

Programação em C/C++ - Estrutura Básica e Conceitos Fundamentais

Sérgio Lopes

Programação em C/C++ - Estrutura Básica e Conceitos Fundamentais

Sérgio Lopes

Índice

Sobre o Manual	vi
1. Público Alvo	vi
2. Estrutura e Conteúdo	vi
3. Licença	vi
1. Algoritmia	1
2. Álgebra de Boole	4
2.1. Tabelas de Verdade	4
3. Características e História da Linguagem C	6
3.1. Processo de Desenvolvimento	7
3.2. Code::Blocks	8
3.2.1. Instalação	8
4. Estrutura dos Ficheiros	12
4.1. Lista de Palavras Reservadas	14
4.2. Função main	14
4.2.1. Parâmetros da Função main	14
4.3. Exemplo Completo da Sintaxe	15
5. Variáveis	17
5.1. Constantes	18
5.2. Directiva define	18
5.3. Declaração de Variáveis	19
6. Tipos de Dados Numéricos	20
7. Cadeias de Caracteres - String	21
7.1. Função strlen	21
8. Escrita Formatada	22
8.1. printf	22
8.1.1. Especificadores de Formatação	22
8.2. Caracteres de Escape	23
9. Leitura Formatada	24
9.1. scanf	24
9.2. Leitura para Endereços - Operador &	24
9.3. Caracteres de Escape	25
10. Outras Funções de Leitura e Escrita	26
Bibliografia	27

Lista de Figuras

1.1. Diagrama do Algoritmo para Ligar um Candeeiro	2
3.1. Do Código ao Executável	8
3.2. Detecção das Ferramentas de Compilação	9
3.3. Primeiro Ecrã	9
3.4. Associar Ficheiros de Código	10
3.5. IDE	10
3.6. Gravar Alterações da Interface	11
4.1. Divisão Completa	13
5.1. Estrutura de Memória e Relação com Variáveis	17
5.2. Declaração de uma Variável	19

Lista de Tabelas

2.1. Tabela de Verdade para OU	5
2.2. Tabela de Verdade para E	5
3.1. Características da Linguagem	6
6.1. Tabela de Tipos de Dados Numéricos	20
8.1. Especificadores de Formatação	23
8.2. Conjunto de Caracteres de Escape	23
9.1. Especificadores de Formatação de Leitura	24
9.2. Tabela com caracteres de controlo de leitura	25

Sobre o Manual

Público Alvo

Este manual foi criado, especificamente, para o módulo *0782 - Programação em C/C++ - Estrutura Básica e Conceitos Fundamentais*, ministrado em formações profissionais como indicado pelo catálogo da ANQ, www.catalogo.anq.gov.pt. Foi desenvolvido com uma elevada simplificação de conceitos e de com o objectivo de servir de base para a introdução à programação em linguagem C, sempre com forte apoio prático, e em conjunto com os módulos *0783 - Programação em C/C++ - Ciclos e Decisões*, *0784 - Programação em C/C++ - Funções e Estruturas* e *0785 - Programação em C/C++ - Formas Complexas*.

Estrutura e Conteúdo

Este manual aborda as áreas de algoritmia, lógica e introdução à linguagem de programação C, com foco na sintaxe base da linguagem e estrutura dos ficheiros de código. Faz uso e explica a instalação do IDE Code::Blocks como ferramenta de desenvolvimento.

No fim do manual, o formando deverá ter a capacidade para criar programas simples em linguagem de programação C, começando pela criação de algoritmos e passando à sua implementação usando apenas uma função principal, função *main*. Deverá ser capaz de compreender a declaração de variáveis e a sua manipulação através do uso de leitura e escrita formatada com as funções *printf* e *scanf*.

Licença

Esta obra é licenciada sob Creative Commons - Attribution-ShareAlike 3.0 Unported e poderá ser usada e partilhada segundo a mesma licença. Um resumo das obrigações pode ser consultada em <http://creativecommons.org/licenses/by-sa/3.0/> e o texto completo da licença está disponível em <http://creativecommons.org/licenses/by-sa/3.0/legalcode>.

Qualquer redistribuição da obra deverá manter a indicação do autor original, incluindo o endereço de e-mail.

Capítulo 1. Algoritmia

A definição de algoritmo tende a ser ligeiramente diferente de área para área, no caso da programação e de forma genérica, podemos considerar um algoritmo como **um conjunto finito de instruções, executadas de forma sequencial que permitem atingir um objectivo**. Esta definição simples e genérica permite, mesmo assim, encaixar no termo **algoritmo** um número vasto de acções que vão muito além da programação ou do desenvolvimento de software. Tomemos como exemplo a confecção de um bolo ou o processo de ligar um candeeiro:

1. Receita de um Bolo de Chocolate

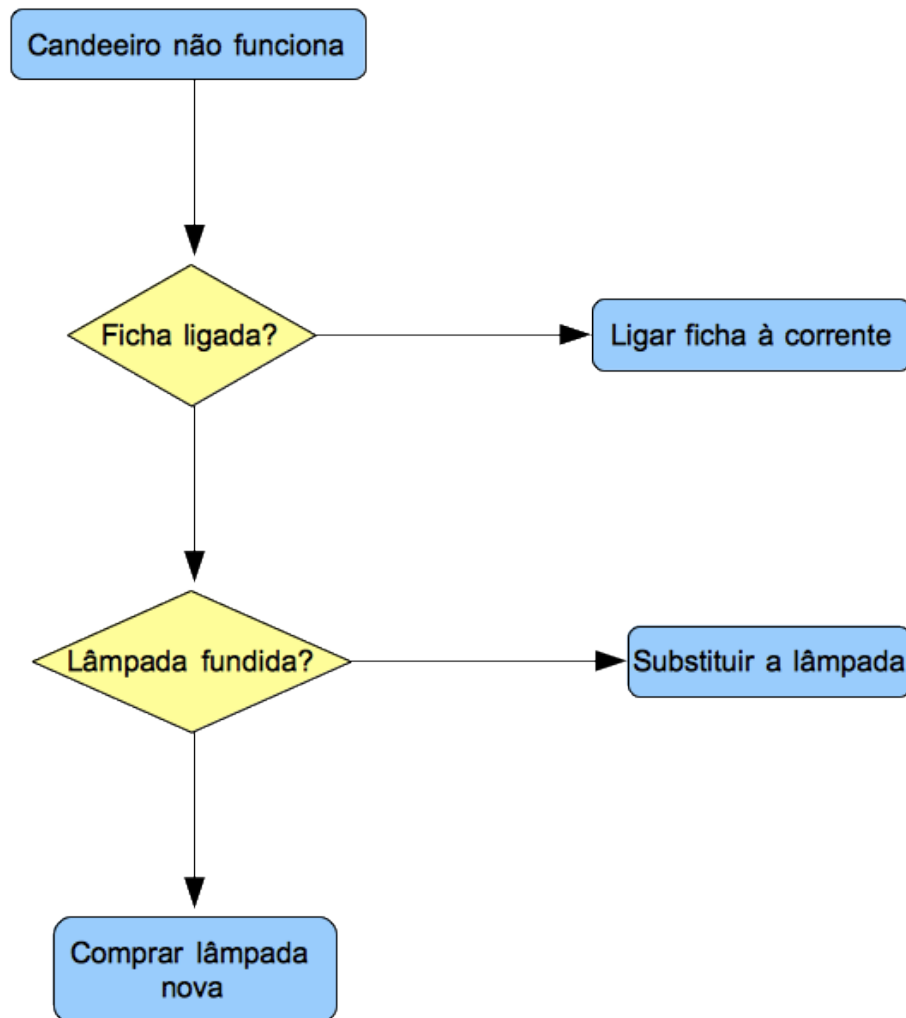
Ingredientes:

2 chávenas de farinha de trigo
2 chávenas de açúcar
1 chávenas de leite
6 colheres de sopa cheias de chocolate em pó
1 colher de sopa de fermento em pó
6 ovos

Processo:

Bater as claras em neve;
Acrescentar as gemas;
Bater novamente;
Adicionar o açúcar;
Bater outra vez;
Colocar a farinha, o chocolate em pó, o fermento, o leite;
Bater novamente;
Untar um tabuleiro;
Colocar no tabuleiro o preparado;
Assar durante 40 minutos em forno médio
Enquanto o bolo assa faça a cobertura com:
 2 colheres de chocolate em pó,
 1 colher de margarina,
 meio copo de leite;
Leve ao fogo até começar a ferver;
Jogue quente sobre o bolo já assado;
Como até rebentar!

