

# **Programação em C/C+ + - Funções e Estruturas**

**Sérgio Lopes**

---

# Programação em C/C++ - Funções e Estruturas

Sérgio Lopes

---

# Índice

Sobre o Manual .....	vi
1. Público Alvo .....	vi
2. Estrutura e Conteúdo .....	vi
3. Licença .....	vi
1. Funções .....	1
1.1. Declaração e Implementação .....	1
1.2. Passagem de Parâmetros .....	2
1.3. Funções Recursivas .....	3
1.4. Visibilidade das Variáveis .....	5
2. Vectores .....	6
2.1. Dimensão de Vectores .....	8
2.1.1. Vectores Unidimensionais .....	9
2.1.2. Vectores Multidimensionais .....	9
2.2. Vectores e Funções .....	10
3. Strings .....	11
3.1. Strings Constantes .....	11
3.2. Vectores de Strings .....	11
3.3. Ler e Escrever .....	12
3.3.1. printf .....	12
3.3.2. scanf .....	13
4. Estruturas .....	15
4.1. Sintaxe de Estruturas .....	15
5. Uniões .....	17
5.1. Sintaxe das Uniões .....	17
6. Ponteiros .....	18
6.1. Uso de Ponteiros: Declaração os Operadores * e & .....	19
6.2. Ponteiros e Vectores .....	20
6.3. Ponteiros para Funções .....	20
6.4. Resumo .....	21
Bibliografia .....	22

---

## Lista de Figuras

1.1. Relação entre Parâmetros e Variáveis Locais .....	3
1.2. As Sucessivas Invocações e o Retorno .....	4
2.1. Iniciação do Vector .....	8
2.2. Acesso a Posições Individuais .....	8
6.1. Variáveis Tradicionais .....	18
6.2. Ponteiros na Memória .....	19

---

## Lista de Tabelas

3.1. Especificadores de Formatação .....	13
3.2. Conjunto de Caracteres de Escape .....	13
3.3. Especificadores de Formatação de Leitura .....	14
3.4. Tabela com Caracteres de Controlo de Leitura .....	14

---

# Sobre o Manual

## Público Alvo

Este manual foi criado, especificamente, para o módulo *0784 - Programação em C/C++ - Funções e Estruturas*, ministrado em formações profissionais como indicado pelo catálogo da ANQ, [www.catalogo.anq.gov.pt](http://www.catalogo.anq.gov.pt).

## Estrutura e Conteúdo

Neste manual são abordados, pela ordem indicada, os temas de: funções, vectores, estruturas, uniões e ponteiros. Na parte de funções não são explicadas as técnicas para organizar as funções por ficheiros, sendo todo o código criado no mesmo ficheiro onde a função *main* reside. A secção de vectores e ponteiros é também simplificada não sendo abordada alguma da sintaxe mais complexa na utilização de vectores, especialmente vectores com muitas dimensões ou aritmética de ponteiros em vectores multidimensionais.

A opção de colocar os ponteiros como última matéria prende-se com o facto de que os ponteiros são uma característica da linguagem que é complicada e que pode ser facilitada se todos os conceitos de funções, variáveis, âmbito das mesmas e tipos de dados do utilizador já tiverem sido explicados, formando assim uma base na qual será necessário acrescentar os ponteiros. Como o uso de ponteiros é transversal a todas as outras áreas da linguagem de programação C, optou-se por colocar a matéria de ponteiros no fim do manual.

## 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. Funções

Uma **função** é bloco de código com um *nome*, que tem de ser único em todo o programa, uma *lista de parâmetros* e um *tipo de retorno*, formando um sub-programa com objectivo específico. Uma **função** deverá executar uma tarefa simples.

Um programa escrito em linguagem C faz sempre uso de funções, sejam funções da biblioteca padrão, como o caso do *printf*, do *scanf*, do *getchar*, etc., seja através do uso da função *main*. Sendo esta última uma função obrigatória, podemos dizer que um programa em linguagem de programação C tem sempre, pelo menos, uma função. Mas as funções padrão nem sempre são suficientes e pode ser necessário criar as nossas próprias funções, chamadas de *funções de utilizador* dado que são criadas pelo utilizador da linguagem (o programador).

As funções permitem dividir o nosso código de modo a melhorar a organização e estrutura do mesmo e facilitar a escrita e manutenção do programa. Ao agrupar instruções que possuem uma relação lógica entre si, por exemplo todas as instruções necessárias para ler os dados pessoais de um utilizador, criando pequenos sub-programas com objectivos bem definido e que podem depois ser usadas como peças na construção de um programa maior conseguimos que o desenvolvimento seja mais fácil, que mais programadores possam estar a desenvolver o mesmo programa em simultâneo (cada programador pode fazer um função, ou um conjunto de funções) e melhoramos a qualidade do programa.

Se pensarmos na função *printf*, o objectivo da função é pequeno e específico: mostrar texto no ecrã. É também uma função que isolada não produz resultado útil e que é sempre usada para ajudar na criação e outras funções ou programas. Não sabemos também como é que a função *printf* está implementada, mas isso não é importante para que a possamos usar, o que é importante é que essa função, com um objectivo específico, permite que não estejamos sempre a implementar o mesmo código e torna a criação de programas mais simples.

