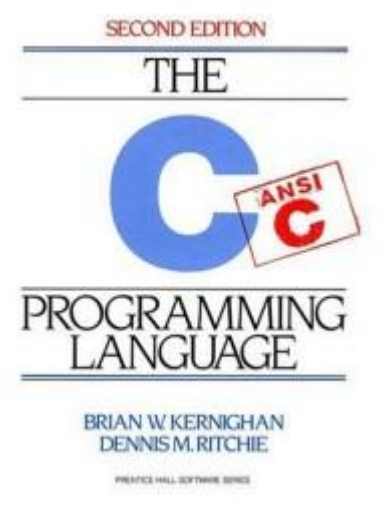
# Introdução à linguagem C




**Dennis Ritchie Brian Kernighan**

# Nota prévia

Esta é a segunda cadeira em que os alunos lidam com a linguagem C. Nas próximas aulas faremos um percurso pela maioria dos mecanismos da linguagem C, discutido-os com base nos conceitos gerais apresentados na cadeira.

Mas a ênfase será colocada nas seguintes partes, onde se espera que os alunos ganhem novas competências:

- Estruturas de dados dinâmicas e manipulação de apontadores.
- Polimorfismo implementado usando apontadores e macros.
- Módulos.
- Discussão das inseguranças da linguagem C.

Nesta linha:

- Toda a aula prática 2 será sobre estruturas de dados dinâmicas e manipulação de apontadores.
- O projeto de programação também será principalmente sobre estruturas de dados dinâmicas e manipulação de apontadores.
- No exame, quase sempre os problemas sobre C são sobre estruturas de dados dinâmicas e manipulação de apontadores.

# Algumas características do C

- Concebida e implementada por Dennis Ritchie entre 1969 e 1973 nos Bell Labs da AT&T. A primeira versão do Unix foi escrita em assembler, mas em 1973 o Unix foi reescrito em C.
- Uma das linguagens atualmente mais populares tem sido utilizada para escrever todo o tipo de aplicações tais como compiladores, bases de dados, sistemas operativos (Unix por exemplo), editores gráficos e de texto, etc.
- É uma linguagem padronizada com standards ANSI e ISO.
- Linguagem de alto nível, mas que oferece facilidades para manipulações de baixo nível, nomeadamente acesso direto à memória. Originalmente concebida para programação de sistemas, o C permite ao programador escrever código muito eficiente e com acesso direto aos recursos da máquina.
- Apesar dos mecanismos de baixo nível, a linguagem encoraja independência da máquina. Este aspeto é extremamente importante pois cumpre um dos objetivos das linguagens de programação que é a possibilidade de escrever programas independentes de cada máquina particular.
- A maioria das implementações são muito eficientes.
- Necessidade de uma boa disciplina na programação para produção de programas legíveis --> WOP - 'write only programming' (observação irónica).
- Tipos básicos: caracteres, inteiros, reais, booleanos, enumerados.

- Tipos derivados: vetores, registos, uniões, apontadores.
- Tipificação fraca: por exemplo, os caracteres e os valores enumerados são tratados como inteiros.
- Não têm gestão automática de memória (ao contrário do Java e do OCaml). É necessário efetuar gerir a memória manualmente (usando as funções "malloc" e "free"). A gestão manual de erros é uma conhecida fonte de erros no software.
- Estruturas de controlo: condicionais, iterativas, seletivas.
- Funções. Recursividade é suportada. Não suporta aninhamento de funções.
- Aritmética de apontadores.
- Modularidade e compilação separada.
- Biblioteca padrão.
- Oferece um grande controlo ao programador:
    - a nível sintático devido ao sistema de macros implementado no pré-processador;
    - a nível de dados por causa dos apontadores, uniões e campos de bits;
    - a nível de abstrações de execução por causa dos apontadores para função;
    - a nível de modularidade devido a um sistema de módulos simples e flexível que suporta compilação separada e ocultação de informação.

## Exemplo 1

```
#include <stdio.h>

int fact(int i)
{
    if( i==0 ) return 1 ;
    else return i * fact(i-1) ;
}

int main(void)
{
    int n ;

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

## Exemplo 2

```
int main(int argc, char *argv[])
{
    int i ;
    for( i = 0 ; i < argc ; i++ )
        printf("%s\n", argv[i]) ;
    return 0 ;
}
```

# Padronizações do C

- K&R C - Padrão informal estabelecido em 1978 com a publicação da 1ª edição do livro "The C Programming Language". Os argumentos das funções não eram validados e as funções retornavam inteiros por omissão.
- Ansi C89 - Padrão criado em 1983 mas só ratificado em 1989. Foram introduzidos **protótipos** que, quando presentes, permitem validar os argumentos e resultado das funções.
- ISO C90 - O padrão anterior foi adotado com muito ligeiras alterações pela ISO em 1990.
- ISO C99 - Padrão de 1999. Introduziu diversas características úteis tais como flexibilidade quanto ao ponto onde se definem as variáveis, vetores de tamanho variável e booleanos.
- ANSI C99 - Em 2000, a ANSI adotou o padrão anterior sem alterações.

O GCC é uma das implementações de C atualmente mais usadas. Suporta a maioria do C99 e mais algumas extensões, das quais a mais notável é a possibilidade de definir funções locais a outras funções, algo que sempre foi proibido no padrão do C.

Como o suporte para C99 no GCC ainda não é completo, por omissão o GCC ainda se baseia no C89. Para forçar o GCC a reconhecer o C99, tanto quanto possível, é necessário invocar o GCC assim:

```
gcc -std=c99
```

# Documentação sobre o compilador de C

Olhamos o que diz o início do manual do comando `cc`, no Linux:

```
$ man cc
GCC(1)                                          GNU
GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below for the remainder.  g++
accepts
       mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation, assembly and
linking.
       The "overall options" allow you to stop this process at an intermediate stage.
For exam-
       ple, the -c option says not to run the linker.  Then the output consists of
object files
       output by the assembler.

       Other options are passed on to one stage of processing.  Some options control the
prepro-
       cessor and others the compiler itself.  Yet other options control the assembler
and
       linker; most of these are not documented here, since you rarely need to use any
of them.
```

---

# Elementos Básicos

## Programa em C

Sequência de constantes variáveis, definições de tipos e funções, possivelmente distribuídas por vários ficheiros.

## Identificadores

Começam por uma letra podendo conter letras, algarismos e ainda o carácter sublinhado '_'. É feita distinção entre maiúsculas e minúsculas.

## Delimitadores de comentário

```
/* comentário  */

// os comentários de linha foram introduzidos no padrão C99
```

## Terminador de instrução

Todas as declarações e todas as instruções são terminadas por um ponto e vírgula ';'. Exceção: as funções são terminadas por uma chaveta a fechar '}'.

## Literais

Há literais dos tipos carácter, inteiro, real, string e booleano. Exemplos:

- carácter: 'a' '\n' '\t' '\r' '\0' '\123'
- inteiro: 1 5 21056 -56
- real: 5.6 4e7 -5E-5
- string: "" "supercalifragilisticoexpialidoso"
- bool: false true

## Definição de constantes

```
#define PI   3.1415962
#define a6   "aaaaaa"
#define um   1
```

## Inicialização de variáveis

```
int i = 100 ;
double d = 12.3e56 ;
```

As variáveis estáticas são inicializadas a zero por omissão.

## Atribuição a variáveis

```
v = 14 ;
x = 5 + 7 + v ;
```

---

# Tipos básicos

## Tipos numéricos

A linguagem suporta os seguintes cinco tipos básicos numéricos:

- char
- short
- int
- float
- double

Um char ocupa 1 byte, um short ocupa pelo menos 2 bytes, um int ocupa pelo menos 2 bytes (normalmente tem 4 bytes em máquinas de 32 bits).

O ficheiro <limits.h> contém informação sobre o tamanho exato de cada tipo, na implementação de C que estiver a ser usada.

O operador `sizeof` também pode ser usado para saber qual o número de bytes ocupados por um valor de qualquer tipo.

**Qualificadores dos tipos numéricos**

Alguns dos tipos numéricos podem ter a sua semântica modificada por meio dos seguintes qualificadores:

- long (int, double)
- long long (int)
- unsigned (char, short, int, long int)
- signed (char, short, int, long int)

Um `long int` ocupa pelo menos 4 bytes; um `long long int` ocupa pelo menos 8 bytes.

Exemplos de tipos numéricos qualificados:

```
unsigned int
long double
unsigned long int
unsigned char
```

Por omissão todos os tipos são signed exceto o tipo char cujas características dependem da implementação.

O nome do tipo int pode ser omitido, quando qualificado. Portanto os seguintes são tipos válidos:

```
long
unsigned
signed
```

# Tipo booleano

Tradicionalmente, o C não costumava ter um tipo booleano explícito, embora o conceito sempre tenha existido na linguagem. O valor de verdade é representado por qualquer valor diferente de zero, e o valor de falsidade é representado por zero. Existem três operadores lógicos que interpretam os argumentos como "booleanos": &&, ||, !.

Foi introduzido no padrão C99 um tipo chamado `bool` com os literais `false` e `true`, mas o uso desse tipo requer a inclusão do ficheiro <stdbool.h>. O `false` é simplesmente representado usando o inteiro 0 e o `true` é representado usando o inteiro 1.

# Tipos enumerados

Os tipos enumerados foram introduzidos no padrão C89. São úteis para especificar um número de opções para um atributo. Exemplos de possíveis atributos: cor, mês do ano, dia da semana, etc.

Exemplo:

```
typedef enum {
    JANUARY, FEBRUARY, MARCH,
    APRIL, MAY, JUNE,
    JULY, AUGUST, SEPTEMBER,
    OCTOBER, NOVEMBER, DECEMBER
} Month ;
```

Os valores dos tipos enumerados são representados usando inteiros e a representação não é oculta. Por defeito os valores começam em zero e são incrementados sucessivamente - portanto JANUARY vale 0, FEBRUARY vale 1, etc. Veja a seguinte função:

```
Month NextMonth(Month m) {
    return m == DECEMBER ? JANUARY : m + 1 ;
```

```
}
```

O C permite mesmo ao programador especifique a representação inteira de cada valor enumerado. Exemplo:

```
typedef enum {
    RED = 1, GREEN = 2, BLUE = 4, YELLOW = 8
} Color ;
```

## Tipos derivados

Os tipos derivados serão discutidos mais tarde:

- Arrays
- Registos (structures)
- Uniões
- Apontadores

---

# Variáveis

## Definição

As variáveis podem ser definidas globalmente, fora de qualquer função, ou definidas localmente, dentro duma função, logo no primeiro nível ou então dentro dum bloco interno.

```
int count, n ;
double stdvar, media ;
char car, key ;
```

## Inicialização

As variáveis podem ser inicializadas no momento da sua definição.

```
int x = 6 ;
Complex z = { 2.0, 5.7  };
int v[5] = { 1, 2, 3, 4, 5 } ;
```

Na ausência de qualquer expressão de inicialização, as variáveis globais e as variáveis locais estáticas são inicializadas a zero.

As variáveis locais não têm qualquer inicialização por omissão, ficando indefinidas (com um valor aleatório) enquanto não se fizer a primeira atribuição.

## Atributos das variáveis locais

Eis os atributos disponíveis para as variáveis locais, e o seu significado:

- **static** - A variável mantém valor entre ativações da função.
- **auto** - (atributo por omissão) variável automática ou seja não estática.
- **register** - Se possível a variável é guardada num registo do CPU.
- **volatile** - Indica que o valor da variável pode mudar fora do controlo do compilador. Exemplo: posição de memória cujo valor pode ser alterado pelo hardware quando da ocorrência de um interrupt.
- **const** - A variável é inicializada no ponto da sua definição e o seu valor não pode ser alterado depois. Também se aplica a argumentos de funções.

## Atributos das variáveis globais

Eis os atributos disponíveis para as variáveis globais, e o seu significado:

- **static** - A variável é privada no ficheiro, não podendo ser usada a partir de outros ficheiros fonte.
- **volatile** - Indica que o valor da variável pode mudar fora do controlo do compilador.
- **const** - A variável é inicializada no ponto da sua definição e o seu valor não pode ser alterado depois.

Relativamente aos diversos tipos de variáveis, classifique as respetivas ligações em função do momento de ligação e diga qual é o tempo de vida de cada uma delas.

---

# Estruturas de controlo

Observe as posições onde se escreve o terminador ';'.

```
if(exp) stat

if(exp) stat else stat

while(exp) stat

do stat while(exp);

for(init; test; advance) stat

switch(exp){
    case const: stats
    case const: stats
    default: stats
}

{ // bloco
    decls e stats
}

exp;

break;

continue;

return;

return exp;

goto label;

label: stat

; // instrução nula
```

# Operadores

Eis a lista completa dos operadores do C que podem ser usados em expressões:

```
aritméticos:    +  -  *  /  %
lógicos:        !  &&  ||
relacionais:    <  >  <=  >=  ==  !=
bits:           >> << & ^ |~
atribuição:     =  +=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=
condicional     ?:
cast:           (type)
inc/dec:        expr++  ++expr  expr--  --expr
sequenciação:   ,
sizeof:         sizeof
apontadores:    *  &  ->  []
field:          .               (para acesso a campos de registos e uniões)
agrupamento:    (expr)
```

Note que em C (tal como em Java), a atribuição produz um resultado e por isso é considerada uma expressão, não um comando. O valor da expressão `v = exp` é o valor que fica na variável `v` depois da expressão `exp` ter sido avaliada e depois da atribuição ter sido concretizada.

O C faz sobrecarga (overloading) de alguns operadores. Por exemplo, o operador + é usado para denotar três operações diferentes: a soma inteira; a soma real; a soma entre um apontador e um inteiro.

## Precedências e associatividades

Precedências dos operadores por ordem decrescente de prioridade:

```
()
[] -> . expr++ expr--                              esq
! ~ ++ -- - (type) * & sizeof ++expr --expr        dir
* / %                                              esq
+ -                                                esq
<< >>                                              esq
< <= > >=                                          esq
== !=                                              esq
&                                                  esq
^                                                  esq
|                                                  esq
&&                                                 esq
||                                                 esq
?:                                                 esq
= += -= etc.                                       dir
,                                                  esq
```

Exemplo:

```
if( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
    printf("Ano bissexto\n") ;
else
    printf("Ano comum\n") ;
```

---

# Avaliação de expressões

## Ordem de avaliação

O compilador tem a liberdade de rearranjar as expressões por forma a optimizar a eficiência da sua avaliação mesmo que esta envolva efeitos laterais. Os parêntesis podem mesmo não ser respeitados (!) se a sua mudança de posição não violar nenhuma das regras da álgebra. O compilador também ter a liberdade de avaliar os argumentos nas chamadas das funções por qualquer ordem.

Portanto a ordem de avaliação não está geralmente definida e devemos evitar escrever expressões cujos efeitos ou resultados dependam da ordem de avaliação. Exemplos:

```
i = i++ ;          /* o valor final de i não está definido */
f(i++, i++) ;      /* o valor dos argumentos não está definido, mas o valor final de i
não tem problema */
f(*p1++, *p2++) ;  /* o valor dos argumentos não está definido no caso dos dois
apontadores referirem a mesma posição de memória */
f() + g()          /* qualquer das funções pode ser executada em primeiro lugar */
```

O último exemplo só é problemático se as duas funções produzirem efeitos laterais que sejam dependentes da ordem de avaliação.

## Pontos de sequenciação

Mas nem tudo está indefinido na ordem de avaliação de expressões. Temos os **pontos de sequenciação** para nos ajudar. São pontos dentro das expressões que garantem que os efeitos laterais das expressões anteriores já foram completamente concretizados.

Os pontos de sequenciação do C estão associados aos seguintes operadores:

```
,  &&  ||  ?:
```

## Hierarquia dos tipos numéricos

Os tipos numéricos podem ser livremente misturados em expressões. Quando isso acontece, são efectuadas promoções automáticas de tipo de acordo com a seguinte hierarquia:

```
char short
int
unsigned int
long int
unsigned long int
long long int
unsigned long long int
double
long double
```

Fora desta hierarquia encontra-se o tipo `float`, o que é promovido para `double` nos contextos onde se fazem contas com doubles.

## Conversões automáticas de tipos numéricos

Aplicam se sempre sucessivamente as seguintes regras na avaliação de uma expressão:

1. char, short, enum -> int
2. Para cada operando binário com operandos de tipo diferente, o menos importante é convertido no tipo do mais importante, antes de se efectuar a operação.
3. Nas atribuições (v = exp) o tipo do valor da direita é convertido num valor do tipo da esquerda antes de se fazer a atribuição. Muitas vezes isso implica uma despromoção de tipo e uma truncagem de valor.

Exemplos:

```
5 + 2.0 * 'a'
(5.3 + 5) + 7
int i = 200 * 'a'
```

---

# Tipos derivados

Há 4 variedades de tipos derivados:

- Vectores (arrays)
- Registos (structures)
- Uniões
- Apontadores

Podem ser usados directamente, como na seguinte declaração de variável

```
struct
{
    double re, im ;
} v ;
```

mas muitas vezes usa-se a construção **typedef** para lhes associar um nome. Exemplos:

```
typedef struct
{
    double re, im ;
} Complex;

typedef union
{
    int x;
    char c;
```

```
} IntChar;

typedef int Vector[5];

typedef int Matriz[2][3];

typedef char String[256];

typedef void *Pointer;

typedef int *IntPointer;

typedef IntPointer *PointerToIntPointer;

typedef int **PointerToIntPointer;

typedef (*IntFunction)(void);

typedef IntFunction IntFunctionArray[100];
```

Eis algumas declarações de variáveis usando os tipos declarados anteriormente:

```
Complex z;
IntChar u;
Vector vector;
Matriz matriz;
String str;
Pointer v;
IntPointer pt;
IntFunctionArray ops ;
```

---

# Apontadores

## Introdução: Variáveis e apontadores

Os programas processam dados armazenados na memória do computador. O mecanismo mais simples que permite aceder a essa memória são as **variáveis**. Cada variável tem um nome e um tipo e representa um pedaço da memória do computador onde podem ser guardados valores desse tipo.

Através da utilização de variáveis, é possível ir muito longe na escrita de programas em C. Mas há algumas situações em que o uso de variáveis não é suficiente e é necessário usar mecanismo mais flexível de manipulação da memória:

- Trata-se do mecanismo dos **apontadores**!

## Conceitos básicos sobre apontadores

Portanto os apontadores constituem uma segundo mecanismo de acesso à memória:

- Em vez de se usar um nome para identificar um pedaço da memória, usa-se directamente o endereço dessa zona de memória para a identificar e chama-se a esse endereço um **apontador**. Diz-se que o apontador **aponta** para uma determinada zona de memória.

Normalmente os apontadores são guardados em **variáveis de tipo apontador**.

Em C há tipos específicos para representar apontadores. O tipo dos apontadores que apontam para valores de tipo T escreve-se:

```
T *
```

Para exemplificar, eis a definição duma variável de tipo apontador para inteiro:

```
int *pt ;
```

Em C, há duas operações principais para manipular apontadores: o operador **&** permite obter um apontador para uma variável qualquer, assim

```
&Variável
```

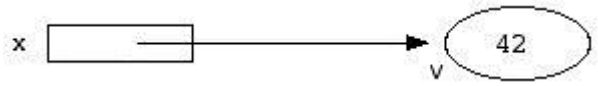
e o operador ***** permite aceder ao valor apontado por um apontador, assim:

```
*Apontador
```

Vejamos um exemplo. Abaixo, define-se uma variável inteira normal `v`. A seguir define-se uma variável de tipo apontador para inteiros `x` e fazemo-la apontar para a variável v. Depois, usamos o apontador para colocar o valor 42 na zona de memória apontada por `x`.

```
int v = 0 ;
int *x = &v ;
*x = 42
```

A seguinte figura, ilustra a situação após a atribuição do valor 42 a `*x`.

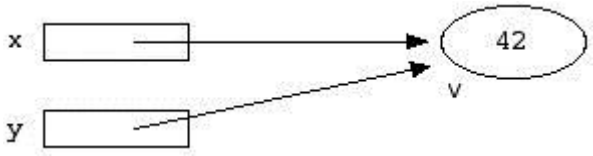



Repare que a variável `v` pode ser acedida de duas formas: (1) usando o nome `v`; (2) usando a expressão `*x`.

Para perceber melhor as possibilidades dos apontadores vamos definir agora um segundo apontador, `y` a fazê-lo também apontar para a variável `v`:

```
int *y = x ;
```

Obtém-se a seguinte situação:




Agora o conteúdo da variável `v` pode ser acedido de três formas diferentes: (1) usando o nome `v`; (2) usando a expressão `*x`; (3) usando a expressão `*y`.

O operador `&` chama-se **operador endereço**. O operador `*` chama-se **operador de desreferenciação**.

Repare na seguinte curiosa equivalência, que é válida em C para qualquer variável `v`:

```
*&v == v
```

Falta ainda uma referência à constante predefinida de tipo apontador, **`NULL`**. Garante-se que este apontador constante não aponta para sítio nenhum. Pode ser atribuído a uma variável de tipo apontador, por exemplo assim:

```
int *z = NULL ;
```

Neste caso serve para assinalar que a variável `z` não está a apontar para sítio nenhum, de momento.

O apontador NULL também pode ser usado em testes, assim:

```
if (z == NULL) ...
```

O apontador NULL não pode ser desreferenciado, pois não aponta para sítio nenhum.

# Apontadores para registos

O seguinte tipo registo permite representar datas:

```
typedef struct {
    int day, month, year ;
} Date ;
```

Vamos definir agora uma variável de tipo `Date` e coloquemos um apontador de tipo `Date *` a apontar para a primeira:

```
Date d = {25, 12, 2008};
Date *p = &d;
```

Para aceder, através do apontador, ao ano da data `d`, podemos escrever:

```
(*p).year
```

Mas a utilização de apontadores para registos em C é tão frequente, que foi criada uma notação mais compacta e sugestiva para fazer isso: o operador **->**. A seguinte expressão é equivalente à anterior:

```
p->year
```

Em geral, a seguinte notação geral permite aceder a campos de registos através de apontadores:

```
Apontador->Etiqueta
```

## Compatibilidade entre apontadores

Em C, tipos de apontadores que apontem para valores de tipos diferentes são incompatíveis entre si. Com uma excepção: o tipo especial `void *` - o tipo `void *` é compatível com todos os tipos de operadores.

```
int *pti;
double *ptd ;
void *v ;

pti = ptd ;             /* Errado */
pti = (int *)ptd ;      /* Válido por causa do cast */

v = ptd ;               /* Válido porque se trata de void * */
pti = v ;               /* Válido porque se trata de void * */
```

## Utilidade dos apontadores

Os exemplos anteriores são interessantes, mas não mostram ainda as situações práticas em que a utilização de apontadores é essencial na linguagem C. As situações práticas em que se usam apontadores em C são as seguintes:

1. **Implementação de parâmetros de saída nas funções.**
2. **Manipulação de vectores.**
3. **Manipulação de variáveis criadas dinamicamente usando a função de biblioteca `malloc`.**
4. **Programação genérica através de apontadores de tipo `void *`.**

---

# Apontadores: Parâmetros de saída de funções

Vamos tentar programar uma função para trocar o valor de duas variáveis inteiras. Este é um exemplo clássico que ilustra a necessidade de suportar **parâmetros de saída** na linguagem C.

Esta primeira tentativa não funciona:

```
void Swap(int a, int b)   /* Não funciona!!!! */
{
    int temp = a;
    a = b;
    b = temp;
}
```

A chamada de `Swap(x,y)` não muda nada, porque os parâmetros das funções são implementados usando variáveis locais, inicializadas com cópias dos valores que aparecem na chamada. A função `Swap` faz a troca das cópias locais, mas não troca o conteúdo das variáveis originais. Diz-se que os parâmetros `a` e `b` são **parâmetros de entrada**, porque eles permitem apenas transferir dados de fora para dentro da função.

Para resolver este problema, temos de usar apontadores. A função `Swap` precisa de aceder às variáveis originais através dos apontadores para efectuar a troca. Fica assim:

```
void Swap(int *a, int *b)   /* Funciona!!!! */
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Agora a chamada escreve-se `Swap(&x,&y)` e a troca é realmente efectuada. Diz-se que os parâmetros `a` e `b` são **parâmetros de saída**, porque eles permitem passar dados de dentro para fora da função.

Repare que agora ficámos a conhecer duas maneiras de fazer uma função produzir dados para o seu exterior:

1. Através do seu resultado.
2. Através de parâmetros de saída (usando apontadores).

O seguinte exemplo mostra uma função com dois *resultados*, sendo os *resultados* implementados usando parâmetros de saída. A função faz o seguinte: dado um vector de reais e o respectivo comprimento, a função calcula o máximo e o mínimo do vector, ao mesmo tempo:

```
void MaxMin(double v[], int n, double *max, double *min) /* precondição: n > 0 */
{
    double lmax = v[0];
    double lmin = v[0];
    int i;
    for( i = 1 ; i < n ; i++ ) {
        if (v[i] > lmax) lmax = v[i] ;
        if (v[i] < lmin) lmin = v[i] ;
    }
    *max = lmax;
    *min = lmin;
}
```

Exemplo de chamada:

```
double vect[] = {1.0, 2.9, 34.6, 44.2,  0.01};
double max, min;
MaxMin(vect, 5, &max, &min);
```

Algumas funções de biblioteca também usam parâmetros de saída. Por exemplo, é o caso da função de biblioteca `scanf`. Veja um exemplo de utilização:

```
scanf("%d %lf %c", &i, &r, &c);
```

---

# Apontadores: Manipulação de vetores e relação entre vectores e apontadores

Provavelmente você vai ficar surpreendido(a), mas em C o nome duma variável de tipo vector representa um apontador constante para a primeira componente do vector guardado na variável.

Considere o seguinte vector

```
int vector[100];
```

Para aceder ao primeiro elemento do vector, normalmente nós escrevemos:

```
vector[0]
```

Mas também podemos escrever o que está a seguir, pois o resultado é exactamente o mesmo.

```
*vector
```

A linguagem C também permite a seguinte atribuição:

```
int *pt = vector;
```

e, inclusivamente, permite-se a aplicação de operações sobre vectores a argumentos de tipo apontador. Por exemplo as seguintes expressões são legítimas:

```
pt[0]
pt[5]
pt[99]
```

Note que, quando de passa um vector como parâmetro para uma função, o que realmente se passa é um apontador para a primeira componente do vector. Portanto um parâmetro de tipo vector é sempre um parâmetro de saída, apesar de não ser explicitamente declarado como apontador.