Figura 1.1. Diagrama do Algoritmo para Ligar um Candeeiro



A criação de algoritmos é importantíssima para o bom desenvolvimento de programas de computador. Um algoritmo permite ao programador estruturar a resolução do problema de forma simples, oferecendo-lhe uma visão global dos passos a seguir para passar do algoritmo ao código e para obter um programa que resolva o problema em mãos.

Quando aplicado à programação, um algoritmo é escrito usando termos que o aproximam da linguagem de programação usada ¹. Assim, um algoritmo que descreve um programa de computador faz uso de termos que impliquem acções(fazer, definir, alterar, somar, etc.), que impliquem condições (se, então, enquanto, etc.) e que impliquem saltos para determinado ponto, alterando a sequência de execução (ir para, voltar a, saltar para, etc.).

Exemplo, obtenção do máximo divisor comum entre dois números:

```

1. Tendo x e y, dois números naturais (1, 2, 3, 4, 5, ...)
2. Se x < y
   Então:
       minimo = x
   Senão:
       minimo = y
3. mdc = minimo
  
```

¹Esta aproximação é muitas vezes chamada de pseudo-código.


```
4. restoA = resto da divisão entre x e mdc
5. restoB = resto da divisão entre y e mdc
6. Se restoA = 0 e restoB = 0
    Então:
        imprimir mdc
        terminar
    Senão:
        mdc = mdc - 1
        ir para 4
```

Capítulo 2. Álgebra de Boole

A **Álgebra de Boole** foi desenvolvida por George Boole em 1854 como uma variante da álgebra elementar. Boole criou um sistema algébrico cujas variáveis são lógicas em vez de numéricas e com operações e propriedades diferentes da álgebra numérica. Este tipo de álgebra serve de base à programação de software, sendo também bastante usada na linguagem do dia a dia sem isso seja notado.

A abordagem à álgebra de Boole será bastante superficial, centrando-se nas operações básicas de soma, multiplicação e negação e em algumas regras simples com recurso a quadros de verdade.

Em **álgebra de Boole** usamos apenas dois valores possíveis: **verdadeiro** e **falso**. São estes dois valores que permitem efectuar as operações de **soma**, também chamadas de **ou**, operações de multiplicação, também chamadas de **e**, e negações. Dado que só existem dois valores possíveis, **verdadeiro** e **falso**, qualquer operação que se faça resulta sempre num destes dois valores.

A **Soma booleana**, também representada por **ou**, \vee e $+$, dá um resultado verdadeiro quando **pelo menos uma das parcelas é verdadeira** e falso quando **todas as parcelas são falsas**.

Ex:

```
Fui ao cinema ou ao restaurante.  
É verdade que fui ao cinema, mas o restaurante estava fechado.  
Resultado: verdadeiro.  
  
Tenho mais de 20 anos e faço anos em Janeiro;  
É falso que tenha mais de 20 anos e é falso que faço anos em Janeiro.  
Resultado: falso.
```

A **Multiplicação booleana**, também representada por **e**, \wedge e $*$, dá um resultado verdadeiro quando **todas as parcelas são verdadeiras** e falso quando **pelo menos uma das parcelas for falsa**.

Ex:

```
Fui ao cinema e ao restaurante.  
É verdade que fui ao cinema, mas o restaurante estava fechado.  
Resultado: falso.  
  
Tenho mais de 20 anos e faço anos em Janeiro;  
É verdade que tenho mais de 20 anos e é verdade que faço anos em Janeiro.  
Resultado: verdadeiro.
```

A **Negação booleana**, também representada por **não**, \sim , \wedge , e \neg , altera o valor da expressão ou afirmação para o seu valor complementar. De notar que só existem dois possíveis, pelo que um valor verdadeiro passa a falso e um falso passa a verdadeiro.

Tabelas de Verdade

As tabelas de verdade permitem consultar, rapidamente, o resultado de uma operação *booleana* entre duas parcelas.

Tabela 2.1. Tabela de Verdade para OU

Parcela 1	Parcela 2	Resultado
F	F	F
F	V	V
V	F	V
V	V	V

Tabela 2.2. Tabela de Verdade para E

Parcela 1	Parcela 2	Resultado
F	F	F
F	V	V
V	F	V
V	V	V

Capítulo 3. Características e História da Linguagem C

C é uma linguagem de programação de propósito genérico, criada para o desenvolvimento do sistema operativo Unix em 1972, mas que rapidamente ganhou popularidade devido à sua versatilidade e à divulgação que teve entre as universidades dos Estados Unidos da América.

A forte adopção da linguagem por parte dos programadores levou a que fosse criada uma comissão pelo Instituto de Padrões Nacionais Americanos (American National Standards Institute, ou ANSI) cujo objectivo seria desenvolver um padrão que pudesse ser usado por qualquer entidade e que permitisse que a forma de escrever código C e as funcionalidades e características disponíveis fossem as mesmas para todos os programadores.

Desta comissão nasceu o padrão comumente chamado de C89, que mais tarde foi também adoptado pela Organização Internacional de Padronização (International Organization for Standardization, ou OSI) formando o que é comumente designado C90. C89 e C90 designam o mesmo padrão, devido às diferentes datas de publicação, 1989 e 1990 respectivamente, têm um nome diferente.

Como linguagem genérica, C permite a criação de software para qualquer finalidade, estando disponível para, virtualmente, todos os tipos de dispositivos, desde computadores pessoais a telemóveis, calculadores ou frigoríficos. No entanto, apesar da sua versatilidade e capacidade, actualmente outras linguagens tomaram o lugar do C no desenvolvimento de software para computadores pessoais, ficando este com no papel de linguagem de desenvolvimento de sistemas operativos ou no desenvolvimento de software específico para dispositivos industriais e micro-processadores.

Tabela 3.1. Características da Linguagem

Rápida	A rapidez dos programas bem desenvolvidos em C é extremamente elevada, quase tão boa como a de programas desenvolvidos em Assembly ^a .
Simple	A sintaxe da linguagem é relativamente simples ^b , com um número de palavras reservadas reduzido e com uma forma de apresentar o código muito próxima de como os algoritmos são desenhadas. A maior falha da sintaxe de C é a sua verbosidade, obrigando o programador a escrever caracteres terminadores de linhas, ;, e caracteres separadores de blocos de código, }.
Modular	C permite o desenvolvimento de programas de forma modular, usando funções, facilitando integração com outros programas ou bibliotecas e oferecendo mecanismos para divisão do código como divisão por ficheiros.
Portável ^c	A portabilidade da linguagem C é bastante elevada se for usado apenas o padrão C (comumente chamado de ANSI C, C89 ou C90). Se apenas forem usadas funções e bibliotecas que estejam definidas no padrão, o código pode ser executado em computador sem sofrer alterações.
Bibliotecas Base	O C vem com um conjunto de bibliotecas base bastante completo e que oferece ao programador um vasto leque de ferramentas para o desenvolvimento de programas. Bibliotecas para leitura de dados introduzidos pelo utilizador, para leitura e escrita de ficheiros, manipulação de memória, etc., são fornecidas ao programador como base a partir da qual ele pode construir novos programas.

^aAssembly é uma linguagem de baixo nível, intimamente ligada ao hardware, e é, entre as linguagens de programação, a que permite maior proximidade com o hardware onde o programa executa.

^bQuando comparada com linguagens da mesma geração.

^cA portabilidade define a capacidade de um programa poder ser executado sem alterações em equipamentos com hardware diferente, principalmente com arquitecturas de CPU diferentes.

Processo de Desenvolvimento

C é uma linguagem de programação compilada, quer isto dizer que o nosso código fonte é transformado directamente para linguagem máquina, a partir da qual é executado pelo sistema operativo. Assim, depois de compilado, o código que escrevemos dá origem a um ficheiro executável que pode ser usado em qualquer computador com sistema operativo igual.

