "Se acordar, vou trabalhar. Caso contrário, não vou trabalhar" (em pseudocódigo).

```
if "Acordar" then
  "Trabalhar"
else
  "Não trabalhar"
endif
```

Novos Termos:

- **if** → se
- **then** → então
- **else** → caso contrário
- **endif** → fim do se

O pseudocódigo acima já não está em português; já não é Portugol. O que encontra acima já se assemelha ao que irá visualizar nas linguagens de programação.

Retornando novamente ao trecho de código escrito acima, repare que `Trabalhar` só será executado se e apenas se o indivíduo `Acordar`. Caso contrário, o trecho `Não trabalhar` será executado.

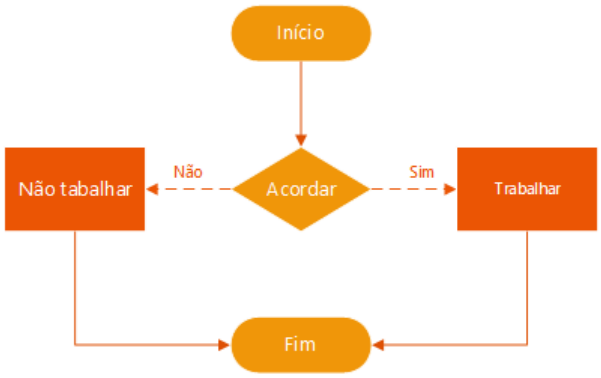

Figura 1.5: Exemplo de uma decisão em fluxograma

Agora veja o seguinte exemplo em que a condição "Dói-me a cabeça. Se doer muito pouco, vou trabalhar. Se doer pouco, tomo um comprimido e vou trabalhar. Se doer muito, vou ao médico e falto ao trabalho" é executada.

```
case "Dor de cabeca"
  when "muito pouco" then "trabalhar"
  when "pouco" then "tomar comprimido"; "trabalhar"
  when "muito" then "ir ao médico"; "não trabalhar"
```

Novos Termos:

- **case** → caso
- **when** → quando
- **else if** → caso contrário se

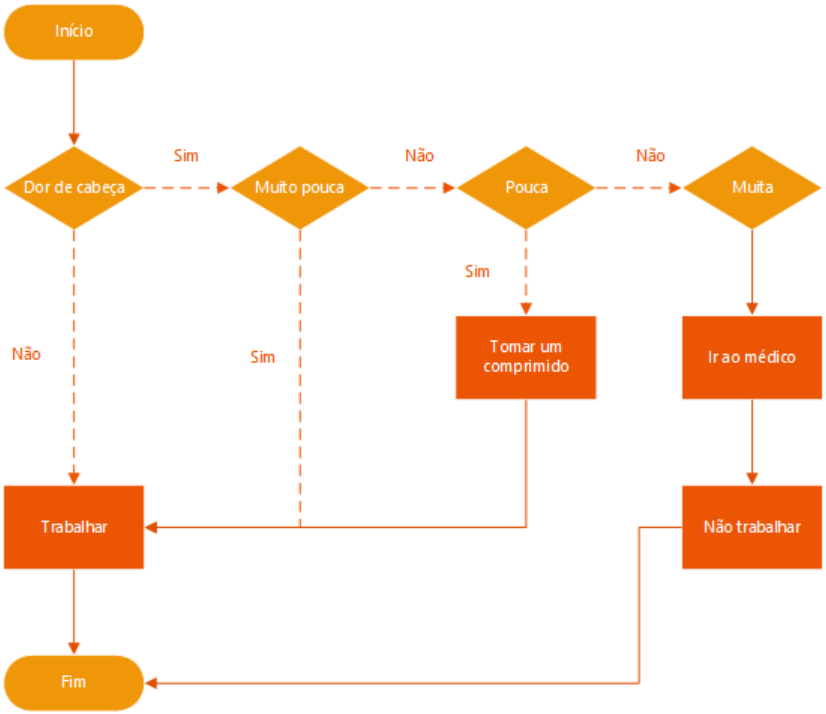

Figura 1.6: O que farei se me doer a cabeça

Este é mais um exemplo mas utilizando diferentes comandos. Este trecho poderia ser também escrito através de *primitivas if/else* da seguinte forma (ver Figura 1.6):

```
if "Dor de cabeca"
  if "muito pouco" then
    "trabalhar";
  else if "pouco" then
    "tomar comprimido";
    "trabalhar";
   else if "muito" then
    "ir ao médico";
    "não trabalhar";
   endif
endif
```

Iteração

Neste tipo de estrutura, também conhecido como repetição, um trecho de código é repetido um número finito de vezes dependendo do resultado de um teste lógico.

Abaixo encontra a repetição "não saio de casa enquanto não estiver vestido" em pseudocódigo:

```
do {
  "não sair de casa";
} while ( "não estou vestido" )
```

Novo termo:

- **do** → fazer

Ou seja, o código acima pode ser lido da seguinte forma: **fazer** "não sair de casa" **enquanto** "não estou vestido". De forma generalizada, fazer x enquanto y.

Agora veja o retrato da repetição "enquanto durmo, não me visto" em pseudocódigo:

```
while ( durmo )
 naoMeVisto();
```

Ou seja, **enquanto** acontece algo, faço outra coisa.

Visualize agora o código correspondente à ação "lavar os dentes 20 vezes".

```
for ( i = 0; i++; i < 20 )
  lavarOsDentes();
```

Ou seja, enquanto não acontece qualquer coisa, faço qualquer coisa.

Mais um exemplo, mas referente à expressão "Para cada dente, lavo-o muito bem".

```
for each dente in boca
 lavarMuitoBem();
```

Novos Termos:

- **each** → cada
- **in** → em

Ou seja, para cada item do conjunto, fazer qualquer coisa.

1.4.4 Paradigma declarativo

O Paradigma Declarativo contrasta com o Imperativo pois é capaz de expressar a lógica sem descrever como o fluxo de comandos funciona, ou seja, apenas diz ao computador **o que** fazer e não **como** fazer.

Um excelente **exemplo** de uma linguagem que utiliza este paradigma é Prolog, muito utilizado na área de inteligência artificial.

1.4.5 Paradigma funcional

O Paradigma Funcional engloba todas as linguagens de programação que utilizam funções matemáticas e evita estados. Estas linguagens de programação são muito utilizadas no campo da matemática.

Algumas linguagens que seguem o paradigma funcional são, por **exemplo**, Matlab, Wolfram Language, B.

1.4.6 Paradigma orientado a objetos

A Programação Orientada a Objetos permite a criação de **objetos** com base em **classes**. Estes objetos são instâncias dessas classes e possuem todos os atributos e funções presentes nas classes em questão.

Este paradigma é muito extenso e tem muita informação que mais à frente irá ser abordada. Atualmente existem muitas linguagens que utilizam este paradigma (Java, C++, C#, PHP, por exemplo).

Os paradigmas de programação não se limitam aos 6 (seis) apresentados pois existem inúmeros outros. Estes são apenas os paradigmas de programação mais abrangentes. Existem paradigmas baseados noutros paradigmas, outros que contrastam com outros, etc.

1.5 A linguagem C

Os anos de 1969 a 1973 foram de extremo entusiasmo dentro da AT&T Bell Labs porque foi quando a linguagem de programação C **foi maioritariamente desenvolvida**.

O principal desenvolvedor desta linguagem foi **Dennis Ritchie** que descreveu o ano de 1972 como o mais produtivo e criativo.

Figura 1.7: Dennis Ritchie

A linguagem desenvolvida por Ritchie chama-se **C** porque esta linguagem baseou-se imenso numa outra linguagem de programação chamada **B**, tendo C diversas características em comum com B.

Inicialmente, esta linguagem de programação, C, tinha como principal finalidade o desenvolvimento do Unix, que já havia sido escrito em **Assembly** - uma outra linguagem de programação.

A versão mais recente de C é C11 e foi lançada a dezembro de 2011. Esta linguagem foi uma das influências de muitas das linguagens de programação que atualmente são muito utilizadas. Entre muitas outras, C influenciou AWK, BitC, C++, C#, C Shell, D, Euphoria, Go, Java, JavaScript, Limbo, Logic Basic, Objective-C, Perl e PHP. **Isto não significa que estas linguagens não tenham sido influenciadas por outras.**

Nós iremos começar por abordar C porque é uma linguagem "mãe", que influenciou muitas outras. Por ser uma linguagem de baixo nível, pode ter um contacto mais próximo com o *hardware*.

1.5.1 Características da linguagem C

Como abordámos anteriormente, os paradigmas de programação são muito importantes e influenciam a forma como devemos escrever.

C é uma linguagem de programação, em relação aos paradigmas, **estruturada**, **imperativa** e **procedimental**. Outras características desta linguagem são o facto de ser padronizada pela ISO e de propósito geral.

Linguagem de programação compilada

Linguagens de Programação Compiladas são aquelas que passam pelo processo de compilação, ou seja, onde o seu código fonte é diretamente transformado na linguagem da máquina por via de um compilador.

> **Definição 11. Código fonte** é um conjunto de instruções lógicas, escritas de acordo com uma linguagem de programação existente.

Figura 1.8: Linguagem C

Aprendendo a linguagem C, fica preparado para se iniciar com muitas outras linguagens de programação pois tem uma sintaxe muito utilizada e, além disso, sabe a lógica.

1.5.2 Ambiente de desenvolvimento

Para começar a desenvolver programas, necessita ter um ambiente de desenvolvimento, preparado com as diversas ferramentas necessárias.

Instalação de um compilador

> **Definição 12.** Um **compilador** é a ferramenta que transforma o código-fonte na linguagem da máquina através do processo de compilação.

O compilador que iremos utilizar denomina-se GCC (*GNU Compiler Collection*). Este compilador é de fácil instalação e utilização.

Debian e derivadas

A instalação deste compilador na distribuição Linux Debian ou derivadas, como por exemplo, Ubuntu, é bastante simples. No terminal, execute os seguintes comandos:

```
> sudo apt-get update && apt-get upgrade
> sudo apt-get install build-essential
```

Depois desses dois comandos deverá ter instalado o compilador GCC, entre outras ferramentas. Para verificar se o GCC ficou instalado corretamente execute, o seguinte comando:

```
> gcc -v
```

Este comando deverá retornar a versão atualmente instalada do GCC.

Outras distribuições Linux

Para outras distribuições Linux é recomendável seguir as instruções encontradas na página oficial do projeto GCC[2].

Windows

No sistema operativo da Microsoft, o GCC pode ser instalado recorrendo a projetos como o MinGW ou o Cygwin. Recomendo o primeiro[3].

OS X

No sistema operativo da Apple, o GCC costuma vir com o Xcode, um IDE multi-linguagem criado pela Apple, podendo tudo isto ser instalado através do terminal com o seguinte comando:

```
> xcode-select --install
```

Pode verificar a versão instalada do GCC nos dois últimos sistemas operativos recorrendo ao comando anteriormente mencionado.

Um **IDE** é um Ambiente de Desenvolvimento Integrado, do inglês *Integrated Development Environment*. É um programa de computador que reúne diversas ferramentas para apoiar no desenvolvimento de software.