# Aritmética de apontadores

Os operadores + e - podem ser aplicados a apontadores e inteiros nos seguintes casos:

- **pt + i** - o resultado é um apontador que referência um valor i posições depois de pt.
- **pt - i** - o resultado é um apontador que referência um valor i posições antes de pt.
- **pt1 - pt2** - o resultado é um inteiro que indica o numero de objectos entre pt1 e pt2. Estes dois apontadores têm de ser do mesmo tipo.

Para o vector abaixo são verdadeiras as equivalências indicadas:

```
Vector v ;

v[0] == *v
v[1] == *(v+1)
v[-1] == *(v-1)
&v[0] == v
&v[1] == v + 1
```

Fazer `v = v + 1` é proibido pois v é um apontador constante.

**Exemplo - Duas formas diferentes de copiar vectores (neste caso bidimensionais)**

```
#define SIZE    20

int i1[SIZE][SIZE], i2[SIZE][SIZE] ;
int *pt1, *pt2, *ptEnd ;
int i, j;

/* Forma 1 */
for( i = 0 ; i < SIZE ; i++ )
    for( j = 0 ; j < SIZE ; j++ )
        i2[i][j] = i1[i][j] ;

/* Forma 2 */
for( pt1 =i1 , pt2 = i2, ptEnd = pt2 + SIZE * SIZE ;
            pt2 < ptEnd ;
                *pt1++ = *pt2++ ) ;
```

Repare que a primeira forma envolve um ciclo embutido noutro e que, durante a execução, é necessário fazer muitas contas para repetidamente determinar quais as localizações correspondentes às expressões `i2[i][j]` e `i1[i][j]`.

Quanto à segunda forma, envolve um único ciclo e evita a necessidade de se fazerem as contas atrás referidas.

A segunda forma é mais difícil de perceber, mas é um pouco mais eficiente.

# Apontadores: Criação dinâmica de variáveis

A linguagem C não dispõe dum gestor de memória automático. A criação e a eliminação de variáveis dinâmicas são da responsabilidade do programador.

Usa-se a função de biblioteca `malloc` para criar e eliminar variáveis dinâmicas. A função `malloc` recebe o número de bytes do bloco de memória a criar e retorna um apontador (de tipo `void *`) para o bloco que foi criado. Eis o cabeçalho desta função, tal como está definido no módulo `stdlib`:

```
void *malloc(size_t size) ;
```

Exemplo de utilização:

```
#include <stdlib.h> /* declara a função malloc, entre muitas outras coisas */

int *a = malloc(10 * sizeof(int)) ;   /* cria var. dinâmica; o respetivo apontador é
guardado na var. a */
a[3] = 10 ;                           /* altera uma célula do bloco de memória apontado
(usando notação de vetor) */
```

A função `malloc` retorna a constante `NULL` se o gestor de memória não tiver mais memória disponível.

Para libertar a memória ocupada por uma variável dinâmica que já não irá mais ser usada, usa-se a operação:

```
void free(void *ptr) ;
```

Interessa criar variáveis dinâmicas principalmente nas duas seguintes situações:

- Quando é preciso criar um vetor, cujo tamanho só é conhecido depois do programa começar a correr.
- Para criar estruturas de dados dinâmicas, e.g. listas e árvores.

Mais abaixo, mostra-se como se manipulam com listas em C. Uma lista é uma estrutura de dados dinâmica que só pode ser manipulada através de apontadores. Para organizar bem o nosso exemplo, vamos apresentá-lo sob a forma de módulo. Mas primeiro temos de aprender a lidar módulos em C...

---

# Compilação separada e modularidade

A linguagem C suporta modularidade e compilação separada, existindo mecanismos de controlo de visibilidade de símbolos ao nível do ficheiro.

## Controlo de visibilidade

Por omissão, todas as variáveis globais e funções definidas num ficheiro são públicas, isto é podem ser acedidas a partir de outros ficheiros. Consegue-se impedir esse acesso precedendo a definição dessas entidades pela palavra `static`.

```
static Vetor privado ;
static int f(int x) { return x + 1 ; }
```

Antes de se poder aceder a uma entidade definida noutro ficheiro é preciso declarar essa entidade para que o compilador a possa conhecer. Isso faz-se usando a palavra reservada `extern`. No caso da declaração das funções o atributo `extern` pode ser omitido, pois o compilador vê que a declaração não tem corpo e assume que se trata duma entidade externa.

```
extern char key ;
extern int f(int x) ;    /* neste caso a palavra extern pode ser omitida */
```

## Módulo

Jogando de forma disciplinada com os mecanismos de visibilidade atrás descritos, é possível implementar **módulos** em C.

Um módulo "fich" é tipicamente definido usando dois ficheiros:

- Um **ficheiro de interface** "fich.h";
- um **ficheiro de implementação** "fich.c".

No ficheiro de interface definem-se todas as entidades exportadas pelo módulo, o que pode incluir: funções, variáveis globais, tipos e constantes.

O cliente do módulo só precisa fazer

```
#include "fich.h"
```

para ter acesso à definição dos símbolos exportados.

---

# Módulos abertos em C

Apresenta-se uma implementação de listas ligadas, feita num módulo aberto em C. Estas listas são homogéneas e podem conter valores de tipo `double`.

Este módulo diz-se **aberto** porque a representação das listas é pública. Veremos na secção seguinte como ocultar essa representação.

Definimos um pequeno conjunto de operações, só para exemplificar.

## Ficheiro LinkedList.h

A definição do símbolo `_LinkedList_`, protege o ficheiro de interface relativamente ao problema da inclusão múltipla do mesmo ficheiro.

Uma **lista ligada** consiste numa sequência de nós. Cada nó contém um valor dum dado tipo Data e um apontador indicando qual o nó seguinte. O valor NULL serve para indicar que não existe nó seguinte.

É interessante comparar a utilização de listas face à utilização de vetores, para guardar sequências de valores:

- A vantagem duma lista relativamente um vetor é o facto da sucessão de nós não ter de estar em posições contíguas de memória. Isso permite a inserção e remoção de valores em tempo constante.
- A desvantagem duma lista relativamente um vetor é não ser possível acesso indexado em tempo constante.

```
#ifndef _LinkedList_
#define _LinkedList_

#include <stdbool.h>

typedef double Data ;
typedef struct Node {
    Data data ;
    struct Node *next ;
} Node, *List ;  /* Uma lista e' um apontador para um no' */

List ListMakeRange(Data a, Data b) ;
int ListLength(List l) ;
bool ListGet(List l, int idx, Data *res) ;
List ListPutAtHead(List l, Data val) ;
List ListPutAtEnd(List l, Data val) ;
void ListPrint(List l) ;

#endif
```

## Ficheiro LinkedList.c

Repare que a função auxiliar `NewNode` é declarada como estática para ficar privada ao módulo. A palavra reservada `static`, quando aplicada a uma função ou a uma variável global, torna essas entidades privadas: portanto o **âmbito** da sua definição é o ficheiro onde essa definição ocorre.

Repare que todas as funções deste módulo estão programadas com base em raciocínios imperativos e sem usar recursividade. Essa é a forma normal de trabalhar em C.

Algumas destas funções são complicadas de escrever, especialmente as que requerem a utilização de diversos apontadores auxiliares. **Geralmente, a melhor forma de programar estas funções é começar por escrever o ciclo**

**principal, e só se pensa em qual deve ser a inicialização desse ciclo - realmente, antes de escrever o ciclo principal, é difícil adivinhar quais são as suas necessidades.**

A função `ListMakeRange` ilustra uma técnica importante. É sempre mais fácil fazer crescer as listas acrescentando novos nós à cabeça. Mas usando a técnica apresentada, consegue-se criar uma nova lista acrescentando os novos nós no final. Usa-se de forma habilidosa um apontador auxiliar `p` que aponta sempre para o último nó da lista.