Aquilo a que chamamos vulgarmente *programa* não é mais que o resultado do processo de compilação aplicado ao nosso código fonte.

Em C, estão envolvidos três passos no processo que leva ao produto final, são eles a **escrita de código**, a **compilação do código** e a **linkagem** ¹. O primeiro passo diz respeito à escrita do código, parte que cabe ao programador, os restantes passos são efectuados depois de escrito o código e permitem converter o código para o produto final.

Durante a fase de compilação o compilador analisa o nosso código, detecta erros de sintaxe e em alguns casos efectua optimizações, expande todas as macros e todas as linhas de pré-processamento ². Depois de executar, o compilador cria um ficheiro **objecto** que irá ser usado pela ferramenta de *linkagem*.

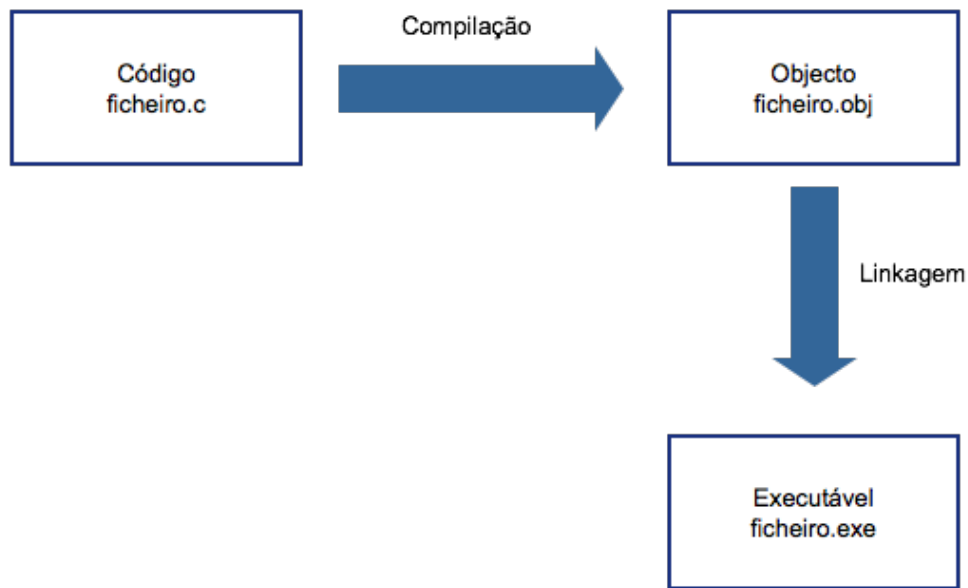
O *linker*, ferramenta que efectua o processo de *linkagem*, pega no ficheiro **objecto** e adiciona-lhe informação específica do sistema operativo em que a aplicação vai ser executada e que permite o arranque do nosso programa. Depois deste passo o resultado é um ficheiro executável que podemos correr nos nossos computadores.

Embora estes três passos possam ser feitos de forma independente, a maioria das ferramentas usadas actualmente inclui o passo de compilar e de *linkar* num processo automático.

¹O termo Inglês *linkage* não tem uma tradução para português, é comum usar a transformação *linkar* ou *linkagem*.

²Estes termos serão explicados nos capítulos seguintes

Figura 3.1. Do Código ao Executável



Code::Blocks

Um **IDE**, do inglês **I**ntegrated **D**evelopment **E**nvironment, ou como é chamado em português, **Ambiente de Desenvolvimento Integrado**, é uma ferramenta que agrupa no mesmo local um conjunto variado de outras ferramentas, necessárias ao desenvolvimento de programas de computador. Estas ferramentas podem ser usadas de forma individual mas obrigam o programador a dispendar mais tempo com tarefas paralelas à programação. Torna-se também útil reunir todas as ferramentas necessárias num único programa de modo a agilizar o processo de desenvolvimento, evitando ter de executar os passos de compilação e linkagem de forma individual, e podemos simplesmente usar a opção do IDE que nos fornece o resultado final: o nosso programa pronto a testar.

Existem inúmeros IDEs diferentes, cada um com as suas virtudes e defeitos, e que podem ser usados para programar em C. No entanto, a escolha para este manual recaiu sobre o IDE Code::Blocks, um editor disponibilizado como software livre e gratuito, podendo ser usado sem restrições ou com necessidade de pagamento, disponível para vários sistemas operativos e que oferece um bom equilíbrio entre funcionalidades e facilidade de utilização.

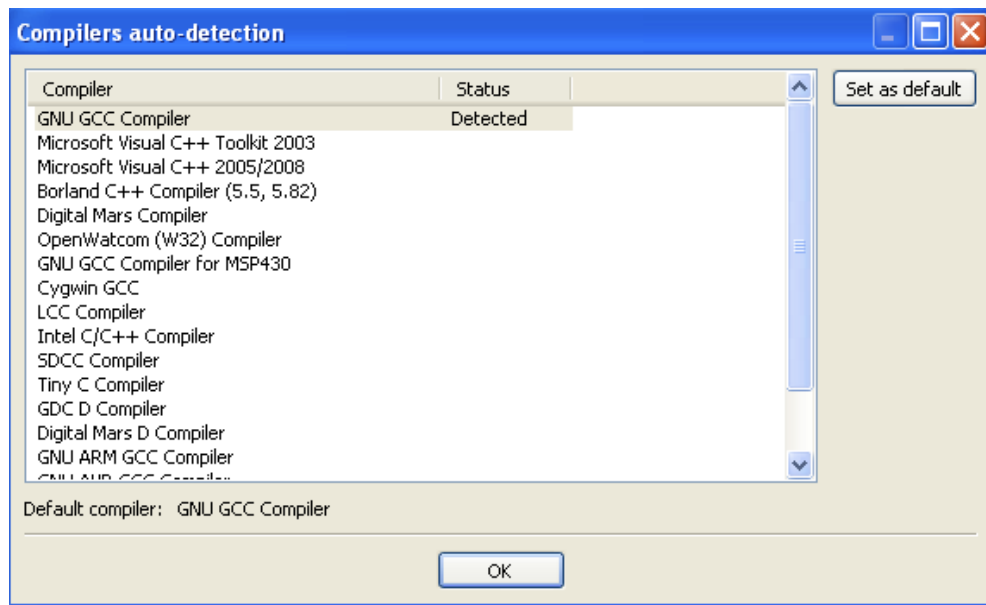
O IDE pode ser obtido em <http://www.codeblocks.org>.

Instalação

A instalação deste IDE é bastante simples, podendo ser mantidas todas as opções que estão preenchidas por omissão.

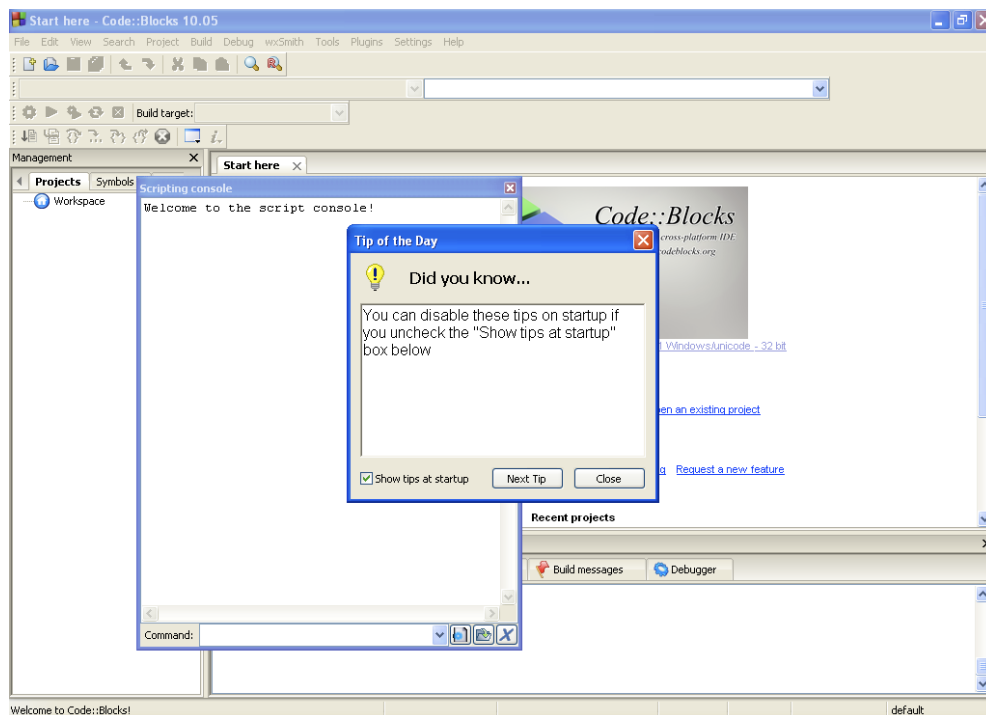
No final da instalação é-nos pedido que indiquemos quais as ferramentas de compilação pretendemos usar, isto porque o Code::Blocks é um IDE que consegue trabalhar com várias ferramentas diferentes. No nosso caso, é seguro escolher a opção que foi detectada automaticamente na instalação.