Editor de texto

Além do compilador, irá também precisar de um editor de texto. Qualquer editor de texto funciona, mas recomendo um que suporte a sintaxe da linguagem C.

Existem vários editores de texto que pode utilizar. Aqui deixamos algumas recomendações:

- **Windows** → Notepad++, Atom, Sublime Text;
- **Linux** → Gedit, Atom, Sublime Text (algumas distribuições), Vim;
- **OS X** → TextWrangler, Sublime Text, Atom.

1.5.3 "Hello World!"

Como seria o mundo da programação sem o famoso "Hello World"? É uma tradição o primeiro programa criado por alguém imprimir a mensagem "Hello World" no ecrã.

Crie um ficheiro, onde queira, com o nome HelloWorld.c. Tenha em atenção à extensão do ficheiro que tem que ser .c, ou seja, da linguagem C.

Abra esse mesmo ficheiro com um editor de texto, copie e cole o seguinte texto e então guarde

[2] Ver gcc.gnu.org
[3] *Download* em mingw.org/download

as alterações.

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

int main() {
  printf("Hello World!\n");
  return 0;
}
```

Este trecho de código irá imprimir no ecrã a mensagem "Hello World!". Para executarmos este comando, deverá abrir a Linha de Comandos/Terminal e navegar até ao local onde guardou o ficheiro. Depois, execute o seguinte comando:

```
> gcc HelloWorld.c -o HelloWorld
```

Onde HelloWorld.c é o ficheiro de entrada e HelloWorld o ficheiro de saída. A extensão do ficheiro produzido deverá ser diferente consoante o sistema operativo.

No meu caso, como estou a utilizar o Windows, foi criado um ficheiro **.exe**. Agora, para executar o seu ficheiro, basta o executar através da linha de comandos:

```
> HelloWorld
```

Deverá receber uma mensagem na Linha de Comandos a dizer "Hello World".

1.5.4 `#include`

A primeira linha do trecho acima **não** é C mas sim uma indicação para o **compilador**.

> C é uma linguagem utilizada em locais que necessitam de alta velocidade, como o *kernel* - núcleo - do Linux, pois esta tem essa capacidade.
>
> Devido à alta velocidade que pode ser proporcionada por esta linguagem, C não está preparado, por omissão, para todos os tipos de tarefas. Assim, precisamos de incluí-las para ter disponíveis mais funções.

A linguagem C não vem "empacotada" de funções por padrão e é necessário incluí-las. Para as incluir, utilizamos a diretiva `#include` que diz ao compilador que precisa de incluir ficheiros ponto H (.h) que são ficheiros do tipo *header*. Nesta caso adicionámos o ficheiro `stdio.h` que quer dizer *standard input/output*, ou seja, sistema padrão de entrada e saída.

1.5.5 Função `main`

Todos os programas C têm que ter, obrigatoriamente, uma função `main` que será automaticamente executada. Esta função é o ponto de partida para criar um programa. É o seu cerne.

Voltaremos a falar sobre este tema e da sua importância.

1.5.6 Função `printf`

Este é um comando/função que está contido no ficheiro stdio.h. Caso não incluamos o ficheiro, será gerado erro. Esta função significa *print formatted*, ou seja, "escrita de dados formatados".

Esta função aceita vários parâmetros; permite-nos enviar várias coisas que irão ser processadas por ela. De momento, apenas iremos falar do primeiro argumento.

O primeiro argumento é uma *string*, ou seja, um conjunto de caracteres. Este conjunto de caracteres deve ser colocado dentro de aspas.

Neste caso escrevemos `Hello World!\n!` o que quer dizer que será imprimido "Hello World!" na janela. E o que faz `\n`? Simples, é um carácter especial que imprime uma nova linha. Chama-se *new line*.

A \ serve para inserir caracteres especiais.

%%	Por cento
\t	Tab
\r	Carriage Return (coloca o cursor no início da linha)
\a e \7	Alguns sons

Tabela 1.1: Alguns caracteres especiais em C

1.5.7 `return`

Como referi acima, a função `main` irá, neste caso, retornar um número inteiro. É aqui que o comando *return*, que quer dizer retorno, entra. Este retorna o número 0 que é o **binário para falso**.

De momento, ainda não temos muito a acrescentar sobre esta diretiva mas mais à frente iremos falar de novo sobre ele quando abordarmos funções e procedimentos.

1.5.8 Comentários

Os comentários, em programação, são colocados para auxiliar ou dar mais informações sobre alguma parte ou trecho do código. Os comentários são completamente ignorados pelo compilador.

Os comentários que iniciam por "//" começam nesse local e prolongam-se até ao final da linha em questão. Os blocos de comentários maiores começam por "/*" e terminam com "*/" e tudo o que está contido entre eles é um comentário.

Estes não devem ser utilizados para coisas óbvias como, por exemplo, "aqui somam-se as variáveis a e b". Devem-se utilizar os comentários de dupla barra em variáveis e constantes (ou para comentários curtos). Para comentar algo mais longo como funções ou classes, deverá ser utilizada a estrutura "*/".

```
1 #include <stdio.h>
2
3 int main() {
4     int n = 1; //Até ao final da linha é um comentário.
5
6     /*
7      TUDO ISTO é um comentário
8      */
9 }
```

Em C, o tipo de comentários mais utilizado é aquele que começa por "/*", independentemente do tamanho do comentário.

Capítulo 2

Dados

No capítulo 1 foi referida a existência de diversos tipos de dados, que podem variar de linguagem para linguagem e também a existência de variáveis e constantes.

> **Definição 13.** Os **tipos de dados** constituem uma variedade de valores e de operações que uma variável pode suportar. São necessários para indicar ao compilador (ou interpretador) as conversões necessárias para obter os dados da memória.

Os tipos de dados subdividem-se ainda em dois grupos: os tipos primitivos e os tipos compostos.

> **Definição 14. Tipos primitivos**, nativos ou básicos são aqueles que são fornecidos por uma linguagem de programação como um bloco de construção básico.

> **Definição 15. Tipos compostos** são aqueles que podem ser construídos numa linguagem de programação através de tipos de dados primitivos e compostos. A este processo denomina-se **composição**.

2.1 Variáveis

No capítulo 1 foi abordada a existência de variáveis e constantes que permitem armazenar dados. Nesta secção é explicado como se devem declarar as variáveis na linguagem de programação que será utilizada ao longo do resto do livro, a linguagem C.

Relembro que variáveis permitem o armazenamento de valores que podem ser alterados durante a execução de um programa. A declaração de variáveis em C é bastante simples:

```
1 tipo_de_dados nome_da_variavel;
```

Onde:

- `tipo_de_dados` corresponde ao tipo de dados que a variável vai armazenar;
- `nome_da_variavel` corresponde ao nome que a variável vai tomar, à sua identificação.

Imagine que quer criar uma variável chamada `idade` do tipo inteiro. Bastaria proceder da seguinte forma:

```
1 int idade;
```

2.2 Constantes

Em C, tal como noutras linguagens de programação, existem as constantes. Relembro que constantes permitem armazenar valores imutáveis durante a execução de um programa.

Existem diversas formas de declarar constantes em C. Iremos abordar as duas mais utilizadas: as constantes declaradas e as constantes definidas.

2.2.1 Constantes Definidas com `#define`

Chamam-se Constantes Definidas àquelas que são declaradas no cabeçalho de um ficheiro. Estas são interpretadas pelo pré-processador que procederá à substituição da constante em todo o código pelo respetivo valor. A principal vantagem deste tipo de constantes é que são sempre globais. Ora veja como se define:

```
#define identificador valor
```

Onde:

- `identificador` corresponde ao nome da constante que, convencionalmente, é escrito em maiúsculas e com *underscore* (_) a separar palavras;
- `valor` corresponde ao valor que a constante armazena.

Imagine que, por exemplo, precisa de uma constante que armazene o valor do π e que depois se calcule o perímetro de um círculo. Poderia proceder da seguinte forma:

```
#include <stdio.h>
#define PI 3.14159

int main (){

  double r = 5.0;
  double circle;

  circle = 2 * PI * r;
  printf("%f\n", circle);
  return 0;
}
```

O que acontece quando é definida uma constante através da diretiva `#define` é que quando o pré-compilador lê a definição da constante, substitui todas as ocorrências no código daquela constante pelo seu valor, literalmente.

Pode utilizar a biblioteca `math.h` que tem a constante `M_PI` com o valor do pi.

2.2.2 Constantes Declaradas com `const`

As Constantes Declaradas, ao contrário das Constantes Definidas são, tal como o próprio nome indica, declaradas no código, em linguagem C. A declaração destas constantes é extremamente semelhante à declaração das variáveis. Apenas temos que escrever `const` antes do tipo de dados. Ora veja como se declara uma constante:

```
const tipo nome = valor;
```

Onde:

- `tipo` corresponde o tipo de dados que a constante vai conter;
- `nome` corresponde ao nome da constante;
- `valor` corresponde ao conteúdo da constante.

Se tentar alterar o valor de uma constante durante a execução de um programa irá obter um erro. Analise então o seguinte excerto de código:

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

int main() {
    const double goldenRatio = (1 + sqrt(5)) / 2;

    goldenRatio = 9; // erro. A constante não pode ser alterada.

    double zero = (goldenRatio * goldenRatio) - goldenRatio - 1;
    printf("%f", zero);
    return 0;
}
```

Existem vantagens ao utilizar cada uma destas formas de declarar constantes. As constantes declaradas podem ser locais ou globais porém as definidas são sempre globais.

> **Definição 16.** Uma constante/variável **local** é uma constante/variável que está restrita a uma determinada função e só pode ser utilizada na função em que é declarada.

2.3 Números inteiros - `int`

Comecemos por abordar o tipo `int`. Esta abreviatura quer dizer *integer number*, ou seja, número inteiro (exemplos: 4500, 5, -250). Em C, pode-se definir o intervalo em que se situam os números de cada variável do tipo `int`.

> **Definição 17. Inicialização de variáveis** consiste em dar o primeiro valor a uma variável no código. Uma variável não tem que ser obrigatoriamente inicializada no código. Pode-se, por exemplo, declarar uma variável e dar-lhe o valor de uma leitura em que o utilizador introduz os dados, sendo a variável inicializada com os dados inseridos pelo utilizador.

No seguinte exemplo pode visualizar como se declara uma variável, ou seja, como se reserva um endereço da memória RAM, do tipo `int` com o nome `a`. Seguidamente é inicializada com o valor 20 e imprimida recorrendo à função `printf`.

```
1  #include <stdio.h>
2
3  int main() {
4
5      int a;
6      a = 20;
7
8      printf("Guardei o número %d.", a);
9      return 0;
10 }
```

Antes de continuar, deve ter reparado que foi utilizado um `%d` dentro do primeiro parâmetro. Este é substituído pelo valor da variável `a` quando é impresso no ecrã.