Repara ainda que a função `ListPutAtEnd` define uma operação "destrutiva", que altera a lista original.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

static List NewNode(Data val, List next)
{
    List n = malloc(sizeof(Node)) ;
    if( n == NULL )
        return NULL ;
    n->data = val ;
    n->next = next ;
    return n ;
}

List ListMakeRange(Data a, Data b)
{
    if( a > b )
        return NULL ;
    double i ;
    List l = NewNode(a, NULL), p = l ;
    for( i = a + 1 ; i <= b ; i++ )
        p = p->next = NewNode(i, NULL) ;
    return l ;
}

/* Outra maneira, mais palavrosa, de escrever a funcao anterior:

List ListMakeRange(Data a, Data b)
{
    if( a > b )
        return NULL ;
    double i ;
    List l = NewNode(a, NULL) ;
    List p = l ;
    for( i = a + 1 ; i <= b ; i++ ) {
        List q = NewNode(i, NULL) ;
        p->next = q ;
        p = q ;
    }
    return l ;
}
*/

int ListLength(List l) {
    int count ;
    for( count = 0 ; l != NULL ; l = l->next, count++ ) ;
    return count ;
}

bool ListGet(List l, int idx, Data *res)
{
    int i ;
    for( i = 0 ; i < idx && l != NULL ; i++, l = l->next ) ;
    if( l == NULL )
        return false ;
    else {
        *res = l->data ;
```

```
        return true ;
    }
}

List ListPutAtHead(List l, Data val)
{
    return NewNode(val, l) ;
}

List ListPutAtEnd(List l, Data val)
{
    if( l == NULL )
        return NewNode(val, NULL) ;
    else {
        List p ;
        for( p = l ; p->next != NULL ; p = p->next ) ;
        p->next = NewNode(val, NULL) ;
        return l ;
    }
}

void ListPrint(List l)
{
    for( ; l != NULL ; l = l->next )
        printf("%lf\n", l->data) ;
}
```

---

# Módulos fechados em C

Os módulos fechados em C baseiam-se na possibilidade de definir um tipo apontador para um tipo estrutura que ainda não foi definido, assim:

```
typedef struct Node *List ;
```

Onde é que, finalmente, se define o tipo `struct Node`? Somente no interior ficheiro LinkedList.c.

## Ficheiro LinkedList.h

Deste ficheiro apaga-se a maior parte da definição do tipo, ficando apenas a linha que se vê. O resto do ficheiro fica igual.

```
#ifndef _LinkedList_
#define _LinkedList_

#include <stdbool.h>

typedef double Data ;
typedef struct Node *List ;

...
```

## Ficheiro LinkedList.c

A definição do tipo passa para dentro do ficheiro ".c". Mas é removida a parte "*List" pois isso já está presente no ficheiro ".h". O resto do ficheiro fica igual.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

typedef struct Node {
    Data data ;
    List next ;
} Node ;
```

...

# Recursividade e método indutivo usados no processamento de estruturas com apontadores

Eis uma nova implementação do nosso módulo, agora usando recursividade e raciocínios indutivos.

A técnica indutiva tem a vantagem de dar origem a código mais legível, mas tem a desvantagem de poder fazer crescer muito a pilha de execução a ponto de isso causar "stack overflows". Note que, ao contrário do OCaml, a linguagem C não está equipada de forma especial para suportar recursividade de forma económica.

Portanto a técnica recursiva é de evitar quando se processam lista. No processamento de listas, a complexidade espacial (tamanho da pilha de execução) dos algoritmos fica linear, como se viu numa das aulas anteriores.

Contudo, para processar árvores e outras estruturas não-lineares, já faz sentido usar recursividade em muitas das operações. Sem usar recursividade essas operações ficariam demasiado complicados. Além disso, a complexidade espacial dos algoritmos recursivos sobre árvores é muitas vezes `log N`.

```
/* ESTE É UM EXEMPLO NEGATIVO. É má ideia programar listas
   ligadas em C desta forma, apesar de funcionar */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

typedef struct Node {
    Data data ;
    List next ;
} Node ;

static List NewNode(Data val, List next)
{
    List n = malloc(sizeof(Node)) ;
    if( n == NULL )
        return NULL ;
    n->data = val ;
    n->next = next ;
    return n ;
}

List ListMakeRange(Data a, Data b)
{
    if( a > b )
        return NULL ;
    else
        return NewNode(a, ListMakeRange(a+1, b)) ;
}

int ListLength(List l)
{
    if( l == NULL )
        return 0 ;
    else
        return 1 + ListLength(l->next) ;
}

bool ListGet(List l, int idx, Data *res)
```

```
{
    if( l == NULL )
        return false ;
    else if( idx == 0 ) {
        *res = l->data ;
        return true ;
    }
    else
        return ListGet(l->next, idx-1, res) ;
}

List ListPutAtHead(List l, Data val)
{
    return NewNode(val, l) ;
}

List ListPutAtEnd(List l, Data val)
{
    if( l == NULL )
        return NewNode(val, NULL) ;
    else {
        l->next = ListPutAtEnd(l->next, val)
        return l ;
    }
}

void ListPrint(List l)
{
    if( l != NULL ) {
        printf("%lf\n", l->data) ;
        ListPrint(l->next) ;
    }
}
```

# Pré-processador

Programa que corre sempre antes do compilador de C e que faz substituição de macros, inclusão de ficheiros, e possibilita compilação condicional.

Em Linux, o pré-processador chama-se `cpp` e pode ser invocado diretamente pelo utilizador. Mas é mais prático deixar que seja o compilador de C a invocar o pré-processador automaticamente. Em todo o caso, vejamos o que diz o início do manual do comando `cpp`:

```
$ man cpp

CPP(1)                                                  GNU
CPP(1)

NAME
       cpp - The C Preprocessor

SYNOPSIS
       cpp [-Dmacro[=defn]...] [-Umacro]
           [-Idir...] [-iquotedir...]
           [-Wwarn...]
           [-M|-MM] [-MG] [-MF filename]
           [-MP] [-MQ target...]
           [-MT target...]
           [-P] [-fno-working-directory]
           [-x language] [-std=standard]
           infile outfile

       Only the most useful options are listed here; see below for the remainder.

DESCRIPTION
```

```
       The C preprocessor, often known as cpp, is a macro processor that is used
automatically by
       the C compiler to transform your program before compilation.  It is called a
macro proces-
       sor because it allows you to define macros, which are brief abbreviations for
longer con-
       structs.

       The C preprocessor is intended to be used only with C, C++, and Objective-C
source code.
```

Exemplos de macros:

```
#define max(a,b)      ((a) > (b) ? (a) : (b))
#define new(type)     ((type *)malloc(sizeof(type)))
#define not           !
#define MAX_STACK     1000

#undef max
#undef new
```

Exemplos de inclusão de ficheiros

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include "MyQueue.h"
```

Exemplos de compilação condicional

```
#define DEBUGLEVEL   1

#ifdef DEBUG
    ...
#  if DEBUGLEVEL > 2
    ...
#  else
    ...
#  endif
    ...
#else
    ...
#endif


#ifndef DEBUG
    ...
#else
    ...
#endif
```

---

# Polimorfismo implementado usando mecanismos de baixo nível

## Função monomórfica

A seguinte função permite trocar os valores de duas variáveis inteiras. É uma função monomórfica pois os seus argumentos são de tipo fixo.

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

Esta função pode ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(&x, &y) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A função `swap` é segura pois o compilador valida todas as suas utilizações.

# Polimorfismo implementado usando manipulação direta de memória

A seguinte função permite trocar os valores de duas variáveis de qualquer tipo. É uma função polimórfica pois os seus argumentos podem ser aplicados a argumentos de tipos diversos.

```
#include <string.h>

void swap(void *a, void *b, int n)
{
    char aux[n] ;     /* array criado com tamanho variável - novidade do C99 */
    memcpy(aux, a, n) ;
    memcpy(a, b, n) ;
    memcpy(b, aux, n) ;
}
```

Esta função ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(&x, &y, sizeof(int)) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A função `swap` não é segura pois o compilador não tem a possibilidade de validar as chamadas.

# Polimorfismo implementado usando macros

A seguinte macro também permite trocar os valores de duas variáveis de qualquer tipo.

```
#define swap(a,b,T)         \
    do {                    \
        T __aux = (a)  ;    \
        (a) = (b) ;         \
        (b) = __aux ;       \
    } while( 0 )
```

Esta macro ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(x, y, int) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A macro `swap` é segura pois o compilador consegue detetar qualquer erro de tipo na sua utilização. Repare que o tipo dos valores a trocar é passado como parâmetro.

Repare nos cuidados que é preciso ter para escrever uma macro que não dê problemas. Os argumentos devem ser envolvidos em parêntesis, e qualquer nova variável que seja introduzido deve ter um nome que não entre em conflito com possíveis nomes existentes. Porque todos estes cuidados? E porque razão a macro foi definida usando um `do-while`? (Elimine o `do-while`, teste, e logo descobrirá o problema.)

# Polimorfismo implementado usando manipulação direta de memória e macros

A seguinte implementação da operação `swap` é a mais agradável de usar do ponto de vista sintático. Até parece que está em causa uma operação primitiva da linguagem. Contido este implementação não é segura, pelo que o utilizador tem de ter algumas cautelas.

```
#include <string.h>

void __swap(void *a, void *b, int n)
{
    char aux[n] ;
    memcpy(aux, a, n) ;
    memcpy(a, b, n) ;
    memcpy(b, aux, n) ;
}

#define swap(a,b)    __swap(&(a), &(b), sizeof(a))
```

Esta função ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(x, y) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A macro `swap` não é segura pois o compilador não tem a possibilidade de validar as chamadas.

# Em GCC

O pré-processador do GCC possui uma construção `typeof` que permite escrever macros ainda mais sofisticadas do que as anteriores e seguras. Por exemplo, a expressão `typeof(a)` representa o tipo da variável `a` e pode ser usada como um tipo normal, inclusivamente para definir outras variáveis com o mesmo tipo de `a`.

É pena o C padrão não suportar a construção `typeof`.

# Exemplo de polimorfismo na biblioteca padrão do C - função qsort

A função **qsort** permite ordenar vetores de qualquer tipo.