Figura 3.2. Detecção das Ferramentas de Compilação



Depois de instalado, o primeiro ecrã deverá mostrar algo parecido com a imagem seguinte.

Figura 3.3. Primeiro Ecrã

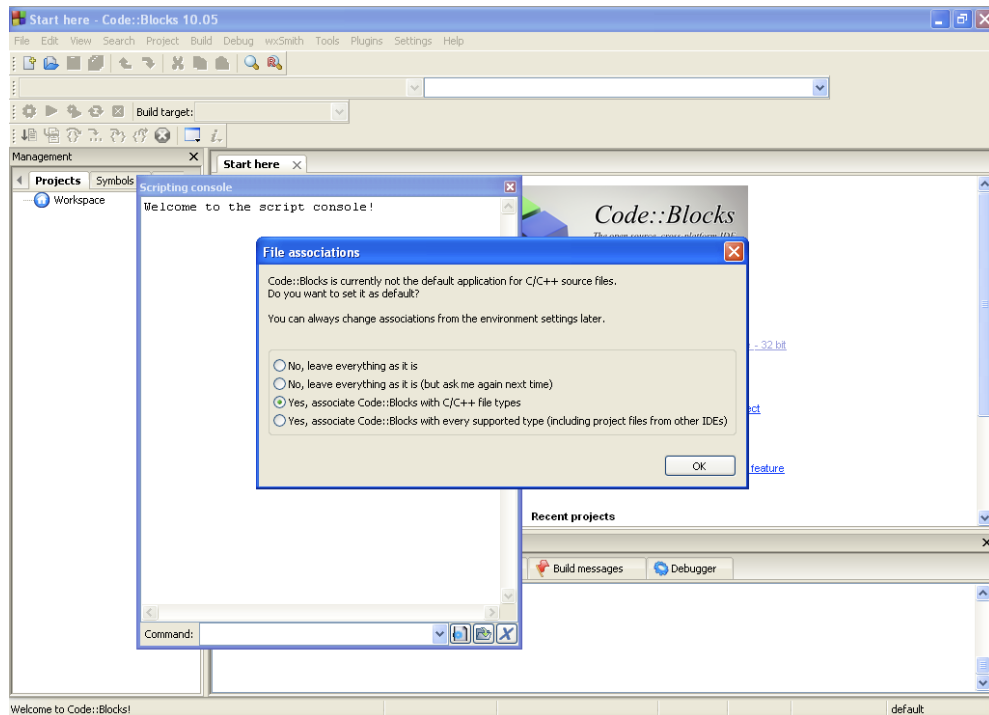


Podemos remover estas janelas e marcar as opções que possibilitam que as janelas não voltem a surgir em sessões posteriores. Podemos também remover algumas das barras de ferramentas que estão presentes no IDE de forma a simplificar a interface e a termos disponível o essencial para programarmos mas sem complicar com a presença de ferramentas que não usamos inicialmente.

Depois de fecharmos algumas das janelas iniciais somos confrontados com uma mensagem que nos pergunta se queremos associar os ficheiros do tipo C ao IDE que acabámos de instalar. Se este for o

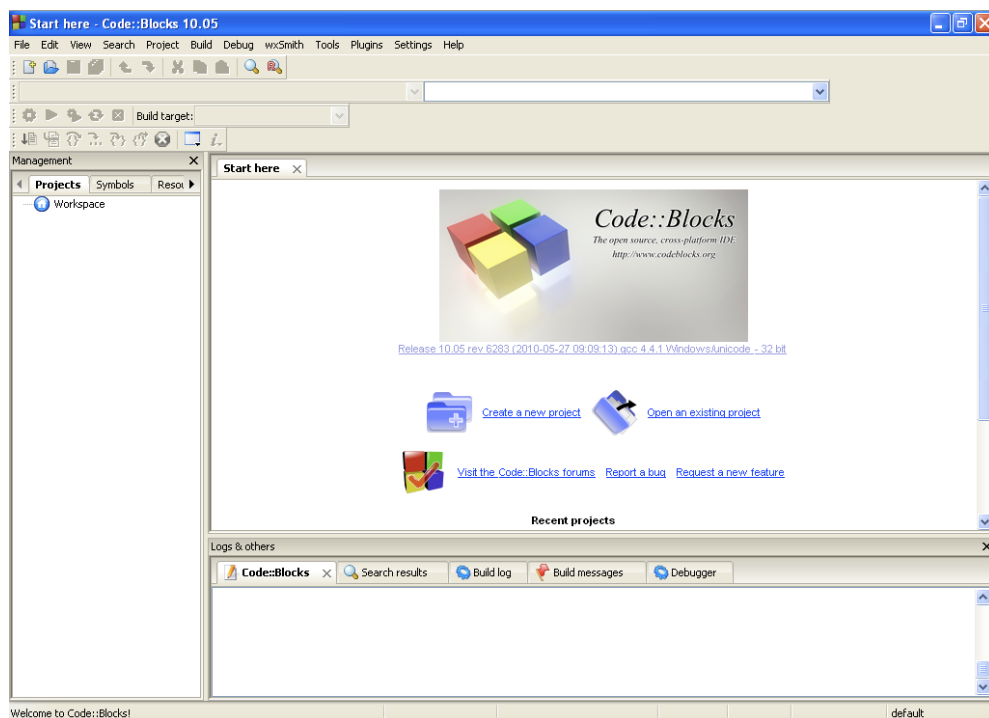
único IDE no sistema ou, existindo outro este é o que iremos usar, então podemos registar a associação para que sempre que tentarmos abrir um ficheiro com código C, o IDE seja iniciado automaticamente.

Figura 3.4. Associar Ficheiros de Código



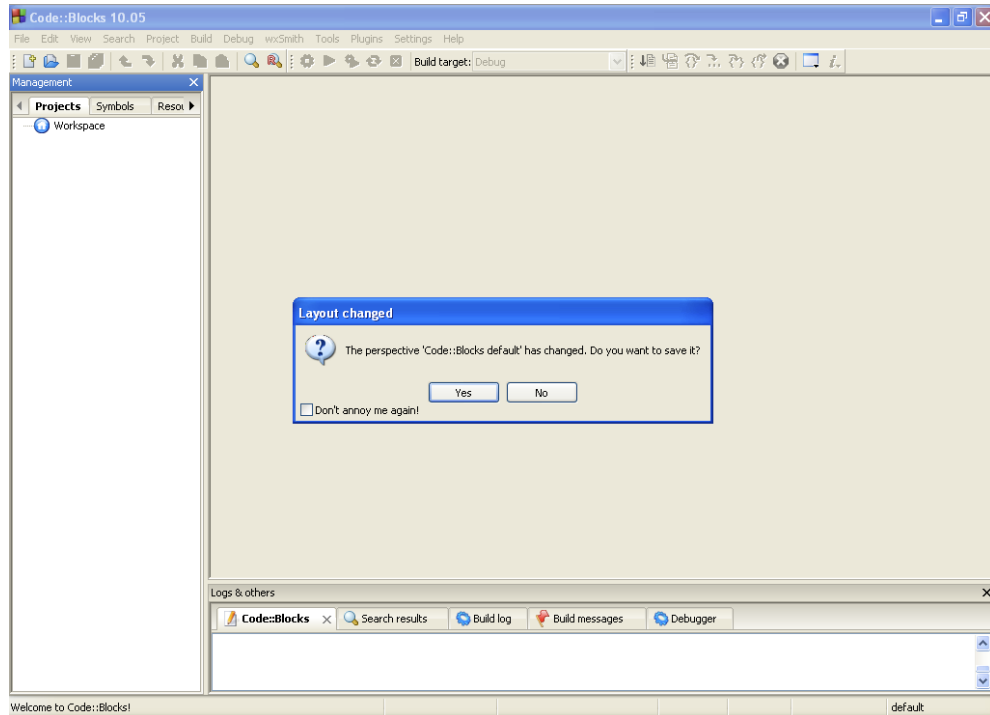
Por fim temos acesso ao nosso IDE para podermos começar a trabalhar.