2.3.1 Função `printf` e números inteiros

Utiliza-se `%d` quando se quer imprimir o valor de uma variável dentro de uma frase. A variável deve ser colocada nos parâmetros seguintes, por ordem de ocorrência. Por exemplo:

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b;
5      a = 20;
6      b = 100;
7
8      /* imprime: O primeiro número é: 20 */
9      printf("O primeiro número é: %d.\n", a);
10     /* imprime: O segundo número é: 100 */
11     printf("O segundo número é: %d.\n", b);
12     /* imprime: O primeiro e segundo números são 20 e 100 */
13     printf("O primeiro e segundo números são %d e %d.\n", a, b);
14
15     return 0;
16 }
```

2.3.2 Modificadores `short` e `long`

Este tipo de variáveis ocupa, normalmente, entre 2 a 4 *bytes* na memória de um computador. E se quiser utilizar uma variável para um número pequeno? Não poderei gastar menos recursos? E se acontecer o contrário e precisar de um número maior?

Bit (Binary Digit) é a menor unidade de informação que pode ser armazenada ou transmitida. Um *bit* só pode assumir dois valores: 0 e 1, ou seja, o código binário. Os *bits* são representados por "B" minúsculo.

Byte (Binary Term) é um tipo de dados de computação. Normalmente é utilizado para especificar a quantidade de memória ou capacidade de armazenamento de algo. É representado por um "B" maiúsculo. Geralmente:

- **1 Byte** = 8 bits (Octeto)
- **1/2 Byte** = 4 bits (Semiocteto)

Existem os **Prefixos Binários (IEC)** que são nomes/símbolos utilizados para medidas indicando a multiplicação da unidade, neste caso *byte*, por potências de base dois. Como por exemplo os *megabibytes* (MiB) que equivale a dois elevado a vinte *bytes*.

Por outro lado, existem os prefixos do **Sistema Internacional de Unidades (SI)**, que são os mais comuns, que correspondem na unidade multiplicada por potências de base dez. Como por exemplo os *megabytes* (MB) que correspondem a dez elevado a seis.

Nestas situações, pode-se utilizar modificadores.

Definição 18. Um **modificador** consiste numa palavra-chave, numa *keyword*, que se coloca antes de um elemento de forma a modificar uma propriedade do mesmo.

Para alterar a capacidade de armazenamento, ou seja, o número de *bytes* ocupado por uma variável do tipo `int`, podem-se utilizar os modificadores `short` e `long`. Estes permitem-nos criar variáveis que ocupem um maior ou menor número de bytes, respetivamente.

Uma variável do tipo `int`, ao assumir um número de *bytes* diferentes, também está a alterar a sua capacidade de armazenamento. Assim, temos os seguintes valores:

- 1 *byte* armazena de -128 a +127
- 2 *bytes* armazenam de -32 768 a +32 767
- 4 *bytes* armazenam de -2 147 483 648 a +2 147 483 647
- 8 *bytes* armazenam de -9 223 372 036 854 775 808 a +9 223 372 036 854 775 807

A utilização destes modificadores é feita da seguinte forma:

```
short int nomeDaVariavel = 20; // ou "long"
```

Função *sizeof*

O número de *bytes* atribuído utilizando um destes modificadores pode depender do computador onde o código está a ser executado. Este "problema" depende da linguagem. Existem linguagens de programação cuja capacidade de cada tipo de variável é igual em todas as máquinas.

Para descobrirmos qual o tamanho de bytes que utiliza o seu sistema, basta recorrer à função `sizeof` da seguinte forma:

```
#include <stdio.h>

int main() {
    printf("int : %d bytes\n", sizeof(int) );
    printf("short int: %d bytes\n", sizeof(short) );
    printf("long int: %d bytes\n", sizeof(long) );
    return 0;
}
```

No computador que estou a utilizar, por exemplo, `short` refere-se a 2 *bytes*, `long` a 8 *bytes* e o tamanho padrão de `int` é 4 *bytes*.

2.3.3 Modificadores `signed` e `unsigned`

Como sabe, os números inteiros podem assumir forma positiva e negativa. Por vezes, na programação, os números negativos podem atrapalhar (ou então ajudar), dependendo do caso.

Para termos controlo sobre a "positividade" ou "negatividade" de um número, podemos atribuir os modificadores `signed` e `unsigned`. Para que uma variável possa conter tanto números positivos como negativos, devemos utilizar o modificador `signed`. Caso queira que o número seja apenas positivo, incluindo 0, utilize `unsigned`.

Tendo em conta que variáveis marcadas com o modificador `unsigned` não podem conter números negativos, podem conter um intervalo de números positivos superior ao regular. Imaginando que uma variável `int` suporta números entre -32 768 e 32 767; a mesma variável com o modificador `unsigned` irá suportar números entre 0 e 65 535.

2.4 Números reais - `float` e `double`

Além dos números inteiros, existem outros tipos de dados que nos permitem armazenar números que, ao invés de serem inteiros, são decimais (por exemplo 1,3; 5,540; etc).

Existem dois tipos de dados que nos permitem armazenar valores do tipo decimal/real, ou seja, que têm casas decimais. Estes tipos são `float` e `double`. Devem ser utilizados da seguinte forma:

```
float pi = 3.14;
double pi = 3.14159265359;
```

Como pode ter reparado, em C (e na maioria das linguagens de programação), não se utiliza a vírgula, mas sim um ponto para separar a parte inteira da decimal.

A diferença entre `float` e `double` é que o segundo ocupa mais memória que o primeiro, logo consegue armazenar números de maior dimensão.

Normalmente, o tipo float ocupa 4 *bytes* de memória RAM enquanto o segundo tipo, `double`, ocupa 8 *bytes* de memória. Mais uma vez, relembro que estes valores podem alterar de máquina para máquina.

Para quem precise de fazer cálculos mais precisos, o tipo de dados `double` é o mais aconselhado pois é o que permite uma maior extensão do valor.

2.4.1 Função `printf` e números reais

Recorrendo à função `printf` abordada na secção **1.5.7**, utiliza-se `%f` para se imprimir o valor de uma variável dos tipos `float` e `double`. A variável deve ser colocada nos parâmetros seguintes, por ordem de ocorrência. Por exemplo:

```
#include <stdio.h>

int main() {
    float piMinor = 3.14;
    double piMajor = 3.14159265359;

  // imprime: Pi pode ser 3.140000 mas, de forma mais exata, é 3.141593
    printf("Pi pode ser %f mas, de forma mais exata, é %f.", piMinor, piMajor);

    return 0;
}
```

Pode visualizar que, quando se utiliza `%f`, é utilizado um número específico de casas decimais. Caso o número de casas decimais seja mais pequeno do que o da variável original, o número é arredondado.

Pode definir o número de casas decimais que quer que sejam apresentadas da seguinte forma: **`%.{númeroDeCasasDecimais}f`**. Veja o seguinte exemplo, baseado no anterior:

```
#include <stdio.h>

int main() {
    float piMinor = 3.14;
    double piMajor = 3.14159265359;

  // imprime: Pi pode ser 3.14 mas, de forma mais exata, é 3.14159265359.
```

```
    printf("Pi pode ser \%.2f mas, de forma mais exata, é \%.11f.", piMinor, piMajor);

    return 0;
}
```

2.4.2 Notação Científica

Relembre o conceito de notação científica[1], ou seja, números no formato $n * 10^s$, ou seja, n vezes dez elevado a s. Podemos utilizar notação científica nas variáveis do tipo *float* e *double*. Veja o seguinte exemplo:

```
#include <stdio.h>

int main() {
    float num = 24E-5; // 24 x 10 elevado a -5
    printf("\%f\n", num); // imprime: 0.000240

    num = 2.45E5; //2.45 x 10^5
    printf("\%.0f", num);  // imprime: 245000

    return 0;
}
```

2.5 Caracteres - `char`

O tipo char é um género de dados que nos permite armazenar um único carácter. É declarado da seguinte forma:

```
char letra = 'P';
```

Como pode visualizar, a variável `letra` agora contém o carácter "P". Pode, ao invés de utilizar este tipo de notação, utilizar números hexadecimais, octais e decimais. Em C pode-se utilizar ASCII.

> **Definição 19. ASCII** (do inglês *American Standard Code for Information Interchange*; em português "Código Padrão Americano para o Intercâmbio de Informação") é um conjunto de códigos binários que codificam 95 sinais gráficos e 33 sinais de controlo. Dentro desses sinais estão incluídos o nosso alfabeto, sinais de pontuação e sinais matemáticos. No **Anexo I** encontra uma tabela ASCII com caracteres que podem ser utilizados em C.

2.5.1 Função `printf` e caracteres

Utiliza-se *%c* quando se quer imprimir o valor de uma variável dos tipo `char` dentro de uma frase. A variável deve ser colocada nos parâmetros seguintes, por ordem de ocorrência. Exemplo:

```
#include <stdio.h>

int main() {
    char letra = 'P';

    printf("O nome Pplware começa por %c.", letra);
    return 0;
}
```

[1] Ler pt.wikipedia.org/wiki/Notação_científica

Capítulo 3

Operadores

Neste capítulo são abordados os operadores, que são deveras importantes na modificação dos valores de variáveis.

> **Definição 20. Operadores** são símbolos para efetuar determinadas ações sobre variáveis. Na programação existem diversos tipos de operadores como, por exemplo, para efetuar operações aritméticas.

Alguns dos operadores também existem em outras áreas do conhecimento como matemática, por exemplo.

3.1 Operadores aritméticos

Os operadores aritméticos são, tal como o próprio nome indica, utilizados para efetuar operações aritméticas, ou seja, operações matemáticas como somas, subtrações, multiplicações, entre outras.

Nome	**Símbolo**	**Exemplo**
Soma	+	5 + 4 = 9
Subtração	-	154 - 10 = 144
Multiplicação	*	5,55 * 10 = 55,5
Divisão	/	40 / 2 = 20
Resto inteiro de uma divisão	%	1500 % 11 = 4

Tabela 3.1: Operadores aritméticos

3.2 Operadores de atribuição

Os operadores de atribuição servem para atribuir um determinado valor a uma variável ou constante. Existem vários operadores de atribuição e muitos deles funcionam como abreviatura para operações aritméticas. Estes operadores são *syntactic sugar*[1], ou seja, têm uma sintaxe elegante de forma a serem facilmente entendidos pelos seres humanos.

3.3 Operadores relacionais

> Pode avançar esta e as secções mais à frente por agora. Mais tarde ser-lhe-á indicado para aqui voltar. Consultar capítulo 4.

Estes operadores (relacionais) permitem-nos estabelecer relações de comparação entre diversas variáveis.

[1] Consulte en.wikipedia.org/wiki/Syntactic_sugar

Nome	Símbolo	Exemplo	Valor de `var`
Atribuição	=	var = 20	20
Adição e atribuição	`+= (var = var + n)`	var += 5	25
Subtração e atribuição	`-= (var = var - n)`	var -= 10	15
Multiplicação e atribuição	`*= (var = var * n)`	var *= 4	60
Divisão e atribuição	`/= (var = var / n)`	var /= 5	12
Resto inteiro e atribuição	`%= (var = var % n`	var %= 5	2

Tabela 3.2: Operadores de atribuição

Nome	Símbolo	Exemplo
Igualdade	==	x == y retorna 1 se x for igual a y e 0 se tiverem valores diferentes
Diferente	!=	x != y retorna 1 se x for diferente de y ou 0 se x for igual a y
Maior	>	x >40
Maior ou Igual	>=	y >= 25
Menor	<	y <20
Menor ou Igual	<=	x <= y

Tabela 3.3: Operadores relacionais

3.4 Operadores lógicos

Os operadores lógicos são normalmente utilizados em testes lógicos quando é necessário incluir mais do que uma condição para que algo aconteça. Existem três operadores lógicos que aqui são abordados: `&&`, || e `!`. Estes operadores também são *syntactic sugar*.

Nome	Operador
Operador "e"	&&
Operador "ou"	||
Negação	!

Tabela 3.4: Operadores lógicos

3.4.1 Operador `&&`

Na programação, o operador `&&` tem como função a conjugação de condições, ou seja, funciona da mesma forma que um "e" na Língua Portuguesa. Com este elemento, para que algo seja considerado verdadeiro, todos os elementos têm que o ser. Veja alguns exemplos em que este operador é utilizado para agregar condições:

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5      a = 4;
6
7      if (a > 0 && a <= 10) {
8          printf("O número %d está entre 1 e 10.", num1);
9      }
10
11   return 0;
12 }
```

A condição acima pode ser transcrita para uma linguagem lógica da seguinte forma: "**Se** a for maior que 0 **e** menor ou igual a 10, então o código é executado".