```
$ man qsort

QSORT(3)                                  Linux Programmer's Manual
QSORT(3)

NAME
       qsort - sorts an array

SYNOPSIS
```

```
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int(*compar)(const void *, const void *));

DESCRIPTION
       The qsort() function sorts an array with nmemb elements of size size.  The base
argument points to the
       start of the array.

       The contents of the array are sorted in ascending order according to a comparison
function pointed  to
       by compar, which is called with two arguments that point to the objects being
compared.

       The  comparison function must return an integer less than, equal to, or greater
than zero if the first
       argument is considered to be respectively less than, equal to, or greater than
the  second.   If  two
       members compare as equal, their order in the sorted array is undefined.

RETURN VALUE
       The qsort() function returns no value.

CONFORMING TO
       SVr4, 4.3BSD, C89, C99.
```

Exercício: Como faria para ordenar um vetor com componentes do seguinte tipo:

```
typedef struct
{
    int day, month, year ;
} Date;
```

---

---

# Módulos genéricos (polimórficos) em C

É possível escrever módulos genéricos em C usando macros com várias linhas e a operação de concatenação de tokens que se escreve ##.

Os módulos genéricos são muito difíceis de escrever e de ler, mas são fáceis de usar.

No seguinte módulo, o tipo `Data` passa a ser argumento de duas macros.

## Ficheiro Generic_LinkedList.h

```
#include <stdbool.h>

#define Generic_LinkedListHeader(Data)                                  \
                                                                        \
typedef struct Data##Node {                                             \
    Data data ;                                                         \
    struct Data##Node *next;                                            \
} Data##Node, *Data##List ;                                             \
                                                                        \
int Data##ListSize(Data##List l) ;                                      \
bool Data##ListGetByIndex(Data##List l, int idx, Data *res) ;           \
Data##List Data##ListPutAtHead(Data##List l, Data val) ;                \
Data##List Data##ListPutAtEnd(Data##List l, Data val) ;                 \
void Data##ListPrint(Data##List l) ;
```

# Ficheiro Generic_LinkedList.c

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

#include "Generic_LinkedList.h"

#define Generic_LinkedListImplementation(Data, Print)           \
                                                                \
Generic_LinkedListHeader(Data)                                  \
                                                                \
static Data##List Data##NewNode(Data val, Data##List next) {    \
    Data##List n = malloc(sizeof(Data##Node)) ;                 \
    if( n == NULL )                                             \
        return NULL ;                                           \
    n->data = val ;                                             \
    n->next = next ;                                            \
    return n ;                                                  \
}                                                               \
                                                                \
int Data##ListSize(Data##List l) {                              \
    int count ;                                                 \
    for( count = 0 ; l != NULL ; l = l->next, count++ ) ;       \
    return count ;                                              \
}                                                               \
                                                                \
bool Data##ListGetByIndex(Data##List l, int idx, Data *res) {   \
    int i ;                                                     \
    for( i = 0 ; i < idx && l != NULL ; i++, l = l->next ) ;    \
    if( l == NULL )                                             \
        return false ;                                          \
    else {                                                      \
        *res = l->data ;                                        \
        return true ;                                           \
    }                                                           \
}                                                               \
                                                                \
Data##List Data##ListPutAtHead(Data##List l, Data val) {        \
    return Data##NewNode(val, l) ;                              \
}                                                               \
                                                                \
Data##List Data##ListPutAtEnd(Data##List l, Data val) {         \
    Data##List n = Data##NewNode(val, NULL) ;                   \
    if( n == NULL )                                             \
        return NULL ;                                           \
    if( l == NULL )                                             \
        return n ;                                              \
    else {                                                      \
        Data##List p ;                                          \
        for( p = l ; p->next != NULL ; p = p->next ) ;          \
        p->next = n ;                                           \
        return l ;                                              \
    }                                                           \
}                                                               \
                                                                \
void Data##ListPrint(Data##List l) {                            \
    for(  ; l != NULL ; l = l->next )                           \
        Print(l->data) ;                                        \
}
```

# Exemplo de instanciação do módulo genérico

### Ficheiro LinkedList_double.h

```
#include "Generic_LinkedList.h"
```

```
Generic_LinkedListHeader(double)
```

**Ficheiro LinkedList_double.c**

```
#include "Generic_LinkedList.c"

static doublePrint(double d) {
    printf("%lf\n", d) ;
}

Generic_LinkedListImplementation(double, doublePrint)
```

# Bibliotecas

Há linguagens de programação onde as funcionalidades específicas para manipular strings, ficheiros, etc. estão disponíveis ao nível da própria linguagem e não em bibliotecas externas. São os casos das linguagens: Fortran, Cobol e Pascal, por exemplo.

Pelo contrário, noutra linguagens essas funcionalidades estão disponíveis em bibliotecas externas e não ao nível da linguagem. São o caso das linguagens: Java, OCaml, C, C++, etc.

## Biblioteca padrão

No caso do C, existe uma pequena biblioteca padronizada, conhecida por **libc**.

A funcionalidade da biblioteca padrão está disponível através de 24 ficheiros de interface, dos quais destacamos os seguintes: <ctype.h>, <limits.h>, <locale.h>, <math.h>, <setjmp.h>. <signal.h>, <stdarg.h>, <stdbool.h>, <stdio.h>, <stdlib.h>, <string.h>, <time.h>.

A aula de hoje envolve temas que permitem a exploração de parte das funcionalidades que estão disponíveis na biblioteca padrão do C.

## Outras bibliotecas

A biblioteca padrão do C é muito útil, mas é relativamente limitada (mais limitada do que a do Java, por exemplo). Por isso alguns programas em C costumam recorrer a outras bibliotecas (tipicamente bibliotecas de código *livre*), tais como:

- C Posix library - Extensão da **libc** que permite aceder ao sistema operativo, especialmente no contexto do Unix e variantes, mas não só. O `gcc`, mesmo na versão para Windows, vem com esta biblioteca incorporada, com o nome `glibc` (GNU libc).
- GLib - Para escrever programas com interface gráfico que correm no ambiente Gnome.
- crypt - Para escrever programas que usam criptografia.
- pthread - Para escrever programas que usam threads.

Num sistema Linux observe o conteúdo das diretorias `/lib` e especialmente `/usr/lib` para ver o enorme número de bibliotecas disponível. Os ficheiros de interface correspondentes a todas essas bibliotecas são geralmente guardados na diretoria `/usr/include`.

## Usar bibliotecas

Para usar uma biblioteca é preciso primeiro incluir um ou mais ficheiros de interface no código fonte. Por exemplo:

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

```
#include <math.h>
#include <crypt.h>
#include <pthread.h>
#include <glib-2.0/glib.h>
```

Depois de escrito o programa, na altura de compilar é preciso indicar quais as bibliotecas a ligar. Usa-se para isso a opção `-l` do compilador. Exemplo:

```
cc -o myprog myprog.c mym1.c mym2.c -lm -lcrypt -lpthread -lglib-2.0
```

# Algumas funções de biblioteca

Apresentamos algumas das funções da biblioteca padrão do C. No Linux, o comando `man` permite obter a documentação sobre qualquer uma destas funções (desde que o pacote "manpages-dev" esteja instalado).

# Input/Output

```
#include <stdio.h>

#define EOF    (-1)

typedef struct{...} FILE ;

FILE *stdin, *stdout, *stderr ;

FILE *fopen(char *, char *) ;
FILE *fclose(FILE *) ;
int getc(FILE *) ;
int putc(int, FILE *) ;
int fprintf(FILE *, char *, ...) ;
int fscanf(FILE *, char *, ...) ;
int getchar(void)
int putchar(int)
int printf(char *, ...) ;
int scanf(char *, ...) ;
```

# Matemática

```
#include <math.h>

double sin(double) ;
double asin(double) ;
double acos(double) ;
double sqrt(double) ;
```

# Strings

```
#include <string.h>

int strlen(char *) ;
char *strcpy(char *, char *) ;
char *strcat(char *, char *) ;
int strcmp(char *, char *) ;
```

# Memória

```
#include <stdlib.h>

void *malloc(int) ;
void free(void *) ;
void *realloc(void *, int) ;
```

## Exemplo - Cópia de ficheiros

Exemplo que usa funções de biblioteca e ilustra a manipulação de ficheiros.