Figura 3.5. IDE



Quando fecharmos o IDE, e dado que alterámos a interface inicial, ser-nos-á perguntado se pretendemos gravar as alterações que fizemos. Naturalmente as alterações são para se manter, pelo que as iremos gravar.

Figura 3.6. Gravar Alterações da Interface



Capítulo 4. Estrutura dos Ficheiros

Genericamente, um ficheiro de código C não tem regras para a localização dos vários elementos, desde que o código no ficheiro respeite a sintaxe da linguagem, o número de espaços entre os elementos, ou o número de linhas em branco existentes não afecta a correcta compilação do código.

Um ficheiro de código C pode ser dividido em duas partes genéricas: zona de directivas de pré-processamento e zona de implementação ¹. A zona de directivas corresponde ao início do ficheiro e é onde colocamos algumas directivas de pré-processamento. Neste capítulo ainda não falámos destas directivas mas podemos dizer que são precedidas por um cardinal, #, e fornecem instruções ao compilador.

Exemplo do conteúdo de um ficheiro de código C.

```
//Secção 1
#include <stdio.h>
#define IVA 0.21
//Fim da secção 1

//Secção 2
int main(void)
{
    int x = 0, y, z = 5;
    char opcao, texto = "Uma grande raposa azul!", palavra[50];
    float l, d = 0.0;

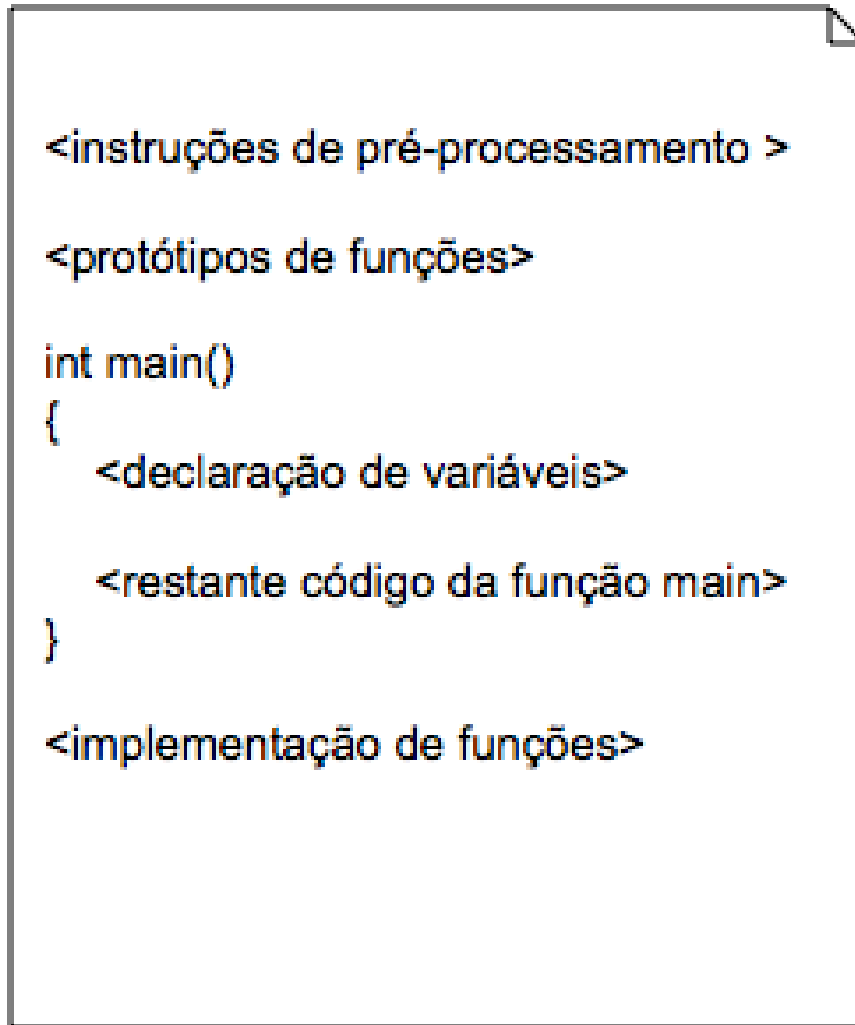
    printf("Olá utilizador :) ");

    return 0;
}
//Fim da secção 2
```

1. Duas directivas de pré-processamento.
2. Início da zona de implementação

Uma divisão diferente pode ser vista na imagem seguinte, onde se consideram mais secções. O importante a reter é que a organização e divisão feita é apenas uma organização ou divisão lógica e não afectar, geralmente, a compilação de código nem a forma como o programa executa.

¹Mais tarde estas duas zonas serão divididas, mas para já podemos considerar apenas duas.

Figura 4.1. Divisão CompletaA diagram showing the structure of a C program. It is enclosed in a rectangular box with a folded top-right corner. The text inside the box is as follows:

```
<instruções de pré-processamento >

<protótipos de funções>

int main()
{
    <declaração de variáveis>

    <restante código da função main>
}

<implementação de funções>
```

Seguindo as regras da linguagem C, a sua sintaxe, os nomes de variáveis (que veremos a seguir) e de funções (que serão explicadas em módulos seguintes) são dependentes de maiúsculas e de minúsculas, diz-se que o C é uma linguagem de programação **case-sensitive**. Assim, escrever "**main**", "**Main**" ou "**MAIN**" implica que nos estamos a referir a 3 coisas diferentes, além disso, todas as palavras especiais da linguagem são escritas em minúsculas. Como a matéria de funções não é abordada neste módulo podemos ficar apenas com a noção de que todos os programas em C são compostos por, pelo menos, uma função. A única função que precisa estar presente num programa C é a função **main** e dentro desta função podem ser chamadas muitas outras.

Todos os blocos de código são escritos entre chavetas, { e }, e para a escrita de comentários podemos usar o par // quando os comentários ocupam várias linhas, ou os dois caracteres seguidos // quando os comentários são colocados apenas numa linha.

Lista de Palavras Reservadas

As palavras reservadas são um conjunto de palavras com significado especial para a linguagem de programação. A seguinte tabela lista toda as palavras que são reservadas à linguagem de programação C.

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

Função main

A função mais importante para os nossos programas é a função **main**. Esta função é a única função que é obrigatório usar², uma vez que é a função usada pelo sistema operativo para iniciar o programa. Quando executamos o nosso programa, seja na linha de comandos seja em através de duplo clique, o sistema operativo tenta encontrar a função **main** e inicia o programa a partir dessa função, assim sendo esta é a função fundamental para que os nossos programas sejam executados.

A função **main** pode ser escrita com algumas variações:

```
int main() {...}
int main(void) {...}
int main(int argc, char *argv[]) {...}
```

Estas três formas são descritas no *standard* da linguagem, e podem ser usadas consoante a escolha do programador. É importante que a função **main** devolva um valor inteiro, tipicamente zero, para que o sistema operativo saiba que o programa terminou e em que estado isso aconteceu³.

Uma forma alternativa, que pode ser vista em vários livros da área e documentação referente à linguagem é a forma em que a função não devolve um valor inteiro, no entanto esta forma não está de acordo com o *standard*, e não deve ser seguida.