## Declaração e Implementação

Para escrevermos as nossas funções é necessário duas coisas: **declarar** a função e **implementar** a função. Algo semelhante ao uso de variáveis, quando pretendemos criar uma função somos obrigados a declarar a função, situação onde indicamos o **nome da função, o tipo de retorno e o(s) tipo(s) do(s) parâmetro(s) que a função recebe**. Depois da declaração é necessário implementar a função, isto é, **escrever o código que a função vai conter** e que lhe confere a funcionalidade pretendida.

Estas duas partes, *declaração* e *implementação*, são feitas em locais separados mas sempre no nosso ficheiro principal com o código. No topo do ficheiro, imediatamente **antes da função main**, colocamos as declarações. No fundo do ficheiro, imediatamente **depois da função main**, colocamos a implementação das nossas funções. Embora o local onde colocamos as declarações e implementações possa ser diferente do que foi apresentado, durante este módulo iremos seguir sempre esta regra.

### Exemplo dos Locais de Declaração e Implementação.

```
//ZONA DE DECLRAÇÃO
float dobro(float valor);

//FUNÇÃO MAIN
int main()
{
```

```

float x, resultado;

printf("Insira um numero: ");
scanf("%f", &x);

resultado = dobro(x);

printf("Dobre de %.3f: %.3f", x, resultado);

return 0;
}

//ZONA DE IMPLEMENTAÇÃO
float dobro(float x)
{
    return 2 * x;
}

```

## Passagem de Parâmetros

Quando necessário para o código das funções, é possível passar para dentro das mesmas valores externos, provenientes de outras funções ou de outros blocos de código. Um exemplo desta situação, e que temos usado frequentemente, acontece na função *printf* em que colocamos dentro dos parêntesis o texto a mostrar (o primeiro parâmetro) e seguidamente as variáveis que pretendemos mostrar, se existirem algumas. Assim, com a passagem de parâmetros, conseguimos enviar valores para dentro as funções e usar esses mesmos valores no seu código.

A passagem de parâmetros para as nossas funções deve respeitar a declaração das mesmas, incluindo a ordem dos parâmetros. Devemos ter atenção ao facto de que ao passarmos parâmetros para dentro das nossas funções estamos a enviar cópias dos valores e que os valores originais nunca serão mudados<sup>1</sup>. É, também, importante notar que os parâmetros são transformados em variáveis locais da função e que deixam de existir quando a função terminar.

### Exemplo da Implementação de uma Função.

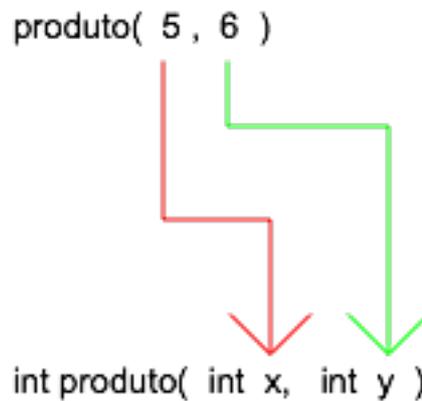
```

//Esta função recebe apenas um parâmetro chamado x
//e devolve um valor que corresponde ao
//resultado da operação de multiplicação de x por 2
float dobro(float x)
{
    return 2 * x;
}

```

Na imagem seguinte podemos ver a relação entre os valores que colocamos quando usamos uma função e a declaração de parâmetros, onde se incluem os nomes pelos quais os parâmetros vão ser conhecidos dentro da função.

<sup>1</sup>Mais tarde veremos que esta situação não é exactamente assim.

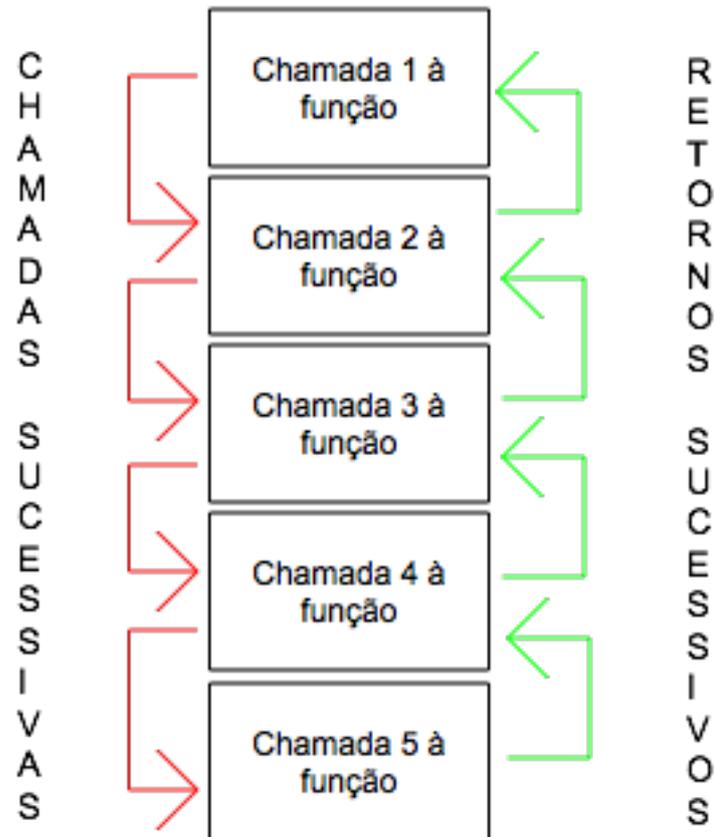
**Figura 1.1. Relação entre Parâmetros e Variáveis Locais**