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

bool Copy(char* dest, char* orig) { /* Copia ficheiro, carácter a carácter. */
    FILE *f, *g ; /* Em C os "canais" do ML chamam-se "ficheiros internos". */
    int c ;    /* Tem de ser int porque fgetc retorna 257 valores diferentes. */

    if( (f = fopen(dest, "w")) == NULL )
        return false ;
    if( (g = fopen(orig, "r")) == NULL )
        return false ;
    while( (c = fgetc(g)) != EOF )
        fputc(c, f) ;
    fclose(f) ;
    fclose(g) ;
    return true ;
}

int main(void) {
    char in[256], out[256] ;

    printf("IN> ") ; scanf("%s", in) ;
    printf("OUT> ") ; scanf("%s", out) ;
    if( !Copy(out, in) )
        fprintf(stderr, "Copy failed!\n") ;
    return 0 ;
}
```

---

# Strings

Na linguagem C, as strings são simples vetores de caracteres e cada string é terminada pelo carácter nulo, que se escreve `'\0'`. Na seguinte inicialização de variável, a string tem um comprimento nominal de 3, mas internamente ocupa 4 bytes, por causa do terminador.

```
char *str = "ola" ;
```

O facto das strings terem todas uma marca de fim, faz com elas sejam muito flexíveis e práticas de usar. A sua flexibilidade é equivalente à dos vetores acompanhados, pois o programa pode controlar o comprimento duma string.

## Funções sobre strings

Para exemplificar a manipulação de strings, eis uma função que conta o número de ocorrências dum carácter numa string. Examine com atenção o ciclo for, porque este é típico das funções que manipulam strings.

```
int Count(char str[], char c)
{
    int i, soma = 0 ;
    for( i = 0 ; str[i] != '\0' ; i++ )
        if (str[i] == c) soma++ ;
    return soma ;
}
```

A chamada `Count("hello",'l')` produz o resultado 2.

A seguinte função acrescente um carácter no final duma string, assumindo que há espaço na string:

```
int Append(char str[], char c)
{
    int i, soma = 0 ;
    for( i = 0 ; str[i] != '\0' ; i++ ) /* procura o final da string. */
        ;
    str[i] = c ;                        /* escreve c por cima de '\0' */
```

```
    str[i+1] = '\0' ;                    /* acrescenta '\0' no final da string */
}
```

Eis outra forma de programar as mesmas duas funções, desta vez usando apontadores em vez de indexação:

```
int Count(char str[], char c)
{
    int soma = 0 ;
    char *pt ;
    for( pt = str ; *pt != '\0' ; pt++ )
        if( *pt == c ) soma++;
    return soma;
}

int Append(char str[], char c)
{
    int i, soma = 0;
    char *pt ;
    for( pt = str ; *pt != '\0' ; pt++ ) /* procura o final da string. */
        ;
    pt[0] = c;                           /* escreve c por cima de '\0' */
    pt[1] = '\0';                       /* acrescenta '\0' no final da string */
}
```

# A biblioteca padrão 'string'

A seguinte diretiva no início do nosso programa

```
#include <string.h>
```

permite ganhar acesso a numerosas funções predefinidas de manipulação de strings. Eis as funções de biblioteca mais usadas:

```
size_t strlen(const char *s) ;

char *strcat(char *dest, const char *src) ;
char *strncat(char *dest, const char *src, size_t n) ;

int strcmp(const char *s1, const char *s2) ;
int strncmp(const char *s1, const char *s2, size_t n) ;

char *strcpy(char *dest, const char *src) ;
char *strncpy(char *dest, const char *src, size_t n) ;
```

O seguinte código copia uma string:

```
char a[10], *aa = a, *b = "ola" ;
while( *aa++ = *b++ ) ;
```

Este código tem o mesmo efeito:

```
char a[10], *b = "ola" ;
strcpy(a, b) ;
```

---

# Funções com número variável de argumentos

Vamos adicionar ao módulo LinkedList uma função com número variável de argumentos para criar listas novas. A função tem de ter pelo menos um argumento com nome (neste caso `n`) com informação sobre os argumentos que se seguem (neste exemplo, `n` indica o número de argumentos); os argumentos anónimos são representados por `...`.

```
#include <stdarg.h>

List ListNew(int n, ...)
{
    va_list va ;
    List l = NULL ;
    va_start(va, n) ;
    while( n-- )
        l = ListPutAtEnd(l, va_arg(va, double)) ;
    va_end(va) ;
    return l ;
}
```

Pode ser testada usando:

```
int main(void) {
    List l = ListNew(5, 1.2, 2.6, 3.7, 4.1, 5.0) ;
    ListPrint(l) ;
    return 0 ;
}
```

Os argumentos anónimos não podem ser validados no ponto da chamada. Se, na chamada, forem passados argumentos a mais, os argumentos em excesso são ignorados pela função. Se forem passados argumentos a menos, ou argumentos de tipo errado alguns argumentos da função conterão valores indefinidos.

Para aprender mais sobre funções com número variável de argumentos, usar o comando `man stdarg`.

---

# Tratamento de erros

## Tratamento de erros gerados ao nível do sistema operativo ou das bibliotecas

Na linguagem C, muitas das chamadas a funções do sistema operativo ou a chamadas de funções de biblioteca retornam um valor especial a informar que um erro ocorreu. Esse valor deve ser testado e medidas apropriadas devem ser tomadas.

Alguns exemplos:

- A função `fopen` retorna NULL para informar que não conseguiu abrir o ficheiro.
- A função `fputc` retorna EOF para informar que não conseguiu escrever no ficheiro (provavelmente porque o disco está cheio).
- A função `malloc` retorna NULL para informar que não há mais memória disponível para criar variáveis dinâmicas.

Depois de detetado o erro é possível obter mais informação sobre este usando o módulo da biblioteca padrão `errno`. Ao incluirmos o ficheiro de interface <errno.h> ganhamos acesso a uma variável inteira chamada `errno`. Após um erro de sistema, essa variável contém sempre um código que indica qual a razão exata do erro. O valor `0` significa que não há erro. Exemplo:

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

void Error(char *errmesg)
{
    fprintf(stderr, "%s!\n", errmesg) ;
    exit(1) ;
}

FILE *OpenFile(char *name) {
    FILE *f ;
    if( (f = fopen(name, "w")) == NULL ) {
        switch( errno ) {
            case EMFILE: Error("Too many open files") ;
            case ENAMETOOLONG: Error("Filename too long") ;
            case ENFILE: Error("Too many open files in system") ;
            case ENOENT: Error("No such file") ;
            case EROFS: Error("Read-only file system") ;
            default: Error("File error") ;
        }
    }
    return f ;
}

int main(void) {
```

```
    FILE *f = OpenFile("ola") ;
    /* mais código ... */
    fclose(f) ;
    return 0 ;
}
```

Este esquema de tratamento de erros é bastante mau pois obriga a misturar o tratamento de erros com a lógica normal do programa.

# Saltos não-locais

A linguagem C não dispõe de qualquer mecanismo de alto-nível para tratamento exceções. Contudo a biblioteca padrão contém um módulo chamado `setjmp` que fornece um mecanismo de saltos não locais que permite tratar exceções a baixo nível.

O módulo `setjmp` disponibiliza as funções `setjmp` e `longjmp`:

- A função `setjmp` guarda numa variável de tipo `jmp_buf` parte do estado da execução do programa, incluindo a indicação de qual é o registo de ativação corrente e quais os valores dos diversos registos do processador, incluindo o program counter. Quando a função `setjmp` é chamada, ela retorna o valor `0`.
- A função `longjmp` repõe o estado da máquina, o que implica um "regresso ao passado" e força `setjmp` a retornar outra vez, desta vez com o valor passado em `longjmp`, um valor que deve ser diferente de `0`. Diz-se que isto é um "salto não-local", porque de certa maneira, a função `setjmp` causa um salto para dentro da função `jmp_buf`.

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

static jmp_buf jbuf ;

FILE *OpenFile(char *name) {
    FILE *f ;
    if( (f = fopen(name, "w")) == NULL )
        longjmp(jbuf, errno) ;
    return f ;
}

int main(void) {
    switch( setjmp(jbuf) ) {
        case 0: {   /* código protegido */
            FILE *f = OpenFile("ola") ;
            /* mais código ... */
            fclose(f) ;
            break ;
        }
        case EMFILE: Error("Too many open files") ;
        case ENAMETOOLONG: Error("Filename too long") ;
        case ENFILE: Error("Too many open files in system") ;
        case ENOENT: Error("No such file") ;
        case EROFS: Error("Read-only file system") ;
        default: Error("File error") ;
    }
    return 0 ;
}
```

Comparando com o OCaml, repare que em C a função `longjmp` desempenha o mesmo papel da expressão `raise`. A função `setjmp`, juntamente com o switch envolvente, desempenha o mesmo papel da expressão `try-with`.

# Tratamento de erros gerados ao nível do hardware

Para apanhar erros do género `divisão-por-zero`, a aplicação tem de instalar rotinas de serviço para detetar determinadas interrupções geradas pelo hardware. Para isso é necessário usar os serviços do módulo `signal` da biblioteca padrão.

O seguinte exemplo ilustra a utilização da função `signal` e apanha todas as interrupções geradas a partir do teclado usando CTRL-C.

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

static void InterruptHandler(int sig)
{
    if( sig == SIGINT ) {
        signal(SIGINT, SIG_IGN) ;
        printf("Interrupt\n") ;
        signal(SIGINT, InterruptHandler) ;
    }
}

int main(void) {
    int i ;
    signal(SIGINT, InterruptHandler) ;
    for( i = 0 ; i < 1000000000 ; i++ )
        if( i % 100000000 == 0 )
            printf("%d\n", i) ;
   return 0 ;
}
```

É possível fazer um `longjmp` para fora duma rotina de serviço de interrupções? O padrão não dá garantias sobre o assunto e geralmente não funciona mesmo.