Exemplo de forma que não deve ser usada.

```
void main() {...}
```

Parâmetros da Função main

O *standard* de C define que a função **main** recebe dois parâmetros, *argc* e *argv* respectivamente, em que o primeiro indica o número de parâmetros passados durante o arranque da aplicação e o segundo indica quais são esses parâmetros. Um programa executado em linha de comandos terá no mínimo um parâmetro correspondente ao nome do programa, e no máximo tantos parâmetros como os que foram indicados na linha de comandos.

Quando declarada para receber parâmetros, de acordo com o *standard*, os programas podem aceder ao número de parâmetros passados e aos parâmetros através das variáveis *argc* e *argv*.

²Caso estejamos a desenvolver bibliotecas de funções, a função **main** não é necessária.

³Se o programa terminar com um erro deve devolver um valor diferente de zero. Zero indica uma execução terminada sem problemas.

Exemplos de execuções de programas e conteúdo dos parâmetros:

```
#> ping www.google.com
```

Conteúdo dos parâmetros

```
argv => 2
argc => ping
      www.google.com
```

```
#> netstat --all
```

Conteúdo dos parâmetros

```
argv => 2
argc => netstat
      --all
```

Exemplo Completo da Sintaxe

```
/* olamundo.c
 * 2010 © Sérgio Lopes
 *
 * Programa que exemplifica o uso de toda a sintaxe da linguagem C.
 *
 * Este exemplo serve para exemplificar o uso da linguagem, não pretende
 * explicar como usar alguns dos conceitos, apenas permitir uma
 * familiarização com a sintaxe.
 *
 * Este primeiro comentário de bloco é comum ser usado para explicar o
 * que o programa faz, para adicionar informações de autoria e
 * para adicionar licença de software usada.
 */

//algumas directivas de pré-processamento
#include <stdio.h>

#define IVA 21

//declaração de estruturas
typedef struct
{
    int    x;
    float y;
} t_ponto;

//declaração de uma função, o corpo da função, com o código
//estará presente depois da função main
void escrever(char *);

int main(int argc, char *argv[])
{
    //declaração de variáveis
    int contador, max = 20;
    t_ponto posicao;

    //definir o valor dos campos internos da estrutura
    posicao.x = 0;
    posicao.y = 0;

    //ciclo for
    for(contador = 0; contador < max; contador++)
    {
        //estrutura de controlo if... else
        if(contador % 2 == 0)
        {
            escrever("Contador é par");
        }
    }
}
```

```

    }
    else
    {
        //estrutura de controlo switch
        switch(contador)
        {
            //todos estes case irão passar ao seguinte, excepto o 19
            case 1:
            case 3:
            case 5:
            case 7:
            case 11:
            case 13:
            case 17:
            case 19:
                escrever("Contador é número primo!");
                break;
            default:
                escrever("Contador é impar mas não é número primo.");
        }
    }
}

//ciclo while sem corpo, note-se o ; no fim
while(--contador > 0);

printf("Rebobinamos o contador..., vamos terminar.\n");
printf("Olá mundo, escrevemos o primeiro programa!");
}

/**
 * Este bloco deve conter a descrição da função.
 * Uma vez que é um comentário de bloco pode ocupar várias linhas
 * e todos os caracteres que estejam dentro dos delimitadores são
 * considerados parte do comentário.
 *
 * Podemos assim usar vários asteriscos e outros caracteres que tornam
 * a leitura destes blocos mais simples e até permitem a formatação usando
 * ferramentas próprias.
 */
void escrever(char *texto)
{
    printf(texto);
    printf("\n");
}

```

Capítulo 5. Variáveis

Os nossos programas têm à sua disposição um conjunto variado de tipos de dados que podem manipular. São estes tipos de dados que tornam possível a criação de programas que recebam informação do utilizador, que a processem e que mostrem os vários resultados de volta ao utilizador.

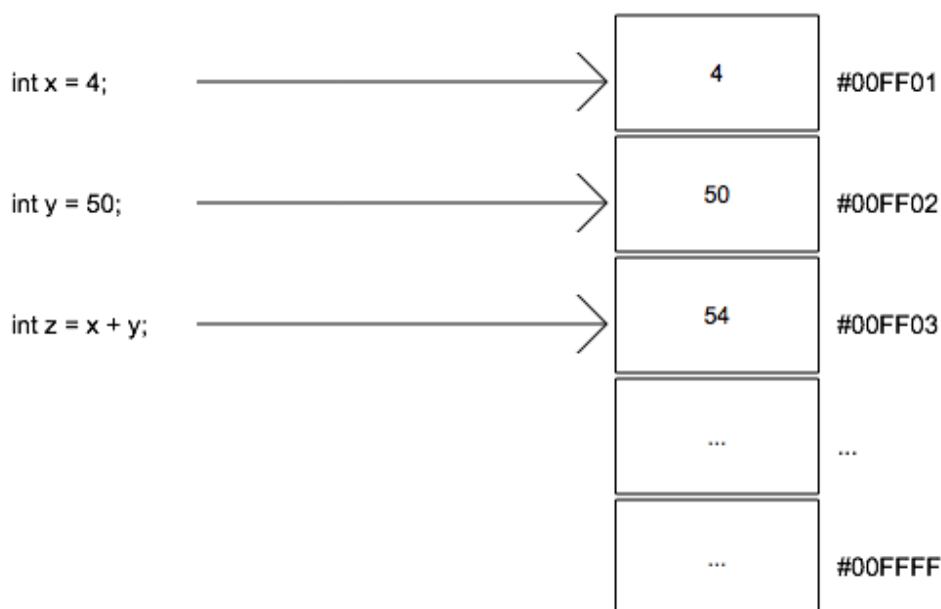
Em C existem os seguintes tipos de dados: * Numéricos, a que correspondem todos os números inteiros ou reais, sejam positivos ou negativos. * Cadeias de caracteres, a que chamamos *Strings* e que representam conjuntos de caracteres (números, letras, sinais de pontuação, etc) que podem ser usados e manipulados como texto.

Exemplos: . 5 é um número inteiro positivo; . -6.54 é um número real negativo; . "Uma grande raposa azul!" é uma cadeia de caracteres ou *String*;

Os tipos de dados são importantes porque afectam as variáveis que usamos no nosso código, e por **variáveis** referimo-nos a **elementos que usamos para guardar informação e que nos permitem manipular essa informação**, quer fazendo contas quer formatando o seu aspecto aquando da impressão no ecrã. **São objectos nos quais guardamos dados.**

Todos os programas de computador têm acesso a uma zona da memória disponível para colocarem informação, é nessa zona que as variáveis são criadas, e cada variável tem o seu espaço próprio, identificado por um endereço. Como esses endereços são números hexadecimais, e além de ser complicado gerir não temos acesso fácil aos endereços, o que fazemos é dar um nome a cada variável através do qual podemos referir-nos a esse espaço de memória.

Figura 5.1. Estrutura de Memória e Relação com Variáveis



No diagrama anterior podemos ver 3 variáveis de nomes **x**, **y** e **z**, bem como uma representação de como as mesmas podem estar guardadas em memória. Como indicado pelo diagrama uma variável não é mais que um nome definido pelo programador para se referir ao endereço de uma zona de memória, nome esse que durante a execução do programa é convertido para o endereço correspondente.

Constantes

Constantes são elementos similares a variáveis mas cujo valor não pode ser alterado **depois** de ter sido definido a **primeira** vez. Isto é, têm as mesmas regras que as restantes variáveis mas só lhes conseguimos colocar um valor e depois de colocado nunca o podemos alterar.

Este tipo de elemento é útil para situações onde precisamos garantir que não alteramos, por engano, o valor da variável, ou para qualquer situação onde nos é importante ter um valor que seja constante ao longo do programa.

Directiva define

Em capítulos avançados iremos aprender a usar as directivas de pré-processamento, como é o caso do **define**, no entanto esta directiva é bastante usada quando se fala de constantes, é assim relevante mencionarmos apenas esta directiva sem entrarmos em detalhes sobre o que são e como funcionam as directivas de pré-processamento.