## Funções Recursivas

Uma característica das funções é a de poderem ser usadas dentro de si mesmas, isto é, podemos criar uma função que usa na sua implementação uma chamada a si própria, a isto chamamos **recursividade**. Esta capacidade é apenas usada em situações muito particulares em que o algoritmo e o problema que estamos a tentar resolver faz uso da recursividade de forma natural.

O uso de recursividade na criação de funções requer atenção para que a função nunca seja criada sem uma condição de paragem, condição essa que permita definir um ponto que faça com que as repetidas invocações terminem. Sem esta condição, as contínuas chamadas só irão parar quando a memória existente for esgotada, sendo esta uma situação ilegal que provoca o fim anormal do programa.

No esquema seguinte podemos ver as chamadas sucessivas, a vermelho, que neste caso são apenas 5, e o retorno, a verde, que permite voltar à função inicial. O retorno inicia apenas a condição de paragem for verdadeira.

**Figura 1.2. As Sucessivas Invocações e o Retorno****Exemplo de Função Recursiva.**

```
#include <stdio.h>

int funcaoRecursiva(int x);

int main()
{
    printf("Iniciar a Chamada a Função Recursiva\n\n");

    funcaoRecursiva(10);

    return 0;
}

int funcaoRecursiva(int x)
{
    //condição de paragem que termina
    //a função quando o x é 1
    if(x == 1)
    {
        return ;
    }

    //chamar a mesma função novamente
    funcaoRecursiva(x - 1);

    //esta linha só será executada quando as várias
    //chamadas recursivas terminarem
    printf("X passou a 1 e estamos a terminar\n");
}
```

}

## Visibilidade das Variáveis

Antes de usarmos funções criadas por nós, todas as variáveis era declaradas dentro da função *main* e eram usadas sem restrições dentro dessa função. Ao passarmos a criar as nossas próprias funções podemos deparar-nos com a questão: onde declarar as variáveis?

Como vimos no módulo inicial, as variáveis possuem uma visibilidade, isto é, só estão disponíveis dentro do âmbito em que foram declaradas. Se isso no primeiro módulo não tinha qualquer implicação prática, neste módulo ao criarmos funções estamos a criar âmbitos diferentes, assim, as variáveis que declaramos dentro de uma função são existem dentro dessa função.

Portanto, qualquer variável está dependente do âmbito em que é criada, e só pode ser usada de acordo com esse âmbito. Sempre que criarmos uma função, as variáveis dessa função são locais não pode ser usadas fora dela. Isto implica que, mesmo que duas variáveis tenham o mesmo nome, se tiverem sido declaras em funções diferentes são variáveis distintas sem qualquer relação entre si.

---

# Capítulo 2. Vectores

Um vector <sup>1</sup> é um tipo de variável que agrupa, através do mesmo nome, um conjunto de elementos contíguos, permitindo que se consigam guardar conjuntos de dados que partilham o mesmo tipo. A criação de vectores segue as mesmas regras que a criação de outras variáveis com o acréscimo de ser necessário indicar o tamanho do vector.

Se considerarmos o exercício simples de calcular a média de um conjunto de notas, podemos tomar a seguinte implementação como exemplo:

```
#include <stdio.h>

int main()
{
    float soma = 0.0, nota, media = 0.0;
    int numNotas = 0;

    do
    {
        printf("Nota: ");
        scanf("%f", &nota);
        if(nota >= 0 && nota <= 20)
        {
            soma += nota;
            numNotas++;
        }
    }
    while(nota >= 0);

    if(numNotas > 0)
    {
        media = soma / numNotas;
    }

    printf("Média: %.2f", media);
}
```

No entanto esta implementação sofre de um problema: como é que conseguimos saber as notas que foram introduzidas? A cada iteração do ciclo a nota que é introduzida não é guardada, o que fica registado no fim do ciclo é a soma de todas as notas e o número de notas que foram introduzidos. Se for necessário mostrar as notas novamente? Ou então fazer outras contas com as notas como podemos prosseguir?

É em situações como esta que o uso de vectores é relevante dado que podemos guardar na mesma variável todas as notas registadas.