Este operador pode ser utilizado mais do que uma vez no interior de um teste condições pois podemos intersetar duas ou mais condições, bastando apenas adicionar `&&` e a outra condição a ser adicionada.

3.4.2 Operador ||

À semelhança do operador anterior, o operador || também tem uma função que pode ser comparada a uma pequena palavra do Português: à palavra "ou". Com este elemento, para que algo seja considerado verdadeiro, basta que um elemento o seja.

Com este operador, podemos executar um trecho de código que **satisfaz** uma das **várias condições** existentes. Por exemplo:

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5      a = 3;
6
7      if (a == 3 || a == 5) {
8          printf("O número é 3 ou 5.\n");
9      }
10
11     return 0;
12 }
```

Numa linguagem puramente lógica, podemos converter a condição anterior para o seguinte: "Se `a` for 3 **ou** 5, então a linha 8 é executada".

3.4.3 Operador *!*

O operador `!` é utilizado para indicar a negação, ou numa linguagem corrente, "o contrário de". Quando é utilizado juntamente com uma condição, quer dizer que o código que está condicionado depende se a negação da condição é satisfeita, tal como o antónimo de uma palavra. Ora veja o seguinte exemplo:

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5      a = 0;
6
7      if (!a) {
8          printf("A variável é 0.\n");
9      }
10
11     return 0;
12 }
```

Relembrando que o número 0 é considerado o binário para falso e que, qualquer número diferente de 0 é, na linguagem C, considerado como verdadeiro, a condição acima pode ser traduzida para: "Se `a` **não** for diferente de 0, então executa o código".

3.4.4 Operadores de decremento e incremento

Os operadores de incremento e decremento são dois operadores essenciais na vida de qualquer programador. Imagine que necessita de contar um número cujo qual nem o programador sabe o fim. Vai adicionar/subtrair valor a valor? Não, não é necessário.

```
1  #include <stdio.h>
2
3  int main() {
4      int num;
5
6      num = 1;
```

```
7       printf("O número é %d\n", num);
8
9       num = num + 1;
10      printf("O número é %d\n", num);
11
12      num += 1;
13      printf("O número é %d\n", num);
14
15      return 0;
16  }
```

O código acima imprime a frase "O número é x", onde x corresponde ao valor da variável `num` nesse momento. Este algoritmo faz com que a variável inicialmente tenha um valor de 1, passando por 2, chegando finalmente a 3. As linhas `num = num + 1` e `num += 1` são equivalentes.

Existe outra forma mais simples para adicionar/subtrair um valor a uma variável: através dos operadores ++ e −. Veja como ficaria o código anterior com estes operadores:

```
1  #include <stdio.h>
2
3  int main() {
4      int num;
5
6      num = 1;
7      printf("O número é %d\n", num);
8
9      num++;
10     printf("O número é %d\n", num);
11
12     num++;
13     printf("O número é %d\n", num);
14
15     return 0;
16 }
```

Este último trecho de código irá imprimir o mesmo que o código anterior. Agora, a diferença aparenta não ser imensa, porém estes operadores serão extremamente úteis em testes lógicos/de repetição.

Para remover uma unidade bastaria colocar − ao invés de ++. Então podemos concluir que este operador torna o incremento/decremento mais rápido, mas só funciona quando o assunto é uma unidade. Veja o seguinte exemplo:

```
1  #include <stdio.h>
2
3  int main() {
4      int num;
5
6      num = 1;
7      printf("O número é %d\n", num);
8
9      num--;
10     printf("O número é %d\n", num);
11
12     return 0;
13 }
```

Posição do operador

Estes operadores podem ocupar duas posições: antes ou depois do nome da variável, ou seja, posso colocar tanto `num++` como `++num`. Quando são utilizados isoladamente, não existe nenhuma diferença. Porém, quando estamos a efetuar uma atribuição, existem diferenças. Analise o seguinte exemplo:

```
1  #include <stdio.h>
2
```

```
int main() {
    int a, b, c;

    a = 0;
    b = a++;
    c = ++a;

    printf("A: %d, B: %d, C: %d", a, b, c);

    return 0;
}
```

Em primeiro lugar, são declaradas três variáveis: `a`, `b` e `c`. Seguidamente, é atribuído o valor 0 à primeira variável. Quando o valor `a++` é atribuído à variável `b`, esta fica com valor 0 ou 1? E o que acontece com a variável `a`? A variável `b` irá assumir o valor 0, e um valor é incrementado a `a` ficando esta com o valor 1, ou seja, `b = a++` é um atalho para o seguinte:

```
b = a;
a++;
```

Depois desta atribuição, é atribuída à variável `c`, o valor `++a`, ou seja, primeiro é incrementado o valor da variável `a` e só depois é que o valor de esta é atribuído à variável `c`. Então, isto é um atalho para o seguinte:

```
++a;
c = a;
```

Capítulo 4

Controlo de Fluxo

Este quarto capítulo tem como objetivo mostrar-lhe as formas de controlar o fluxo de uma aplicação, de um algoritmo, de um programa.

> **Definição 21. Controlo de Fluxo** refere-se ao controlo que se tem sobre a ordem de comandos a serem executados no decorrer de um programa.

Ao controlar o fluxo pode-se direcionar o utilizador para as ações que este escolheu e executar apenas trechos de código dependendo de uma determinada condição, controlando a ordem pela qual os comandos são executados.

Existem diversas estruturas que nos permitem controlar o fluxo de uma aplicação. A maioria das que são aqui abordadas são transversais à maioria das linguagens de programação existentes.

Antes de continuar aconselhamos a que reveja os **operadores relacionais e lógicos** no capítulo 3.

4.1 Estrutura `if/else`

A primeira estrutura a abordar é a conhecida `if else` o que, numa tradução literal para Português, quer dizer "se caso contrário". Com esta estrutura, um determinado trecho de código pode ser executado dependendo do resultado de um teste lógico.

> **Definição 22.** Um **teste lógico** consiste na determinação da verdade ou falsidade de uma condição.

Sintaxe

```
if (condição) {
    // código a executar caso a condição seja verificada
} else {
    // caso contrário, isto é executado.
}
```

Exemplo

Imagine que necessita criar um pequeno programa que deve imprimir se o valor de uma variável é maior, menor ou igual a 50. Como irá proceder? A criação deste algoritmo é deveras simples, bastando inicializar a variável e efetuar um teste lógico. Veja então como este problema poderia ser resolvido:

```
#include <stdio.h>

int main() {
    int n = 25;

    if (n >= 50) {
        printf("O valor %d é maior ou igual a 50.", n);
    } else {
```

```
9           printf("O valor %d é menor que 50.", n);
10      }
11
12      return 0;
13  }
```

Se executar o trecho de código anterior, a mensagem `O valor 25 é menor que 50.` será imprimida, pois não se verificou a condição `n >= 50`, executando-se o código dentro do bloco `else`.

Imagine agora que precisa verificar as seguintes três condições:

- É maior que 50?
- É igual a 50?
- É menor que 50?

O caminho mais óbvio seria o seguinte:

```
1  if (n > 50) {
2      printf("O valor %d é maior que 50.\n", n);
3  }
4
5  if (n < 50) {
6      printf("O valor %d é menor que 50.\n", n);
7  }
8
9  if (n == 50) {
10     printf("A variável é igual a 50.\n");
11 }
```

O algoritmo acima é um pouco repetitivo e extenso; pode ser simplificado ao ser agregado em apenas um teste sequencial como seguinte:

```
1  if (n == 50) {
2      printf("A variável é igual a 50.\n"); //A
3  } else if (n < 50) {
4      printf("O valor %d é menor que 50.\n", n); //B
5  } else {
6      printf("O valor %d é maior que 50.\n", n); //C
7  }
```

Podemos "traduzir" o código anterior para uma linguagem corrente da seguinte forma: `Se n for igual a 50; então faz A, ou se n for menor que 50 faz B; caso contrário faz C.`

4.2 Estrutura `while`

Outra estrutura para controlar o fluxo que é muito importante é a estrutura `while`, que em Português significa "enquanto". Esta estrutura permite repetir um determinado trecho de código enquanto uma condição for verdadeira.

Sintaxe

```
1  while(condição) {
2     //Algo acontece
3  }
```

Ora vejamos um exemplo: precisa imprimir todos os números entre 0 e 100 (inclusive). Para isso não é necessário utilizar o comando `printf` 101 vezes, bastando utilizar a repetição `while`. Ora veja:

```
1  #include <stdio.h>
2
3  int main() {
4      int num = 0;
5
```

```
    while(num <= 100) {
      num++;
        printf("%d\n", num);
    }

    return 0;
}
```

Pode-se traduzir o trecho anterior para uma linguagem corrente da seguinte forma: `Enquanto num for menor ou igual a 100, imprime a variável num e incrementa-lhe um valor.`

4.3 Estrutura `switch`

A estrutura `switch` está presente na maioria das linguagens de programação, que numa tradução literal para Português significa "interruptor". Esta estrutura é deveras útil quando temos que executar uma ação dependendo do valor de uma variável.

A estrutura de controlo `switch` pode ser utilizada como forma de abreviar um teste lógico `if else` longo.

Sintaxe

```
switch(variavel) {
    case "valorUm":
        //Código da operação
        break;
    case "valorDois":
        //Código da operação
        break;

    //...

    default:
        //Código executado caso não seja nenhuma das opções anteriores
        break;
}
```

Imagine que tem que pedir ao utilizador um valor e que vai executar uma ação dependendo o valor que o utilizador escolheu. Esse menu tem 5 opções. Poderia resolver este problema da seguinte forma:

```
#include <stdio.h>

int main() {
    int opcao;

    printf("Insira a opção:\n");
    scanf("%d", &opcao);

    switch(opcao) {
        case 1:
            printf("Escolheu a opção 1");
            break;
        case 2:
            printf("Escolheu a opção 2");
            break;
        case 3:
            printf("Escolheu a opção 3");
            break;
        case 4:
            printf("Escolheu a opção 4");
            break;
        case 5:
            printf("Escolheu a opção 5");
            break;
        default:
```

```
            printf("Opção inexistente.");
            break;
    }

    return 0;
}
```

O código acima faz: em primeiro lugar, tudo depende do valor da variável `opcao`. Caso seja 1, será imprimida a mensagem `Escolheu a opção 1` e por aí a diante. Caso a opção inserida não exista no código, o código contido em `default` irá ser executado.