A instrução **define**, escrita no topo dos ficheiros de código, permite associar a um determinado valor um nome, criando assim uma *macro*. Esse nome é depois usado pelo programador ao longo do código e quando o compilador executar as suas tarefas vai substituir todas as ocorrências do nome que encontrar pelo valor que foi associado.

```
#define IVA 0.21 //1
#define NUM_DIAS 365 //2

int main()
{
    printf("Percentagem de IVA: %f", IVA); //3
    printf("Número de dias do ano: %d", NUM_DIAS); //4

    return 0;
}
```

1. Associar o valor 0.21 ao nome IVA
2. Associar o valor 365 ao nome NUM_DIAS
3. Usar o nome IVA que irá ser substituído durante a compilação
4. Usar o nome NUM_DIAS que irá ser substituído durante a compilação

Esta directiva permite-nos assim ter um nome associado a um valor, uma *macro*. No caso do valor ser alterado, apenas é necessário ir modificar a linha onde fizemos a associação, e a partir desse ponto todos os locais onde usámos a *macro* serão actualizados automaticamente (aquando da próxima compilação).

Embora possam surgir bibliografias onde este tipo de instrução é indicado como uma constante, a verdade é que estas *macros* não são variáveis nem constantes. Ao contrário das variáveis não possuem espaço de memória, não são mantidas depois da compilação e não podem ser usadas como as restantes variáveis. Na verdade, estas *macros* são apenas texto que é substituído automaticamente pelo compilador antes de compilar o código.

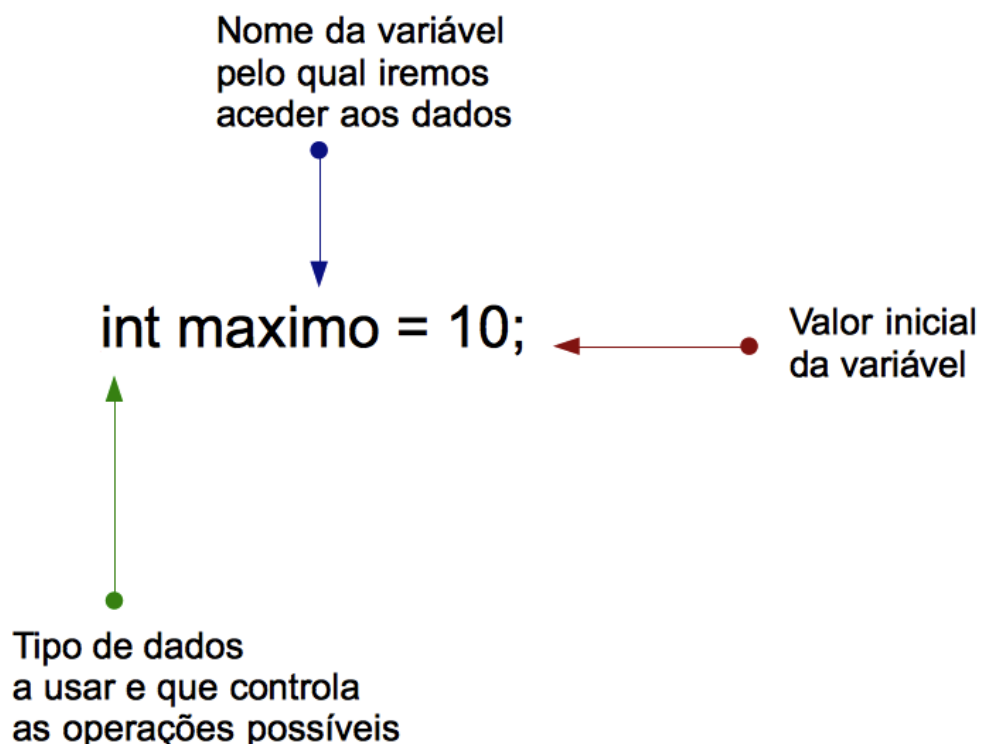
A vantagem deste tipo de *macros* é que definimos o valor num único local e se mais tarde precisarmos de alterar esse valor, apenas o fazemos uma vez e não somos forçados a percorrer todo o código à procura de locais onde o valor esteja a ser usado.

Declaração de Variáveis

A declaração de variáveis é o processo que nos permite dizer ao compilador quais são, e de que tipo são, as variáveis que pretendemos usar no nosso programa. Se não dissermos ao compilador quais as variáveis que usamos e de que tipo são, então o compilador não é capaz de compilar o nosso código. Como regra devemos lembrar-nos sempre que: **O compilador precisa saber como se chamam e de que tipo são as variáveis que usamos nos nossos programas, antes de as tentarmos usar.**

Quando declaramos uma variável é necessário indicar de que tipo é a variável, que nome tem, e opcionalmente podemos definir o valor inicial da variável.

Figura 5.2. Declaração de uma Variável



Seguindo o exemplo da figura, podemos declarar as variáveis de 3 modos diferentes:

```
int x;  
int k = 10;  
int h = 10, z = 9, y;
```

A primeira linha declara uma variável chamada **x** sem lhe colocar qualquer valor inicial, a segunda linha declara uma variável chamada **k** e define o valor com que a mesma começa, por fim, a terceira linha declara três variáveis em conjunto e define o valor inicial das duas primeiras, a **h** e a **z**, mas não define o valor da última, a variável **y**.

Capítulo 6. Tipos de Dados Numéricos

Dentro do tipo de dados numérico existem alguns sub-tipos que permitem controlar com maior precisão os dados que são guardados. É importante que a escolha do tipo de dados tenha em consideração qual o objectivo da variável e que dados vão ser usados, para que os resultados obtidos seja os esperados e que não se percam informações por arredondamentos indesejados.

A tabela apresentada abaixo mostra os tipos de dados numéricos que estão disponíveis para o desenvolvimento de programas em linguagem C. O tamanho exacto destes tipos de dados depende do processador em que o programa é compilado.

Tabela 6.1. Tabela de Tipos de Dados Numéricos

Tipo a Usar	Descrição	Número de bit	Consegue guardar
char	Carácter com sinal	8	-127 a 127
unsigned char	Carácter sem sinal	8	0 a 255
short	Inteiro pequeno	16	-128 a 127
unsigned short	Inteiro pequeno sem sinal	16	0 a 255
int	Inteiro	16 ou 32	-32768 a 32767
unsigned int	Inteiro sem sinal	16 ou 32	0 a 65535
long	Inteiro longo	32 ou 64	-2147483648 a 2147483647
unsigned long	Inteiro longo sem sinal	32 ou 64	0 a 4294967295
float	Valor real de precisão simples	<i>indefindo</i>	<i>indefindo</i>
double	Valor real de precisão dupla	<i>indefindo</i>	<i>indefindo</i>

Capítulo 7. Cadeias de Caracteres - String

Strings é o nome dado a um conjunto de caracteres que, tipicamente, permitem representar palavras e frases. Em C *Strings* são variáveis de um tipo especial chamado vector e que é representado pelo uso de parêntesis rectos junto ao nome da variável, ex: **char palavra[50];**.

Como se pode ver pelo exemplo, *Strings* são do tipo **char**, portanto caracteres, mas em vez de se conseguir guardar apenas um carácter de cada vez, é possível guardar um conjunto deles, colocados de forma sequencial, e assim criar palavras. Uma *String* é apenas um conjunto de caracteres com o pormenor de ter sempre um carácter terminador especial, o **\0**. Este terminador é composto por uma barra descendente, \, seguida e um zero e deve ser colocado entre plicas, ex: **'\0'**.

Assim, para que possamos dizer que estamos a usar uma *String* temos de ter declarado um vector de caracteres, usando os parênteses rectos na declaração, e de ter colocado no fim do texto que pretendemos guarda um **\0**. Este terminador é importante dado que muitas funções e funcionalidades disponíveis esperam que exista um terminador para que saibam onde termina o texto.

Todo o conceito de vectores será explicado em módulos seguintes, para já o importante é reter que uma *String* é **um tipo de dados que permite guardar um conjunto de caracteres, colocados de forma sequencial, e terminados com o carácter especial \0**.

Função strlen

strlen é uma função que permite determinar o tamanho de uma *String*. Esta função recebe a *String* que para a qual pretendemos determinar o tamanho e devolve um valor inteiro que indica o número de caracteres que constituem a *String*.