## Exemplo de Uso de Vector.

```
#include <stdio.h>

int main()
{
    //note-se a alteração de sintaxe e a
    //definição de tamanho para a variável notas
    float soma = 0.0, media = 0.0, notas[50], notaTemporaria;
```

---

<sup>1</sup>A palavra inglesa é *array*, alguns autores usam a tradução vector, outros matriz, outros ainda tabelas. Existem ainda autores que usam a palavra vector quando a dimensão é 1 e matriz quando a dimensão é superior. Neste manual iremos usar a palavra vector como tradução de *array* para qualquer dimensão.

```

int numNotas = 0, i = 0;

do
{
    printf("Nota: ");
    scanf("%f", &notaTemporaria);
    if(nota >= 0 && nota <= 20)
    {
        soma += notaTemporaria;
        numNotas++;

        //incrementar a variável que
        //controla a posição em que inserimos a
        //nota no vector de notas
        notas[i] = notaTemporaria;
    }
}
while(nota >= 0);

if(numNotas > 0)
{
    media = soma / numNotas;
}

printf("Média: %.2f", media);

printf("Notas Lidas: ");
for(i = 0; i < numNotas; i++)
{
    printf("%.2f ", notas[i]);
}
}

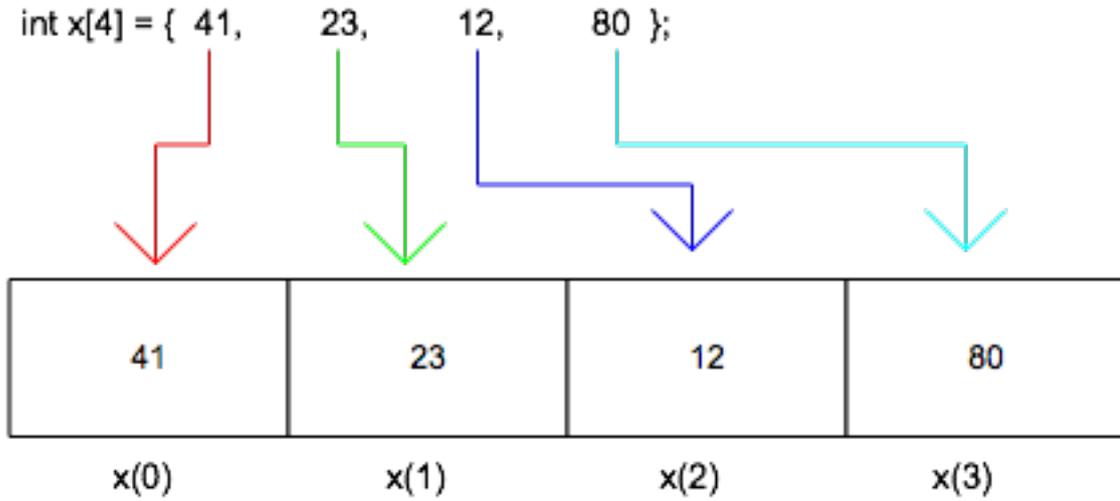
```

A sintaxe de vectores introduz a utilização de parêntesis rectos, para aceder às várias posições do vector e ao seu conteúdo, e chavetas para a iniciação do vector. Enquanto que com variáveis tradicionais a iniciação era feita colocando o valor inicial depois do sinal de igual, com vectores, e porque os vectores correspondem a várias posições, a iniciação tem de ser feita para cada uma das posições, sendo os vários valores colocados dentro de chavetas. Por exemplo, para iniciar um vector de inteiros com duas posições poderíamos usar o seguinte código:

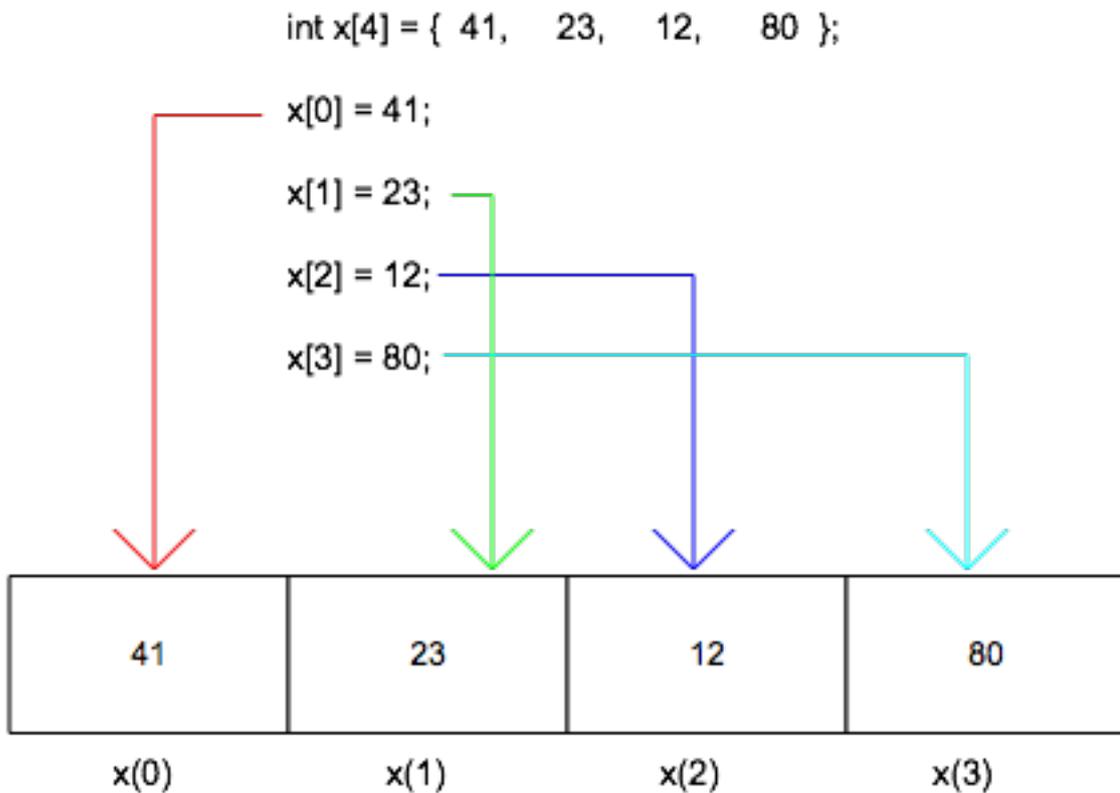
```
int v[2] = { 10, 210 };
```

Além da iniciação, o acesso às posições individuais é feito usando os parêntesis rectos e o índice da posição a que queremos aceder. Podemos ver nas imagens seguintes a relação entre as iniciações e as posições do vector, bem com o acesso às várias posições de forma individual.

**Figura 2.1. Iniciação do Vector**



**Figura 2.2. Acesso a Posições Individuais**



## Dimensão de Vectores

Ao declararmos um vector definimos sempre duas medidas: o tamanho, que corresponde à quantidade de posições e a dimensão. Iremos ver como estas duas medidas são usadas e como afectam o nosso código.

## Vetores Unidimensionais

Vetores de uma dimensão, designados por *unidimensionais*, são vetores que podemos ver como "*uma linha de várias colunas*". Isto é, são os vetores que possuem um tamanho indicado pelo valor entre parêntesis rectos, mas só possuem um par de parêntesis. São a forma mais simples de um vector que organiza as posições como uma única linha.

### Exemplo.

```
//vector de uma dimensão  
int v[100];
```

## Vetores Multidimensionais

Ao usarmos mais pares de parêntesis rectos definimos mais dimensões para os vetores que estivermos a declarar. Um vector de duas dimensões pode ser visto como uma folha quadriculada, em que a primeira dimensão corresponde às várias colunas, e a segunda dimensão às várias linhas presentes na folha, um vector de três dimensões pode então ser visto como um livro, um de quatro dimensões como uma prateleira cheia de livros. Isto é, cada dimensão acrescenta uma nova camada a ser usada para guardar os nossos dados.

### Exemplo.

```
//vector de uma dimensão  
int v[25];  
  
//vector de duas dimensão  
int v2[80][25];  
  
//vector de três dimensão  
int v3[100][80][25];  
  
//vector de quatro dimensão  
int v4[5][100][80][25];
```

Em linguagem C, os vetores de várias dimensões são designados por vetores de vetores. Na verdade um vector de duas dimensões corresponde a um vector em que cada posição guarda um segundo vector.

## Vetores Rectangulares: Matrizes

Um caso especial dos vetores multidimensionais, os vetores de duas dimensões são bastante úteis dada a forma como os podemos equiparar a tabelas. Efectivamente, a forma mais simples de olhar para estes tipos de vetores é considerar que representam uma tabela, composta por várias linhas e colunas<sup>2</sup>. Situações onde possuímos dados que pretendemos representar como uma tabela tornam o uso deste tipo de vetores ideal.

O exemplo mais comum e fácil de perceber será uma lista de nomes de pessoas: ora se um nome de uma pessoa é composto por um conjunto de caracteres, um vector, e se uma lista é também um conjunto de elementos, podemos guardar a informação num vector de duas dimensões, em que a primeira dimensão contém as colunas (os caracteres que constituem o nome) e a segunda dimensão corresponde a cada um dos vários nomes formando a lista.

---

<sup>2</sup>Esta é apenas uma representação que podemos usar para nos ajudar a entender a estrutura de dados, não correspondendo à forma como os dados são organizados internamente

### Exemplo de uma Tabela com Nomes de Pessoas.

```
int main()
{
    char lista[10][200];
    int i;

    //Ler os 10 nomes
    for(i = 0; i < 10; i++)
    {
        printf("Indique o nome da pessoa: ");
        scanf("%s", lista[i]);
    }

    //Mostrar os 10 nomes
    for(i = 0; i < 10; i++)
    {
        printf("%s", lista[i]);
    }

    return 0;
}
```

## Vetores e Funções

A passagem de vetores para funções segue as mesmas regras que a passagem de ponteiros, como só abordaremos os ponteiros no último capítulo, esta secção será uma breve introdução.

Uma função não consegue devolver um vector que tenha sido declarado no seu interior, embora seja possível devolver valores de variáveis que não sejam vetores, não é possível devolver vetores. Esta limitação é imposta pela forma como as variáveis estão dependentes do âmbito em que são declaradas. Efectivamente, um vector só fará sentido existir dentro da função em que é criado e nunca poderá ser devolvido por uma função.

Se for necessário modificar valores de um vector, então é possível passar o vector como parâmetro da função em vez de o declarar dentro da função. Desta forma, devido à relação entre vetores e ponteiros que veremos mais tarde, conseguimos que os dados de um vector seja enviados para fora de uma função, apesar de não estarmos a usar a palavra *return* nem de estarmos a devolver o vector.