Mais à frente iremos falar mais sobre o `break`. Por agora não lhe dê muita importância, mas coloque-o sempre.

O algoritmo anteriormente reproduzido pode também tomar a forma de uma sequência de `if else`, embora com uma menor legibilidade. Ora veja:

```
#include <stdio.h>

int main() {
    int option;

    printf("Insira a opção:\n");
    scanf("%d", &option);

    if (option == 1) {
        printf("Escolheu a opção 1");
    } else if (option == 2) {
        printf("Escolheu a opção 2");
    } else if (option == 3) {
        printf("Escolheu a opção 3");
    } else if (option == 4) {
        printf("Escolheu a opção 4");
    } else if (option == 5) {
        printf("Escolheu a opção 5");
    } else {
        printf("Opção inexistente.");
    }

    return 0;
}
```

4.4 Estrutura `do/while`

Outra forma de controlar o fluxo a abordar é a estrutura `do while`, que tem um nome semelhante à já abordada `while`. A diferença existente entre essas duas estruturas é pequena, mas importante.

Ao contrário do que acontece na estrutura `while`, no `do while`, o código é executado primeiro e só depois é que a condição é testada. Se a condição for verdadeira, o código é executado novamente. Podemos concluir que esta estrutura obriga o código a ser **executado pelo menos uma vez**.

Sintaxe

```
do
{
   //código a ser repetido
} while (condição);
```

Imagine agora que precisa criar uma pequena calculadora (ainda em linha de comandos) que receba dois número e que, posteriormente, efetua uma soma, subtração, multiplicação ou divisão. Esta calculadora, após cada cálculo deverá pedir ao utilizador para inserir se quer continuar a realizar cálculos ou não. Poderíamos proceder da seguinte forma:

```
#include <stdio.h>

int main() {
```

```
  int calcular;

  do {

    char operacao;
    float num1,
        num2;

    // limpeza do buffer. ou __fpurge(stdin); em linux
    fflush(stdin);

    printf("Escolha a operação [+ - * / ]: ");
    scanf("%c", &operacao);

    printf("Insira o primeiro número: ");
    scanf("%f", &num1);

    printf("Insira o segundo número: ");
    scanf("%f", &num2);

    switch(operacao) {
      case '+':
        printf("%.2f + %.2f = %.2f\n", num1, num2, num1 + num2);
        break;
      case '-':
        printf("%.2f - %.2f = %.2f\n", num1, num2, num1 - num2);
        break;
      case '*':
        printf("%.2f * %.2f = %.2f\n", num1, num2, num1 * num2);
        break;
      case '/':
        printf("%.2f / %.2f = %.2f\n", num1, num2, num1 / num2);
        break;
      default:
        printf("Digitou uma operação inválida.\n");
        break;
    }

    printf("Insira 0 para sair ou 1 para continuar: ");
    scanf("%d", &calcular);

  /* Verifica-se o valor da variável calcular. Se for 0 é considerado falso
  e o código não é executado mais vez nenhuma. Caso seja um número diferente
  de 0, a condição retornará um valor que representa true (todos os números
  exceto 0) e continuar-se-à a executar o código. */
  } while (calcular);

  return 0;

}
```

4.5 Estrutura `for`

A última estrutura de controlo de fluxo a abordar, mas não o menos importante, é a estrutura `for`. Esta estrutura *loop* é um pouco mais complexa que as anteriores, mas muito útil e permite reutilizar código.

> **Definição 23.** O termo inglês ***loop*** refere-se a todas as estruturas que efetuam repetição: `while`, `do while` e `for`.

Sintaxe

```
for(inicio_do_loop ; condição ; termino_de_cada_iteração) {
 //código a ser executado
}
```

Onde:

- `inicio_do_loop` → uma ação que é executada no início do ciclo das repetições;
- `condição` → a condição para que o código seja executado;
- `termino_de_cada_iteração` → uma ação a executar no final de cada iteração.

Imagine agora que precisa de imprimir todos os números pares de 0 a 1000. Para isso poderia recorrer à estrutura `while` e a uma condição `if` da seguinte forma:

```
#include <stdio.h>

int main() {
    int num = 0;

    while (num <= 1000) {
        if (num % 2 = 0) {
            printf("%d\n", num);
        }

        num++;
    }

    return 0;
}
```

Utilizando a estrutura `for`, o algoritmo acima poderia ser reduzido ao seguinte:

```
#include <stdio.h>

int main() {
    for(int num = 0; num <= 1000; num++) {
        if (num % 2 = 0) {
            printf("%d\n", num);
        }
    }

    return 0;
}
```

4.6 Interrupção do fluxo

As estruturas de controlo de fluxo são muito importantes, mas aprender como se as interrompe também o é. Por vezes é necessário interromper uma determinada repetição dependendo de um teste lógico interno à repetição.

4.6.1 Terminar ciclo com `break`

O `break` permite que interrompamos o fluxo em *loops* e na estrutura `switch`. Se não colocarmos este comando no final de cada caso da estrutura `switch`, o código continuaria a ser executado, incluindo as restantes opções.

Imagine que por algum motivo precisa encontrar o primeiro número divisível por 17, 18 e 20 entre 1 e 1000000. Iria, provavelmente, utilizar a estrutura `for` para percorrer todos estes números. Veja como poderíamos fazer:

```
#include <stdio.h>

int main() {
    int num = 0;

        for (int count = 1; count <= 1000000; count++) {
            if(num == 0) {
                if((count % 17 == 0) && (count % 18 == 0) && (count % 20 == 0)) {
                    num = count;
                }
            }
        }

```

```
14          printf("O número divisível por 17, 18 e 20 entre 1 e 1000000 é: %d", num);
15          return 0;
16  }
```

Depois de encontrar o número em questão, que é 3060, é necessário continuar a executar o `loop`? Seriam executadas mais 996940 iterações (repetições). Tal não é necessário e podemos poupar os recursos consumidos, parando o ciclo de repetições. Ora veja:

```
1   #include <stdio.h>
2
3   int main() {
4           int num = 0;
5
6           for (int count = 1; count <= 1000000; count++) {
7               if(num == 0 && (count%17==0) && (count%18==0) && (count%20==0)) {
8                   num = count;
9                   break;
10              }
11          }
12
13          printf("O número divisível por 17, 18 e 20 entre 1 e 1000000 é: %d", num);
14          return 0;
15  }
```

4.6.2 Terminar iteração com `continue`

O comando `continue` é parecido ao anterior só que, em vez de cancelar todo o ciclo de repetições, salta para a próxima iteração, cancelando a atual.

Precisa de somar todos os números inteiros entre 0 e 1000 que não são múltiplos de 2 nem de 3. Para o fazer, irá ter que saltar todas as somas em que o número atual é múltiplo de 2 e 3. Veja então uma proposta de como poderia ficar:

```
1   #include <stdio.h>
2
3   int main() {
4       int sum = 0;
5
6       for (int count = 1; count <= 1000; count++) {
7           if(count%2 == 0 || count%3 == 0) {
8               continue;
9           }
10
11          sum += count;
12      }
13
14      printf("Soma %d", sum);
15      return 0;
16  }
```

O que acontece é se percorrem todos os números de 1 a 1000, e se for divisível por 2 ou por 3, salto para o próximo número. Caso esta condição não se verifique, adiciono o número atual à soma.

Capítulo 5

Funções e Procedimentos

Neste capítulo é abordada uma parte fundamental da programação funcional, ou seja, as funções e procedimentos.

> **Definição 24.** Uma **função** é um bloco de código, que tal como o próprio nome indica, tem uma função própria, ou seja, que serve para um finalidade específica.

> **Definição 25. Procedimentos** são blocos de código que contêm uma função específica. A diferença entre funções e procedimentos é que os procedimentos não retornam qualquer valor.

Ao invés de colocarmos um determinado trecho de código diversas vezes, basta criarmos uma função com esse código e de cada vez que for necessário, precisamos apenas de invocar a função. As funções permitem-nos, por exemplo, reutilizar código.

> **Definição 26. Invocar** uma função é o nome que se dá quando o nome de uma função é mencionado no código e esta é "chamada".

Durante os seus anos de programador irá criar milhares de funções para os mais diversos propósitos. Então é recomendável que vá guardando as funções que utiliza porque futuramente poderá vir a precisar delas.

Existe uma função que tem vindo a ser sempre utilizada: o `main`. É a função principal do programa; aquela que é executada automaticamente quando um programa é iniciado.

5.1 Criação de funções

Então, o primeiro passo a dar é saber como se criam funções.

Sintaxe

```
tipo_dado_retorno nome_da_funcao(parametros) {
    // conteúdo da função
}
```

Onde:

- **tipo_dado_retorno** corresponde ao tipo de dados que a função vai devolver (retornar);
- **nome_da_funcao** corresponde ao nome da função em questão;
- **parametros** corresponde aos parâmetros da função, algo que será abordado mais à frente.

Imagine que precisa de um procedimento que imprime a mensagem "Hello World!" no ecrã. Poderia fazer isso da seguinte forma:

```
#include <stdio.h>

int main() {
    dizHello();
```

```
5       return 0;
6   }
7
8   void dizHello() {
9       printf("Olá Mundo!\n");
10  }
```

A função chama-se `dizHello` e não retorna nenhuns dados (`void`). O que está contido entre chavetas é o código que é executado quando o procedimento é chamado.

5.1.1 Argumentos e parâmetros

No exemplo acima é possível visualizar que tanto na invocação da função, como na sua definição foi colocado um par de parênteses sem nada. Isto acontece porque não existem parâmetros.

> **Definição 27. Parâmetros** é o conjunto de elementos que uma função pode receber (*input*).

Imagine agora que necessita de uma função que efetua uma soma: como enviamos os valores para uma função? Definindo o nome e o tipo de parâmetros. Ora veja:

```
1   void soma(int n1, int n2) {
2       int soma = n1 + n2;
3   }
```

A função representada no trecho acima, denominada `soma`, tem dois parâmetros, o `n1` e o `n2`, ambos do tipo `int`. Para utilizar a função, basta então proceder do seguinte modo:

```
1   soma(4, 6);
```

Os números 4 e 6 (que podiam ser quaisquer outros) são denominados argumentos. A função armazena os dois argumentos na variável `soma` que apenas está disponível dentro da função, o valor da soma entre os dois números.

> **Definição 28. Argumento** é o nome dado a um valor que é enviado para a função.

5.1.2 Retorno de uma função

Geralmente uma função retorna um valor de forma a poder ser utilizado para outro fim. Para isso, ao invés de colocarmos `void` na declaração da função, devemos colocar qualquer outro tipo de dados.