É necessário ter atenção que a função recebe um vector de caracteres e se não tivermos o cuidado de colocar o terminador das *Strings* no vector que estamos a passar à função então os resultados serão inesperados, e provavelmente o nosso programa irá ser terminado de forma anormal pelo sistema operativo.

Capítulo 8. Escrita Formatada

Escrita formata refere-se ao processo que permite a formatação dos nossos dados quando os escrevemos no ecrã, num ficheiro de texto, numa impressora, ou qualquer outro dispositivo que permita a apresentação de informações. Neste módulo iremos restringir-nos à escrita no ecrã através do uso da função **printf**.

printf

A função **printf** fornece-nos vários mecanismos para que possamos formatar e escrever dados no no ecrã mediante alguns especificadores de formato definidos pelo programador. Esta função, na sua forma mais simples, apresenta apenas um parâmetro que corresponde ao texto que pretendemos imprimir, na sua forma comum é composta por dois parâmetros, o primeiro correspondente ao texto e ao formato que se pretende usar e a segunda correspondente às variáveis com dados que pretendemos usar.

O primeiro parâmetro da função, além de texto, pode conter alguns especificadores de formato que são substituídos pelos valores das variáveis que o programador indicar antes da impressão no ecrã. Estes especificadores de formato aparecem misturados no texto, não precisando de ter qualquer espaço entre eles e o restante texto, e devem existir tantas variáveis quantos forem os especificadores.

As variáveis com os dados, que **têm de ser em número igual ao de especificadores de formato**, são colocadas depois da vírgula que termina o primeiro parâmetros, e são elas mesmas separadas por vírgulas: **printf(" texto com especificadores ", variável1, variável2, etc);**

Exemplo de uso da função.

```
int num_dentes = 31;
char ultima_letra = 'l';
char animal[] = "Raposa";

printf("Uma grande %s Azu%c com %d dentes!", palavra, ultima_letra, num_dentes);
```

Se colocarmos o código anterior dentro da função *main* e executarmos o programa o resultado obtido é: **Uma grande Raposa Azul com 31 dentes!**

Especificadores de Formatação

Os especificadores de formação são o que nos permite especificar o formato no qual pretendemos que os dados apareçam, são eles que nos permitem converter os dados ou alterar o aspecto como os dados são mostrados. Existem vários especificadores de formação, sempre compostos por dois caracteres, como evidenciado na tabela seguinte.

Tabela 8.1. Especificadores de Formatação

Especificador	Função
%c	Permite formatar a variável como um carácter e apresenta a sua representação.
%d ou %i	Usado para valores inteiros, em formato decimal, sem indicação de sinal.
%f	Usado para valores com vírgulas, floats ou double.
%e ou %E	Usado para valores com vírgulas, apresentam o valor em notação científica.
%o	Apresentam o valor em octal.
%x	Permite apresentar valores em hexadecimal
%g ou %G	Decide entre usar o %f ou %e, conforme o que der um resultado mais curto e não imprime os não significativos
%s	Imprime uma <i>String</i>

Caracteres de Escape

As sequências de escape são usadas como meios de controlar a consola para onde estamos a imprimir. Estas sequências não funcionam em aplicações que não sejam aplicações de consola, e o seu suporte depende da consola e do sistema operativo que estamos a usar. De modo genérico estas sequências, que nada mais são que caracteres especiais como os usados para a formatação dos dados, permitem efectuar operações como mudanças de linha, tabulações, alarmes, etc.

Tabela 8.2. Conjunto de Caracteres de Escape

Carácter	Função
\a	Campainha de sistema. Não funciona em todos os computadores
\t	Imprime uma tabulação, tipicamente 8 espaços
\b	Backspace, move o cursor uma posição para trás
\n	New Line, efectua a mudança de linha
\r	Carriage return, move o cursor para o início da linha
\\, \?, \", \'	Permite a escrita de barras (/), pontos de interrogação (?), aspas (") e apóstrofe (') (1)

(1) Estes caracteres têm significado especial e precisam ser escritos com uma barra descendente antes, \, de forma a que o seu significado especial seja anulado.

Capítulo 9. Leitura Formatada

Tal como na escrita formatada, apresentada anteriormente, a escrita formatada corresponde à formatação de dados que, neste caso, se foca na formatação dos dados pedidos aos utilizadores.

scanf

A função que nos permite ler dados segundo um formato definido por nós é a função **scanf**. Esta função recebe no seu primeiro argumento o texto com o formato e nos argumentos seguintes os endereços da variável ou variáveis onde os dados são guardados. A função **scanf** apenas lê os dados se o utilizador que os está a introduzir respeitar integralmente o formato que foi definido.

A maioria dos caracteres especiais de formatação são partilhados com a função **printf**, por exemplo, para ler um inteiro usamos o conjunto especial **%d** tal como quando pretendíamos mostrar um valor inteiro. Os caracteres que são diferentes dizem respeito às situações de controlo de leitura.

Como é natural, dado que estamos a ler informação do teclado e não a escrever informação no ecrã, os caracteres de controlo como **\n** não fazem sentido, no entanto, faz sentido conseguir controlar alguns dos aspectos da leitura, e por essa razão a função **scanf** possui alguns caracteres especiais para controlo de leitura.

Leitura para Endereços - Operador &

O operador **&** é usado para representar o acesso ao endereço de memória de uma variável. Podemos estabelecer o paralelo com a morada de uma carta, em que escrevemos no envelope a morada de destino. Do mesmo modo, um operador de endereço, **&**, permite especificar o endereço de destino dos dados que pretendemos gravar.

No caso da função **scanf** este operador é importante para permitir que a função guarde o valor lido do teclado na variável correcta e deve ser usado em todas as variáveis que **não** sejam ponteiros, isto é, para variáveis que sejam declaradas como ponteiros ¹ é necessário usar o operador. Das variáveis que foram usadas neste módulo, só as *Strings* são ponteiros, dessa forma, o operador é necessário em todas as variáveis mencionadas, excepto em variáveis que representem *Strings*.

Tabela 9.1. Especificadores de Formatação de Leitura

Especificador	Função
%c	Permite ler um carácter
%d ou %i	Permite ler um valor inteiro
%f	Permite ler um valor real, usado para <i>float</i> ou <i>double</i>
%s	Lê uma <i>String</i> , apenas o texto até encontrar o primeiro espaço
%o	Lê um valor em octal.
%x	Permite a leitura de um valor em hexadecimal
%p	Permite a leitura de um endereço de memória

¹ A noção de ponteiro será fornecida em módulos seguintes.

Caracteres de Escape

Tal como na função **printf** é possível com a função **scanf** usar alguns caracteres de controlo para alterar a forma como a leitura dos dados é processada.

Tabela 9.2. Tabela com caracteres de controlo de leitura

Especificador	Função
%n	Permite a leitura de um \n que por omissão não é possível ler
%[]	Permite a leitura do conjunto de caracteres que o programador colocar dentro dos parêntesis rectos
%^	Efectua a exclusão de caracteres, impedindo que os mesmos sejam lidos

Capítulo 10. Outras Funções de Leitura e Escrita

Além das funções **printf** e **scanf** já faladas e usadas para ler e escrever informação a partir do teclado e para o ecrã respectivamente, existem duas funções que permitem a leitura e escrita de caracteres.

As funções **getchar** e **putchar** permitem ler e escrever, respectivamente, um carácter. Ao contrário das funções de leitura e escrita formatada, **scanf** e **printf**, as funções **getchar** e **putchar** não recebem strings com formato ou usam os endereços de memória das variáveis. No caso da função **getchar** o valor lido é devolvido e no caso da função **putchar** o carácter a mostrar é passado directamente como o único parâmetro da função.

Programa Exemplo.

```
#include <stdio.h>

int main()
{
    char i;

    printf("Pressione uma tecla...");
    i = getchar();
    printf("\nA teclar pressionada foi: ");
    putchar(i);

    return 0;
}
```

Bibliografia

- Richard L. Petersen. *Introductory C, Second Edition: Pointers, Functions, and Files*. Morgan Kaufmann. 7 de Novembro, 1996. ISBN 978-0125521420.