---

# Capítulo 3. Strings

*Strings* é o nome dado a um conjunto de caracteres que, tipicamente, permitem representar palavras e frases. Em C *Strings* são vectores de caracteres com um carácter terminador, o `\0`, e como todos os outros vectores permite guardar um conjunto de elementos, neste caso do tipo `char`, organizados de forma sequencial. A diferença entre os vectores de caracteres e uma *String* é tão só o uso ou não do terminador especial: um vector de caracteres não tem terminador enquanto que um vector de caracteres a que chamamos *String* tem. Este terminador é importante dado que muitas funções e funcionalidades disponíveis esperam que exista um terminador para que saibam onde termina o texto.

Estas *Strings* são aquilo a que chamamos de *Strings variáveis* dado que são colocadas em variáveis. Não confundir com variáveis que sejam constantes.

## Strings Constantes

Chamamos *Strings* constantes a qualquer texto que esteja colocado entre aspas. Estas *Strings* não são consideradas constantes porque são imutáveis e estão registadas no código, e embora seja possível atribuir estas *Strings* a variáveis, a verdade é que estaremos a atribuir sempre cópias, sendo que o texto entre aspas terá uma zona de memória diferente.

## Vectores de Strings

Vectores de *Strings* são vectores bidimensionais, onde a primeira dimensão define a quantidade de caracteres máxima que cada *String* pode ter e a segunda dimensão define o número de *Strings* que estamos a guardar. Um exemplo desta utilização será a criação de listas de nomes, em que cada nome é uma *String*, e nesse sentido um vector de caracteres, e para cada posição da lista termos um outro vector formando assim as duas dimensões.

### Exemplo de Tabela de Nomes.

```
int main()
{
    char pessoas[15][150];
    int inseridos = 0;
    char lixo;

    do
    {
        printf("Indique o nome: ");
        scanf("%s", pessoas[inseridos]);
        while(( lixo = getchar() ) != '\n' && lixo != EOF);
        inseridos++;
    } while(inseridos < 15);

    for(i = 0; i < inseridos; i++)
    {
        printf("Nome da Pessoa na Posicao %d: %s\n", i + 1, pessoas[i]);
    }
}
```

# Ler e Escrever

Tal como vimos no módulo 782 - *Programação em C/C++ - Estrutura Básica e Conceitos Fundamentais*, a leitura de dados, correspondente à introdução de informação pelo utilizador do programa, e a escrita de dados, que diz respeito à apresentação de informação ao utilizador do programa, é feita recorrendo às funções de leitura e escrita formatada *scanf* e *printf* respectivamente.

Este capítulo apresenta um resumo da informação presente no primeiro módulo de introdução à linguagem de programação C.

## printf

A função **printf** fornece-nos vários mecanismos para que possamos formatar e escrever dados 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);
```

Executando o código anterior num programa escrito correctamente, o resultado que obtemos é: **Uma grande Raposa Azul com 31 dentes!**.

## Especificadores de Formatação

A função *printf* possui os seguintes operadores especiais para controlar o formato do texto que é impresso:.

**Tabela 3.1. 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, <i>floats</i> ou <i>double</i> .
%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 3.2. 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	<i>Backspace</i> , move o cursor uma posição para trás
\n	<i>New Line</i> , efectua a mudança de linha
\r	<i>Carriage return</i> , 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.

## 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.

**Tabela 3.3. Especificadores de Formatação de Leitura**

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

## 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 3.4. Tabela com Caracteres de Controlo de Leitura**

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

---

# Capítulo 4. Estruturas

Surge muitas vezes a necessidade agrupar informações diversas sob um mesmo nome, de gerir informação relacionada de forma simples e sem termos de criar centenas de variáveis diferentes com nomes muito parecidos e que se torna complexos de gerir. Tomemos o exemplo de uma agenda electrónica, é importante guardarmos, para uma mesma pessoa, o nome, a data de nascimento, o telefone, o telemóvel, etc., informação esta que faz sentido estar relacionada.

Com os tipos de variáveis que vimos até agora, seja ponteiros, vectores ou variáveis tradicionais, este tipo de relação não é possível. Não existe uma forma simples de relacionar os vários tipos de informação, muitas vezes com tipos de dados diferentes (os nomes serão texto, mas os telefones já poderão ser números inteiros). Para estas situações, onde a relação da informação é notória e importante, a linguagem de programação C fornece-nos um mecanismo designado por **estruturas**.

**Estruturas** são colecções de uma ou mais variáveis, com tipos de dados que podem ser diferentes, ao contrário dos vectores, e que são manipuladas através de um nome único. Estas **estruturas** permitem criar o que se chama de **tipos de dados do utilizador**, isto é, permitem criar novos tipos de dados que não fazem parte da linguagem base (tipos como int, float, char, etc).

## Sintaxe de Estruturas

A sintaxe das estruturas introduz a palavra reservada **struct** e faz uso das já conhecidas chavetas, { e }. As regras para as criações e declarações de estruturas são as mesmas que para as restantes variáveis, os nomes das estruturas devem seguir as mesmas regras que as usadas nas variáveis e nas funções. No fim da definição da estrutura é necessário colocar um ponto e vírgula.

A utilização das estruturas obriga a que seja definidas as estruturas antes das mesmas serem declaradas<sup>1</sup>. Assim, é necessário criar a estrutura, processo pelo qual indicamos ao compilador quais os campos que constituem a nossa estrutura, e declarar a variável que faz uso do novo tipo do mesmo modo que declaramos uma variável de um tipo que seja fornecido pela linguagem.

### Exemplo de Estrutura para Contactos.