Imagine que quer que uma função retorne o valor de uma soma entre dois números. Poderia proceder da seguinte forma:

```
1   int soma(int n1, int n2) {
2       return n1 + n2;
3   }
```

A função acima retorna dados do tipo `int` e tem dois parâmetros: o `n1` e `n2`. Podemos aplicar esta função, por exemplo, no seguinte contexto, onde o programa efetua a soma de dois números dados pelo o utilizador.

```
1   #include <stdio.h>
2
3   int main() {
4       int a, b;
5
6       printf("Insira o primeiro número: ");
7       scanf("%d", &a);
8
9       printf("Insira o segundo número: ");
10      scanf("%d", &b);
11
12      printf("A soma é %d", soma(a,b));
13      return 0;
14  }
15
```

```
16 int soma(int n1, int n2) {
17     return n1 + n2;
18 }
```

5.2 Algumas funções úteis

Aqui encontram-se algumas funções que são úteis no seguimento deste livro. Algumas são relativas à entrada e saída de dados, ou seja, funções que podem receber dados inseridos pelo utilizador e funções que permitem mostrar dados no ecrã. Outras relativas à matemática. Ao longo do livro, outras funções serão abordadas.

5.2.1 Função `puts`

A função `puts` serve simplesmente para imprimir texto no ecrã. Ao contrário da função `printf` não nos permite imprimir texto formatado.

Sintaxe

```
1 puts("frase");
```

Ao imprimir uma frase com esta função, o carácter correspondente a uma nova linha é sempre adicionado ao final da mesma.

Imagine agora que precisa de imprimir uma mensagem que não contem nenhum valor variável e a mensagem era "Bem-vindo ao programa XYZ!". Poderia fazê-lo da seguinte forma:

```
1 puts("Bem-vindo ao programa XYZ!");
```

5.2.2 Função `scanf`

A função `scanf` permite-nos obter diversos tipos de dados do utilizador. A utilização desta função é semelhante à da já conhecida função `printf`; são como um "espelho" uma da outra. A `scanf` reconhece o *input* e a `printf` formata o *output*.

Sintaxe

```
1 scanf(fraseFormatada, &variaveis...);
```

Onde:

- `fraseFormatada` corresponde à formatação do que irá ser imprimido com os espaços para as variáveis;
- `variaveis` corresponde às variáveis onde vão ser armazenados os valores obtidos por ordem de ocorrência. O nome da variável deve ser sempre precedido por um `&`;

Imagine agora que vai criar um algoritmo que peça a idade ao utilizador e a imprima logo de seguida. Poderia fazer isto da seguinte forma:

```
1  #include <stdio.h>
2
3  int main() {
4      int idade;
5
6      printf("Digite a sua idade: ");
7      scanf("%d", &idade);
8
9      printf("A sua idade é %d", idade);
10     return 0;
11 }
```

Como pode verificar através da sétima linha do trecho de código anterior, o primeiro parâmetro da função `scanf` deve ser uma *string* com o tipo de caracteres a ser inserido. Todos os parâmetros

seguintes deverão ser o nome das variáveis às quais se quer atribuir um valor, precedidos por `&`.

Ao se executar o trecho de código anterior, a mensagem "Digite a sua idade:" irá aparecer o ecrã e o cursor irá posicionar-se logo após essa frase aguardando que um valor seja inserido. Deve-se inserir o valor e premir a tecla *enter*. Depois será imprimida uma mensagem com a idade inserida.

Relembro que pode utilizar as seguintes expressões para definir o tipo de dados a ser introduzido:

- `%d` → Números inteiros (`int`);
- `%f` → Números decimais (`float` e `double`);
- `%c` → Caracteres (`char`).

Podem-se pedir mais do que um valor com a função `scanf`. Ora veja o seguinte exemplo:

```
#include <stdio.h>

int main()
{
    int num1, num2;

    printf("Digite dois números: ");
    scanf("%d %d", &num1, &num2);

    printf("Os números que digitou são %d e %d.", num1, num2);

    return 0;
}
```

Assim, quando executar o código acima, terá que escrever dois números, separados por um espaço, *tab* ou *enter*.

5.2.3 Função `getchar`

Existe uma forma simplicíssima de pedir caracteres ao utilizador: utilizando a função `getchar`. Basta igualar uma variável à função. Esta função é recomendável quando se quer receber um único carácter numa linha.

Sintaxe

```
variavel = getchar();
```

No seguinte exemplo é pedido para inserir a primeira letra do seu nome.

```
#include <stdio.h>

int main() {
    printf("Insira a primeira letra do seu nome: ");
    char letra = getchar();

    printf("A primeira letra do seu nome é %c.", letra);
    return 0;
}
```

5.2.4 Limpeza do *buffer*

Quando se fala em entrada e saída de dados, deve-se ter em conta o *buffer* e a sua limpeza, pois é algo extremamente importante que pode fazer a diferença entre um programa que funciona e um que não funciona.

> **Definição 29.** *Buffer* é o nome dado à região da memória de armazenamento física que é utilizada para armazenar temporariamente dados.

Ora analise o seguinte código:

```
#include <stdio.h>

```

```c
int main()
{
    char letra1, letra2;

    printf("Insira a primeira letra do seu nome: ");
    scanf("%c", &letra1);

    printf("E agora a última: ");
    scanf("%c", &letra2);

    printf("O seu nome começa com \"%c\" e termina com \"%c\".", letra1, letra2);

    return 0;
}
```

Ao olhar para o código acima, provavelmente pensará que tudo irá correr como previsto: executa-se o programa, digitam-se duas letras ("X" e "Y", por exemplo) e depois é imprimida a mensagem "O seu nome começa com X e termina com Y.". Infelizmente, não é isso que acontece.

Se executar o algoritmo acima, irá inserir a primeira letra, clicar na tecla *enter*, mas depois o programa irá chegar ao fim dizendo que o seu nome termina com a letra "" (em branco). Por que é que isto acontece? Quando a tecla *enter* é premida, o programa submete a letra inserida, mas o carácter correspondente à tecla *enter*, `\n`, também fica na memória. Assim, quando é pedido um novo carácter, o que estava na memória é automaticamente submetido.

Para que isso não aconteça, basta limpar o *buffer* antes de pedir dados novamente ao utilizador. No Windows pode ser utilizada a função `fflush` e em sistemas operativos Linux a função `__fpurge`. Então, ficaria assim:

```c
#include <stdio.h>

int main()
{
    char letra1, letra2;

    printf("Insira a primeira letra do seu nome: ");
    scanf("%c",&letra1);

  // stdin corresponde à entrada teclado
    fflush(stdin);
    __fpurge(stdin);

    printf("E agora a última: ");
    scanf("%c",&letra2);

    printf("O seu nome começa com \"%c\" e termina com \"%c\".", letra1, letra2);

    return 0;
}
```

A utilização deste tipo de funções não é recomendável, pois não é uma convenção da linguagem C. Então pode tomar ações diferentes dependendo do compilador. O recomendável é utilizar funções que não incluam "lixo" quando é necessário ler algo do utilizador.

5.2.5 Função `rand`

Durante a sua jornada no mundo da programação irá precisar de gerar números aleatórios para os mais diversos fins. Em C podemos gerar números aleatórios recorrendo à função `rand`. Para poder utilizar esta função deve-se incluir a biblioteca `stdlib.h`.

Sintaxe

```c
int numero = rand();
```

Imagine que precisa gerar um número aleatório entre 1 e 10. Em primeiro lugar teria que obter o resto da divisão de `rand` por 10 e somar 1. Ora veja:

```
int numero = (rand() % 10) + 1;
```

Se experimentar executar um algoritmo que contenha a linha de código acima, irá verificar que o número gerado é sempre o mesmo, mas ninguém quer que o número gerado seja sempre o mesmo. Então, temos que "semear" a "semente" que vai dar origem ao número. Para o fazer deve-se recorrer à função `srand`.

A função `srand` permite adicionar um número como ponto de partida de forma a gerar um número aleatório. Podemos, por exemplo, gerar um número baseado na hora e tempo atuais. Ora veja um exemplo:

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

int main() {
  /* "semeia-se" utilizando o tempo atual num
  tipo unsigned, ou seja, só com valores positivos */
  srand((unsigned)time(NULL));
  int numero = (rand() % 10) + 1;

  printf("O número gerado é %d.", numero);
  return 0;
}
```

Agora, mesmo que execute o mesmo código várias vezes, irá verificar que o número gerado é diferente na maioria das vezes. Não se esqueça de incluir a biblioteca `time.h` para poder utilizar a função `time()`.

Capítulo 6

Arrays

Até este ponto do livro, apenas têm sido abordadas variáveis que contêm um e apenas um valor. Chegou a altura de falar de uma estrutura de dados muito importante no mundo da programação, os *arrays*.

> **Definição 30.** *Arrays* (também conhecidos por "tabelas" ou "vetores") são estruturas de dados que permitem armazenar múltiplos valores em posições bem definidas.

Os *arrays* são como matrizes/tabelas de dados onde cada dado se encontra localizado numa determinada posição que pode ser acedida através de "coordenadas", através de um índice (por exemplo, índice 4 para a quarta posição). Existem os *arrays* unidimensionais e multidimensionais.

6.1 Arrays unidimensionais

Os *arrays* unidimensionais podem ser comparados a tabelas com uma única coluna, mas com diversas linhas. São o tipo mais simples de *array*. Eis a declaração geral de um *array*:

```
tipo nomeVariavel[numeroDeElementos];
```

Onde:

- `tipo` corresponde ao tipo de dados que o *array* vai armazenar;
- `nomeVariavel` corresponde ao nome do *array*;
- `numeroDeElementos` corresponde ao número máximo de elementos que o *array* irá conter.

Os *arrays* são *zero-index*, ou seja, a primeira posição é 0 e não 1. Para aceder à última posição de um *array* com 6 linhas, deve-se pedir o que está contido na posição 5. Ora veja um exemplo:

```
int idades[10]; // array de 10 elementos

idades[0] = 14; // atribuição correta
idades[4] = 12; // atribuição correta
idades[7] = 15; // atribuição correta
idades[10] = 20; // atribuição incorreta (tamanho máximo do array ultrapassado)
```

No exemplo anterior, a última declaração está errada porque o índice máximo do *array* foi ultrapassado, ou seja, tentou-se aceder a uma posição inexistente. O *array* tem 10 elementos, 10 posições. O primeiro é no índice 0 e o último no índice 9.

À semelhança do que acontece com as variáveis, pode-se atribuir o valor dos *arrays* no momento da declaração. Por exemplo:

```
int idades[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

6.2 Arrays multidimensionais

Um *array* multidimensional é um *array* que tem mais do que uma dimensão, ou seja, se comparamos a uma tabela tem mais do que uma coluna.

```
tipo nome[linhas][colunas];
```

Relativamente à contagem das posições, começa-se sempre no 0, tal como acontece com os *arrays* unidimensionais. Analise então o seguinte código:

```
float notas[6][5]; // declaração de um array com 6 linhas e 5 colunas

notas[0][0] = 18.7; // linha 1, coluna 1
notas[0][1] = 15.4; // linha 1, coluna 2
notas[3][2] = 19.6; // linha 4, coluna 3
notas[5][4] = 17.5; // linha 6, coluna 5
notas[4][3] = 57.5; // linha 5, coluna 4
notas[6][0] = 20.0; // excedeu o máximo de linhas (6)
notas[5][5] = 17.4; // excedeu o máximo de colunas (5)
```

No exemplo anterior é possível visualizar a criação de um *array* com 6 linhas e 5 colunas. Este *array*, com os dados inseridos, poderia ser traduzido na seguinte tabela:

18.7	15.4			
		19.6		
			57.5	
				17.5

Tabela 6.1: Representação de um *array* 6 por 5

Tal como nos *arrays* unidimensionais, podemos adicionar os valores ao *array* multidimensional no momento da sua declaração. Veja então o seguinte exemplo:

```
int idades[2][4] = {
    {1, 2, 3, 4},
    {0, 1, 2, 3}};
```

Capítulo 7

Apontadores

Este capítulo foca num tema que não tem a mesma significância em todas as linguagens de programação, mas é algo que torna mais clara a visão do funcionamento do computador e da gestão da memória RAM. O tema em questão é os apontadores.

Para os computadores, tudo se resume a *bits*, ou seja, a zeros e uns. Então, para os computadores, as diferenças que nós conhecemos entre os diversos tipos de dados (`char`, `int`, `double`, `float`, etc) são praticamente inexistentes.

Quando uma variável é declarada, uma porção de memória RAM é reservada. Então todas as variáveis/*arrays* têm um endereço único. Os apontadores são um tipo de variáveis que podem armazenar endereços da memória.

> **Definição 31. Apontadores**, também conhecidos por ponteiros ou *pointers*, são um tipo de dados que permite armazenar um endereço da memória, representados, geralmente, em números hexadecimais.

7.1 Tamanho e endereço de uma variável

Primeiramente, deve-se recordar que todos os tipos de dados ocupam um espaço diferente na memória RAM e que para saber quantas *bytes* ocupam um determinado tipo se recorrer à função `sizeof`. Para saber o quanto ocupa qualquer variável do tipo `int`, faria o seguinte:

```
sizeof(int);
```

Para saber, por exemplo, a quantidade de *bytes* ocupadas pelos quatro tipos principais de dados e o endereço da variável utilizada, executaria o seguinte:

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

int main()
{
  char caractere;
  int inteiro;
  float Float;
  double Double;

  printf("Tipo\tNum de Bytes\tEndereço\n");
  printf("-----------------------------------\n");
  printf("Char\t%d byte \t\t%p\n", sizeof(caractere), &caractere);
  printf("Inteiro\t%d bytes \t%p\n", sizeof(inteiro), &inteiro);
  printf("Float\t%d bytes \t%p\n", sizeof(Float), &Float);
  printf("Double\t%d bytes \t%p\n", sizeof(Double), &Double);

  return 0;
}
```

O código acima irá gerar uma espécie de tabela e utiliza `%p` com a função `printf` de forma a imprimir um *pointer*. No meu caso, recebi algo semelhante ao seguinte:

Tipo	Num de Bytes	Endereço
Char	1 byte	0028FF1F
Inteiro	4 bytes	0028FF18
Float	4 bytes	0028FF14
Double	8 bytes	0028FF08

Tabela 7.1: Tamanho e endereço de variáveis

Relembro que o endereço que aparece na tabela 7.1 é o endereço corresponde à posição da variável utilizada para saber o número de *bytes* que o tipo em questão ocupava.

Pode verificar que todos os tipos de dados, exceto o tipo `char`, ocupam mais do que um *byte* na memória RAM. Então, o endereço que é imprimido corresponde apenas ao **primeiro *byte*** ocupado pela variável.

Quando imprimiu o endereço das variáveis no excerto de código anterior, verificou que o endereço imprimido não contém apenas números, mas também letras. Ao tipo de numeração que visualizou dá-se o nome **hexadecimal**.

Hexadecimal para Decimal

A numeração hexadecimal tem como base 16 dígitos, enquanto que a decimal tem como base 10 dígitos. A tabela 10.2 (nos anexos) mostra a correspondência entre um carácter hexadecimal a um carácter decimal.

Ir-se-á converter o número hexadecimal `0028FF1F` para decimal. Em primeiro lugar, ter-se-á que tirar os primeiros dois zeros que são, neste caso, desprezíveis.

De seguida é necessário conhecer o significado de cada dígito hexadecimal em número decimal que está presente nesse número. Em decimal, os números 2, 8, F e 1 são, respetivamente, 2, 8, 15 e 1.

Para efetuar a conversão, tem que multiplicar cada um desses números por uma potência de base 16 cujo expoente é igual à posição de cada um dos números (da direita para a esquerda). Então, tem-se:

$(2 * 16^5) + (8 * 16^4) + (15 * 16^3) + (15 * 16^2) + (1 * 16^1) + (15 * 16^0) =$

$= 2097152 + 524288 + 61440 + 3840 + 16 + 15 =$

$= 2686751$

Sabendo que a variável `inteiro` ocupa 4 *bytes* e que o primeiro *byte* se localiza na posição 2686744 (em decimal), poderemos dizer que a variável completa ocupa os *bytes* cujas posições são 2686744, 2686745, 2686746 e 2686747 (em decimal).

7.2 Declaração de apontadores

A declaração deste tipo de variáveis é bastante semelhante à declaração dos outros tipos de variáveis. A única diferença é que de deve colocar um * (asterisco) antes do nome da variável ou depois do tipo de dados. O tipo de dados de um apontadores deve ser igual ao tipo de dados da variável que para a qual ele vai apontar.

A declaração de apontadores faz-se da seguinte forma:

```
1 tipo* nome;
```

7.3 Inicialização de apontadores

Tal como a declaração, a inicialização de um apontador é semelhante à das variáveis. Basta igualar um apontador ao endereço de uma outra variável. Para obter o endereço de outra variável basta colocar um "e" comercial antes do seu nome. Ora veja o seguinte exemplo:

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

int main() {
    int numero = 5;
    int* ponteiro =  &numero;

  // igual a: printf("%d e %p", numero, &numero);
    printf("%d e %p", numero, ponteiro);
    return 0;
}
```

No código acima, é declarada a variável `numero`, do tipo `int`, que é igual a 5. Seguidamente, é declarado um apontador denominado `ponteiro` e inicializado com o valor `&numero`, ou seja, com a posição da variável `numero`. Podemos dizer que "`ponteiro` aponta para `numero`". No final é imprimido o conteúdo da variável `numero` e o o apontador da mesma, armazenado na variável `ponteiro`.

Imagine agora que precisa efetuar a soma de duas variáveis utilizando apenas apontadores. Sim, este é um exemplo bizarro, mas que funciona.

> Se está a ter problemas com a codificação dos caracteres no Windows, inclua a biblioteca `Windows.h` e altere a página de codificação da linha de comandos como é feito na linha seis do exemplo seguinte.

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

int main() {
    SetConsoleOutputCP(65001);

    int a, b, c;
    int* p, q;

    a = 5;
    b = 4;

    p = &a;
    q = &b;
    c = *p + *q;

    printf("A soma de %d e %d é %d.", a, b, c);
    return 0;
}
```

No exemplo anterior são declaradas cinco variáveis. Três que contêm números inteiros (`a`, `b` e `c`) e duas que contêm endereços da memória RAM (`p` e `q`). Para efetuar a soma entre as variáveis `a` e `b` e armazenar a mesma na variável `c` utilizando apontadores, tem que se igualar os apontadores `p` e `q` ao endereço das variáveis `a` e `b`. De seguida, a variável `c` terá que ser igual à soma do conteúdo que para o qual os apontadores estão a apontar (`*p + *q`.

Imagine agora que precisa de um procedimento que troque os valores de duas variáveis. Inicialmente, poderia pensar em fazer algo deste género:

```c
void trocaDeValores( int x, int y) {
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Sintaticamente, nada está errado, mas o procedimento não vai realmente alterar os valores das variáveis. O que acontece no trecho de código anterior é que o procedimento recebe apenas o conteúdo de duas variáveis e coloca-os numas míseras variáveis locais, ou seja, aquelas que apenas estão disponíveis apenas dentro de uma função/procedimento. O procedimento correto a tomar

seria o seguinte:

```
void trocaDeValores(int* p, int* q) {
    int temp;

    temp = *p;
    *p = *q;
    *q = temp;
}
```

Esta função realmente troca os valores de duas variáveis. Enviam-se dois apontadores, dois endereços, e a variável troca os valores que estão em ambos os endereços da memória RAM, trocando os valores das variáveis.

7.4 Operações com apontadores

Uma das principais vantagens da utilização de apontadores é a fácil modificação de outras variáveis e que algumas linguagens, como a linguagem C, trabalham mais rapidamente com apontadores.

Tal como acontece com as restantes variáveis, podemos incrementar e decrementar apontadores, ou seja, incrementar ou decrementar o endereço que para o qual aponta o apontador.

Imagine que temos um *array* do tipo `double` com dois elementos.. Sabemos também que cada variável desse tipo ocupa, geralmente, 8 *bytes* na memória RAM. De seguida, é criado um apontador cujo endereço aponta para esse *array*. O que vai acontecer é que esse endereço apontará para o primeiro *byte* do *array* e não para todos eles.

```
double numeros[2]; // declaração do array com dois elementos.
double *apontador;

apontador = numeros;
```

Hipoteticamente falando, a variável `apontador` armazena o endereço 4550. Então, sabendo que o *array* ocupa, na totalidade, 16 *bytes* (visto que tem a capacidade para armazenar dois elementos do tipo *double*, que ocupam 8 *bytes* cada), podemos dizer que este *array* ocupa todos os *bytes* entre 4550 e 4565 (inclusive).

Se procedermos a uma incrementação do tipo `apontador++`, a variável `apontador` deixará de apontar para a primeira posição do *array*, passando logo para a segunda. Porquê? Porque quando incrementamos/decrementamos um apontador, não estamos a adicionar/diminuir meramente um *byte*, mas sim a quantidade que ocupa um elemento do tipo de dados que o *array* tem.

Imaginando os endereços de variáveis como números decimais (o que não são, visto serem hexadecimais), as duas linhas seguintes seriam equivalentes:

```
apontador++;
apontador + sizeof(double);
```

Como exemplo, tem abaixo uma progressão aritmética, que faz uso da incrementação de apontadores. Numa progressão aritmética cada termo é igual ao termo anterior somado com a razão, sendo esta última uma constante.