```
//criação de uma estrutura
struct contacto
{
    char nome[150];
    char telefone[9];
    char telemovel[9];
    char morada[500];
    int dataNascimento[3]; //dia,mes,ano
};

int main()
{
    //declarar uma variável chamada x
    //do tipo struct contacto
    struct contacto x;

    //declarar um vector de 30 posições
    //do tipo struct contacto, chamado agenda
    struct contacto agenda[30];
}
```

---

<sup>1</sup>Embora seja possível definir e declarar uma estrutura numa só operação, não iremos seguir essa sintaxe

```
    return 0;  
}
```

---

# Capítulo 5. Uniões

**Uniões** são estruturas de dados similares às **estruturas** que vimos no capítulo anterior, com a importante diferença de que só um dos membros pode estar definido a cada momento. Isto é, se criarmos uma **união** para guardar o número de telefone e o nome de uma pessoa, usando dois vectores de caracteres, apenas um pode estar preenchido a cada momento, sendo que se preenchermos o telefone e de seguida o nome perdemos o valor do telefone.

## Sintaxe das Uniões

As **uniões** seguem a mesma sintaxe que as **estruturas** com a diferença de, em vez da palavra *struct*, se usar a palavra **union**.

**Exemplo da Definição de uma União.**

```
union numero
{
    int comSinal;
    unsigned int semSinal;
};
```

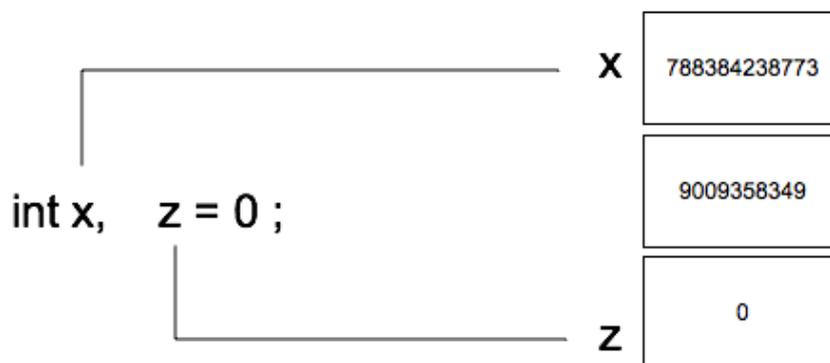
---

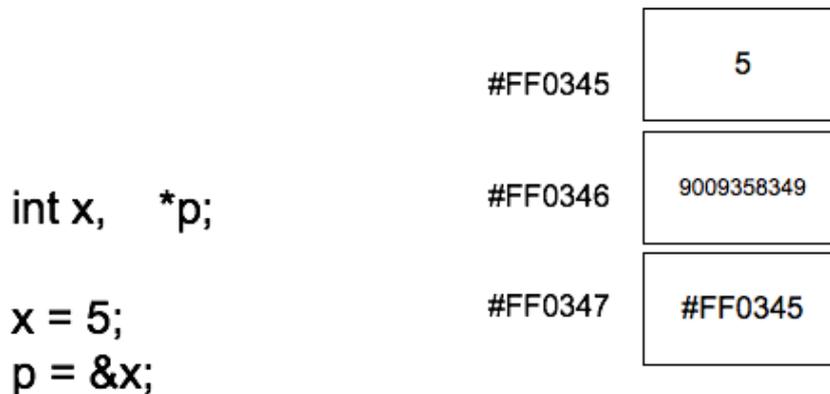
# Capítulo 6. Ponteiros

Um ponteiro é uma variável que, tal como outras variáveis, reside em memória, possui um endereço e um valor. No entanto os ponteiros guardam apenas um tipo especial de valores: **endereços de memória de outras variáveis**.

Quando uma variável é declarada, o sistema operativo trata de reservar um espaço de memória com o tamanho necessário e de devolver ao programa o endereço da zona de memória reservada. É com esse endereço que todos os acessos são feitos, e sem que o programador tenha de conhecer ou manipular endereços de memória. Através do uso da variável o programador está a usar indirectamente o endereço de memória.

**Figura 6.1. Variáveis Tradicionais**



**Figura 6.2. Ponteiros na Memória**

A sintaxe de ponteiros faz uso de dois caracteres principais: `*` e `&`. O primeiro permite aceder ao *valor da variável para a qual o ponteiro aponta*, o segundo é usado para aceder ao *endereço de uma variável e assim atribuir o valor desse endereço* a um ponteiro.

Os ponteiros possuem sempre um tipo de dados associado, e que controla as operações que lhe podem ser aplicadas. Se um ponteiro é declarado como *ponteiro de inteiros* então não poderá apontar para um endereço de uma variável que seja declarada como *float*. A única exceção são os ponteiros para o tipo de dados *void*, dado que este tipo de dados é especial, os ponteiros para *void* podem depois ser usados para apontar para qualquer outra variável, no entanto cabe ao programador ter o cuidado de garantir que todos os acessos são válidos.

## Uso de Ponteiros: Declaração os Operadores `*` e `&`

A sintaxe de ponteiros faz uso de dois caracteres principais: `*` e `&`. O primeiro é usado na declaração e no acesso ao valor da variável para a qual o ponteiro aponta, o segundo é usado para aceder ao endereço de uma variável e assim atribuir o valor desse endereço a um ponteiro.

A declaração de ponteiros segue as mesmas regras que a declaração dos restantes tipos de variáveis que temos visto até ao momento. É necessário definir o **tipo de dados**, o **nome do ponteiro** e o valor inicial, caso exista. Além destes elementos comuns, é necessário usar um asterisco antes do nome da variável.

### Exemplo de Uso de de um Ponteiro para Inteiros.