$n[y] = n[y-1] + r$

Poderíamos fazer este pequeno programa da seguinte forma (o código abaixo está explicado):

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

int main()
{
    int pa[10], razao;
    int *pointer;

    /*
     * Aqui pedímos o termo inicial, ou seja, o primeiro
     * número da PA (Progressão Aritmética).
     *
```

```
13       * De seguida o apontador é definido para apontar para
14       * o array pa.
15       */
16      printf("Insira o termo inicial: ");
17      scanf("%d", &pa[0]);
18      pointer = pa;
19
20      printf("Insira razão: ");
21      scanf("%d", &razao);
22
23      while(pointer != &pa[9]) {
24          /*
25           * Cada valor da PA é definido somando o valor corrente do apontador com a razão
    .
26           * De seguida, o apontador é incrementado de forma a apontar para o próximo
27           * elemento do array. (Isto acontece enquanto o apontador for diferente do
28           * endereço do último elemento da PA.)
29           */
30          *(pointer + 1) =  *pointer + razao;
31          pointer++;
32      }
33
34      printf("PA");
35
36      for(pointer = pa ; pointer <= &pa[9] ; pointer++) {
37          /*
38           * Aqui todos os elementos do array são percoridos
39           * e imprime-se assim a Progressão Aritmética criada.
40           */
41          printf(" → %d", *pointer);
42      }
43
44      return 0;
45  }
```

Capítulo 8

Strings

Na programação, os tipos de dados não se limitam aos já abordados no capítulo 2: `char`, `int`, `float`, `double`, etc. O tipo `char` permite armazenar um carácter. Mas, não é um carácter muito pouco? E se for necessário armazenar uma frase? Aí entram as *strings*.

> **Definição 32.** ***Strings*** são sequências de caracteres. Qualquer frase é considerada uma *string*, pois é uma sequência de caracteres.

Em C, todas as *strings* terminam com o carácter "\0", um delimitador ASCII para indicar o final da *string*.

8.1 Declaração e inicialização de *strings*

As *strings* podem conter, tal como as variáveis do tipo `char`, apenas um carácter. A diferença entre ambas prende-se com a utilização de aspas ou plicas. As aspas são utilizadas para delimitar *strings* e as plicas para delimitar caracteres. Visualize o seguinte exemplo:

```
1 "Programar" → String
2 "P" → String
3 'P' → Caracter
```

Em C, as *strings* são *arrays* de caracteres, ou seja, *arrays* do tipo `char`. Podem ser declaradas de diversas formas.

Uma forma de declarar *strings* em C, é criar um *array* do tipo `char` com um número de caracteres pré-definidos. Por exemplo:

```
1 // o mesmo que: char nome[8] = {'P', 'p', 'l', 'w', 'a', 'r', 'e', '\0'};
2 char nome[8] = "Pplware";
```

No exemplo anterior, é declarada a *string* `nome` que pode armazenar uma frase com 7 caracteres. Porquê 7 se foram declaradas 8 posições no *array*? Isto acontece porque o último carácter, o oitavo carácter, é o delimitador do final da frase `\0`.

Existem, no total, três formas de declarar *strings* em C:

- A primeira consiste na criação de um *array* com o tamanho pré-determinado;
- A segunda consiste na criação de um *array* sem especificar o seu comprimento, tendo que ser inicializada no momento da declaração de forma a que o espaço na memória seja alocado dependendo do tamanho da *string* colocada;
- Ou através de um apontador.

```
1 char nome[8] = "Pplware";
2 char nome[] = "Pplware"; // recomenda-se devido à legibilidade
3 char* nome = "Pplware";
```

Se reparar, este "tipo" de dados sempre foi utilizado. Na função *printf*, por exemplo, o primeiro argumento foi sempre uma *string*, pois é uma sequência de caracteres delimitada por aspas.

8.2 Como imprimir *strings*

As *strings* podem ser imprimidas recorrendo a diversas funções. Aqui são abordadas duas formas: recorrendo à função `printf` e recorrendo à função `puts`.

8.2.1 Com a função `printf`

Para imprimir uma *string* utilizando a função `printf`, basta utilizar o especificador `%s`. Por exemplo:

```
printf("Esta é uma string: \%s", nomeDaString);
```

A função `printf` é útil quando é necessário imprimir uma *string* que pode variar.

8.2.2 Com a função `puts`

Temos também a função `puts`, já abordada no capítulo 5, cujo nome quer dizer *put string*, ou seja, colocar *string*. Esta função é excelente para imprimir uma *string* que não esteja intercalada noutra *string*. Ora veja o seguinte exemplo:

```
char* nome = "José";
puts(nome);
```

8.3 Como ler *strings*

Quando é necessário um dado do utilizador como o nome, por exemplo, saber como se leem *strings* é importante. A leitura de *strings* pode ser feita de diversas formas.

8.3.1 Com a função `scanf`

A função `scanf` já foi falada diversas vezes ao longo deste livro. Tal como o que acontece com a função `printf`, deve-se utilizar o especificador `%s` para ler *strings*. Ora veja como se lê uma *string*:

```
scanf("%s", variavelParaArmazenarAString);
```

Analisando o excerto anterior é possível verificar que, ao contrário do que acontece com os restantes tipos de dados, neste não colocamos o "e" comercial no início do nome da variável que é utilizada para armazenar a *string*. Isto acontece porque as variáveis que contêm *strings* são, ou *arrays* ou apontadores, logo o seu nome já aponta para o endereço da memória.

Imagine agora que precisa do nome, apelido, morada e código postal de um utilizador para criar o seu cartão de identificação. Poderia proceder da seguinte forma:

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

int main() {
    char nome[21],
        apelido[21],
        morada[51],
        codigoPostal[11];

    printf("Por favor insira os seus dados conforme pedido:\n\n");
    printf("Primeiro nome: ");
    scanf("%s", nome);

    printf("Último nome: ");
    scanf("%s", apelido);

    printf("Morada: ");
    scanf("%s", morada);

```

```
20      // limpeza do buffer no Windows; usar "_fpurge(stdin)" em sistemas Unix
21      fflush(stdin);
22
23      printf("Código Postal: ");
24      scanf("%s", codigoPostal);
25
26      printf("\nO seu Cartão de Identificação:\n");
27      printf("Nome: %s, %s\n", apelido, nome);
28      printf("Morada: %s\n", morada);
29      printf("Código Postal: %s\n", codigoPostal);
30      return 0;
31  }
```

Relembro que a utilização de comandos para limpar o *buffer* não é recomendável e que devem ser utilizadas outras funções que não a `scanf` de forma a obter dados do utilizador sem "lixo".

8.3.2 Com a função `gets`

Podem-se imprimir *strings* com a função `gets`, cujo nome quer dizer *get string*, ou seja, obter *string*. A utilização desta função é simples. Ora veja como se utiliza esta função:

```
1  gets(nomeDaVariavel);
```

Onde `nomeDaVariavel` corresponde ao apontador que aponta para o local onde a *string* vai ser armazenada. Recordo que, no caso se ser utilizado um apontador ou um *array*, não é necessário utilizar um "e" comercial no início.

Imaginando agora que precisa criar um boletim de informação com diversos dados sobre o utilizador. Poderia fazer da seguinte forma:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      char nome[21],
6          apelido[21],
7          morada[51],
8          codigoPostal[11];
9
10     printf("Por favor insira os seus dados conforme pedido:\n\n");
11     printf("Primeiro nome: ");
12     gets(nome);
13
14     printf("Último nome: ");
15     gets(apelido);
16
17     printf("Morada: ");
18     gets(morada);
19
20     printf("Código Postal: ");
21     gets(codigoPostal);
22
23     printf("\nO seu Cartão de Identificação:\n");
24     printf("Nome: %s, %s\n", apelido, nome);
25     printf("Morada: %s\n", morada);
26     printf("Código Postal: %s\n", codigoPostal);
27     return 0;
28 }
```

Analisando o código é possível verificar que com esta função não é preciso limpar o *buffer* de forma a não obter caracteres indevidos. Isto acontece porque a função `gets` os ignora.

8.3.3 Com a função `fgets`

Tanto a função `gets` como a função `scanf` têm alguns contratempos; a primeira tem alguns problemas quando as *strings* incluem caracteres como espaços e a segunda obtém caracteres desnecessários. Devido à falta de uma solução efetiva a estes problemas, a função `fgets` poderá ser

a melhor opção.

A função `fgets` permite obter dados, não só do teclado, como de outros locais. Ora veja a sua sintaxe:

```
fgets(char *str, int n, FILE *stream);
```

Onde:

- `str` corresponde ao apontador para um *array* de caracteres onde os dados obtidos serão armazenados;
- `n` é o número máximo de caracteres a serem lidos (incluindo o delimitador final). Geralmente é igual ao tamanho do *arrar*;
- `stream` corresponde ao apontador para o ficheiro ou objeto donde serão lidos os dados.

Imaginando agora que é necessário converter o programa da criação do boletim de informação do utilizador para utilizar a função `fgets`. Ficaria da seguinte forma:

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

int main() {
    char nome[21],
        apelido[21],
        morada[51],
        codigoPostal[11];

    printf("Por favor insira os seus dados conforme pedido:\n\n");
    printf("Primeiro nome: ");
    fgets(nome, 21, stdin);

    printf("Último nome: ");
    fgets(apelido, 21, stdin);

    printf("Morada: ");
    fgets(morada, 51, stdin);

    printf("Código Postal: ");
    fgets(codigoPostal, 11, stdin);

    printf("\nO seu Cartão de Identificação:\n");
    printf("Nome: %s, %s\n", apelido, nome);
    printf("Morada: %s\n", morada);
    printf("Código Postal: %s\n", codigoPostal);
    return 0;
}
```

Se compilar e correr o código acima, irá receber algo semelhante ao seguinte:

```
O seu Cartão de Identificação:
Nome: Apelido
, Nome

Morada: Morada

Código Postal: CP
```

Estas mudanças de linha acontecem porque as *strings* obtidas através da função `fgets` ficaram com o carácter `\n` no final. Para remover este carácter pode-se recorrer à função `strtok`. Esta função utiliza-se da seguinte forma:

```
strtok(char *str, const char *delim);
```

Onde:

- `str` é o apontador para um *array* de caracteres onde a *string* está armazenada;
- `delim` corresponde ao delimitador a remover.

Assim, para que o carácter `\n` seja removido de todas as *strings* utilizadas no programa anterior, bastaria adicionar as seguintes linhas:

```
strtok(nome, "\n");
```

```
2 strtok(apelido, "\n");
3 strtok(morada, "\n");
4 strtok(codigoPostal, "\n");
```

Capítulo 9

Anexos

9.1 Anexo I - Tabela ASCII

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46
(bel)	7	0007	0x07	’	39	0047	0x27	G	71	0107	0x47
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f
`	96	0140	0x60	k	107	0153	0x6b	v	118	0166	0x76
a	97	0141	0x61	l	108	0154	0x6c	w	119	0167	0x77
b	98	0142	0x62	m	109	0155	0x6d	x	120	0170	0x78
c	99	0143	0x63	n	110	0156	0x6e	y	121	0171	0x79
d	100	0144	0x64	o	111	0157	0x6f	z	122	0172	0x7a
e	101	0145	0x65	p	112	0160	0x70	{	123	0173	0x7b

```
f     102 0146 0x66 | q     113 0161 0x71 | |     124 0174 0x7c |
g     103 0147 0x67 | r     114 0162 0x72 | }     125 0175 0x7d |
h     104 0150 0x68 | s     115 0163 0x73 | ~     126 0176 0x7e |
i     105 0151 0x69 | t     116 0164 0x74 | (del) 127 0177 0x7f |
j     106 0152 0x6a | u     117 0165 0x75 |-------------------- |
------------------------------------------------------------- |
```

9.2 Anexo II - Conversão Hexadecimal-Decimal

Hexadecimal	**Decimal**	**Hexadecimal**	**Decimal**
0	0	8	8
1	1	9	9
2	2	A	10
3	3	B	11
4	4	C	12
5	5	D	13
6	6	E	14
7	7	F	15

Tabela 9.1: Conversão hexadecimal para decimal