```
int main()
{
    int *p, x;

    x = 20;
```

```

//colocar o endereço de x no
//ponteiro p
p = &x;

//através do p conseguimos usar o
//valor de x
printf("Valor de x: %d\n", *p);

//conseguimos também alterar o
//valor do x
(*p) = 30;

//efectivamente, na linha de código anterior
//modificámos o valor do x e não do p
printf("Valor de x: %d\n", *p);

return 0;
}

```

## Ponteiros e Vectores

Vectores são ponteiros.

Embora sejam declarados usando uma sintaxe diferente da que aprendemos com ponteiros, usando o `*`, a verdade é que uma variável que seja declarada como vector está na verdade a criar um ponteiro para a primeira posição da zona de memória que foi reservada. É por esta razão que ao falarmos de passagem de vectores para funções referimos que as regras são as mesmas dos ponteiros.

No entanto estes vectores são ponteiros com uma particularidade: são ponteiros constantes, apontando sempre para a mesma zona desde o momento em que foram criados até ao momento em que são destruídos. Esta diferença implica que, se declararmos um vector, e embora ele seja realmente um ponteiro, não possamos atribuir outros endereços de memória à variável.

### Exemplo Errado de Alteração de Vectores.

```

int main()
{
    int v[10], x = 5, *p;

    //se p é um ponteiro e foi declarado
    //como tal, então podemos atribuir-lhe o
    //endereço de uma variável, neste caso o x
    p = &x;

    //ERRADO! o v é um vector, embora
    //um vector seja um ponteiro, é um
    //ponteiro constante e não podemos
    //alterar o local para onde aponta
    v = &x;
}

```

## Ponteiros para Funções

Além de variáveis, a memória de um programa guarda também as funções que estão carregadas e que pertencem a esse programa. Seguindo essa característica, e considerando que um ponteiro guarda apenas endereços de memória, é possível criar um ponteiro que aponte para um endereço de memória onde está a informação de uma função. Estes ponteiros para funções permitem fazer funções genéricas ou até funções que recebem como parâmetros de entrada outras funções.

# Resumo

A seguinte lista apresenta um resumo das regras e cuidados a ter com o uso de ponteiros durante o desenvolvimento dos nossos programas.

- A** Um ponteiro guarda apenas um endereço, não guarda valores numéricos, letras, ou qualquer outro tipo de dados, **apenas um endereço de memória**.
- B** Um ponteiro é dependente do tipo de dados para o qual aponta. Um ponteiro para *char* só pode guardar caracteres e um ponteiro para *int* só pode guardar inteiros.
- C** Como qualquer outra variável, um ponteiro tem um valor e um endereço de memória mas acrescenta a capacidade de referenciar outros valores das variáveis para as quais aponta.
- D** A sintaxe para acesso a ponteiros é confusa! Devem ter atenção ao código escrito e fazer uso de parêntesis sempre que se misturem ponteiros com outras operações, quando não têm a certeza da prioridade dos operadores ou quando se usem formas de ponteiros que não são padrão, ex:

```
int y = 9, *p, b;
p = &y;

b = *p++; // b fica com valor 9,
          //p aponta para zona de memória seguinte

b = *(p++); //igual ao anterior

b = (*p)++; //b fica com o valor 9
           //incrementa b uma unidade, b fica com 10

b = *++p; //p é incrementado uma unidade, p e não o valor!
          //p fica assim para a zona de memória seguinte
          //b fica com valor imprevisível

b = ++*p; //incrementa y, coloca o valor de y em b
          //b fica com valor 10
```

- E** Um vector é um ponteiro para a primeira posição de um conjunto de memória sequencial. No entanto é um ponteiro constante, sem que possa ser alterado o seu valor inicial.
- F** O uso de ponteiros é perigoso! A utilização de ponteiros permite a manipulação directa da memória. O compilador não consegue validar o nosso código e garantir que o que estamos a fazer é válido. Do mesmo modo, o sistema operativo não controla o que o programa faz e é fácil aceder a zonas de memória que não nos pertencem e danificar outros programas instalados e a correr ao mesmo tempo, ou até inutilizar um sistema operativo. Embora os sistemas actuais possuam mecanismos que permitem detectar alguns comportamentos incorrectos e forçar o programa a terminar, é necessário ter atenção ao código que é escrito.

---

# Bibliografia

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