Em muitas implementações de C (e.g. GCC) o módulo `setjmp` inclui funções extra (`sigsetjmp` e `siglongjmp`) que permitem fazer isso. Mas esta funcionalidade faz parte do padrão da biblioteca C. Faz sim parte do padrão POSIX para bibliotecas de acesso aos serviços do sistema operativo.

---

# Output formatado

O output formatado em C é gerado usando funções da família `printf`. A função original escreve o output no canal de saída padrão, mas há uma variante que escreve num ficheiro de saída qualquer e outra variante que escreve numa string. Estas funções aceitam um número variável de argumentos.

```
#include <stdio.h>
int printf(const char *format, ...) ;
int fprintf(FILE *out_stream, const char *format, ...) ;
int sprintf(char *out_str, const char *format, ...) ;

#include <stdarg.h>
int vprintf(const char *format, va_list ap) ;
int vfprintf(FILE *stream, const char *format, va_list ap) ;
int vsprintf(char *str, const char *format, va_list ap) ;
```

A **string de formatação** é quase toda copiada literalmente para o output. A exceção são os **especificadores de formato**, começados pelo carácter especial '%', que provocam a escrita dos parâmetros que estiverem colocados após a strings de formatação. Eis um exemplo, seguido do respetivo output:

```
#include <stdio.h>

int main(void) {
   printf("Characters: %c %c\n", 'a', 67) ;
   printf("Decimals: %d %ld\n", 1977, 650000) ;
   printf("Preceding with blanks: %10d\n", 9999) ;
   printf("Preceding with zeros: %010d\n", 9999) ;
   printf("Some different radixes: %d %x %o %#x %#o\n", 100, 100, 100, 100, 100);
   printf("floats: %4.2f %+.0e%E\n", 3.14159, 3.14159, 3.14159) ;
```

```
   printf("Width trick: %*d\n", 5, 10) ;
   printf("% \n", "Hello world!") ;
   return 0 ;
}

Characters: a C
Decimals: 1977 650000
Preceding with blanks:       9999
Preceding with zeros: 0000009999
Some different radixes: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141590E+000
Width trick:    10
Hello world!
```

A função `printf` retorna o número de caracteres escritos. Em caso de erro retorna um valor negativo.

**printf em C++**

Em C++ existem outras primitivas de output baseadas no operador "<<", mas a operação `printf` também está disponível.

**printf em Java**

A operação `printf` é tão popular, que foi incorporada na biblioteca do Java. Em Java, os especificadores de formato são ainda em maior número do no C.

```
System.out.printf("%d\n", 123) ;

123
```

**printf em OCaml**

A operação `printf` também está disponível na biblioteca do OCaml:

```
# Printf.printf "%d\n" 123 ;;
123
- : unit = ()
```

---

# Input formatado

O input formatado em C é lido usando funções da família `scanf`. A função original lê o input no canal de entrada padrão, mas há uma variante que lê num ficheiro de entrada qualquer e outra variante que lê duma string. Estas funções aceitam um número variável de argumentos.

```
#include <stdio.h>
int scanf(const char *format, ...) ;
int fscanf(FILE *stream, const char *format, ...) ;
int sscanf(const char *str, const char *format, ...) ;

#include <stdarg.h>
int vscanf(const char *format, va_list ap) ;
int vsscanf(const char *str, const char *format, va_list ap) ;
int vfscanf(FILE *stream, const char *format, va_list ap) ;
```

A **string de formatação** é usada de forma literal para fazer emparelhamento com o input, mas há duas exceções: (1) os caracteres brancos (' ', '\t', '\n') emparelham com qualquer sequência de brancos exceção; (2) os **especificadores de formato**, começados pelo carácter especial '%', que causam a leitura de valores para os parâmetros que estiverem colocados após a strings de formatação.

Note que todos os parâmetros após a strings de formatação são parâmetros de saída, portanto são implementados usando apontadores. Eis um exemplo:

```
#include <stdio.h>

int main (void) {
```

```
    char str[100];
    int i;

    printf("Enter name: ") ;
    scanf("%s", str) ;
    printf("Enter age: ") ;
    scanf("%d", &i);
    printf("%s is %d years old.\n", str,i) ;
    printf("Enter hexadecimal number: ") ;
    scanf("%x", &i);
    printf("Value %#x (%d).\n", i, i) ;
    return 0;
}
```

A função `scanf` retorna o número de parâmetros lidos com sucesso; em caso de falhanço de emparelhamento, esse valor pode ser inferior ao esperado. Antes do primeiro argumento ter sido lido, caso o input termine subitamente durante emparelhamento que esteja a ser bem sucedido, então é retornado o valor EOF.

**scanf em C++**

Em C++ existem outras primitivas de input baseadas no operador ">>", mas a operação `scanf` também está disponível.

**scanf não existe em Java**

A operação `scanf` não existe na biblioteca do Java, mas as classes `Scanner` e `Pattern` oferecem uma solução ainda mais complete e sofisticada.

**scanf em OCaml**

A operação `scanf` também está disponível na biblioteca do OCaml:

```
# Scanf.scanf "%d" (fun x->x) ;;
123
- : int = 123
```

---

---

# Funções em C

Quando se define uma função é preciso indicar o seu nome, argumentos e tipo do resultado. Um exemplo:

```
int Sum(int a, int b)
{
    return a + b ;
}
```

As funções podem produzir efeitos laterais. Se o tipo do resultado for **void** isso significa que a função não produz resultado e que essa função só produz efeitos laterais. Exemplo:

```
void OutN(int n)
{
    while( n--)
        printf("%d\n", n) ;
}
```

## Parâmetros

Nas funções, a linguagem C suporta apenas passagem de parâmetros por valor. Se quisermos definir funções com parâmetros de saída é necessário passar apontadores como argumentos (isto já foi estudado na secção sobre apontadores da aula passada). Eis um exemplo de função void com um parâmetro de saída, em que esse parâmetro é usado de forma sofisticada em diversos pontos do corpo da função - estude tente perceber o exemplo completamente:

```
void factorial(int i, int *r)
{
    if( i == 0 )
```

```
        *r = 1 ;
    else {
        factorial(i-1, r) ;
        *r *= i ;
    }
}

int main(void)
{
    int arg = 10, res ;
    factorial(arg, &res) ;  /* passa uma referencia para a variável x */
    printf("fact(%d)=%d\n", arg, res) ;
    return 0 ;
}
```

Os parâmetros vetor são um caso especial pois são tratados como apontadores constantes.

```
int setArray(int array[], int index, int value)
{
    array[index] = value;
}

int main(void)
{
    int  a[1] = {1} ;
    setArray(a, 0, 2) ;
    printf ("a[0]=%d\n", a[0]) ;
    return 0;
}
```

Antes da norma C89 os registos e as uniões não podiam ser passadas como parâmetro; só se permitia passar apontadores para eles.

# Validação dos parâmetros

Antes da norma C89 não era feita verificação do número e tipo dos argumentos no ponto da chamada das funções. O programa compilava e tudo parecia bem até o programa começar a correr!?

Atualmente, a verificação é devidamente feita e o programador deve garantir que o cabeçalho (protótipo) de cada função é conhecido no ponto da chamada, senão o programa torna-se inválido. No seguinte exemplo, se retirarmos o protótipo, o programa deixa de poder ser compilado:

```
#include <stdio.h>

void g(double d) ;    /* protótipo */

void f(void)
{
    g(1) ;
}

void g(double d)
{
    printf("%5.5lf\n", d) ;
}

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

As funções com zero argumentos devem ser definidas com a palavra void na posição dos argumentos, tal como na função f anterior. Quando a lista de argumentos é vazia isso significa que a função pode ser chamada com qualquer número de argumentos, ou seja que a chamada da função não é validada.

# Função main

A função main, onde o programa começa a correr, pode ser escrita usando qualquer dos três cabeçalhos seguintes:

```
int main(void)

int main()

int main(int argc, char *argv[])
```

## Parâmetros funcionais

Usando apontadores para funções, em C é possível escrever funções com parâmetros funcionais. Para adicionar ao módulo LinkedList uma função `ListSearch` para procurar o primeiro valor da lista com uma propriedade dada, faz-se assim. A propriedade é especificada usando uma função booleana.

```
bool ListSearch(List l, bool (*f)(Data), Data *res)
{
    for( ; l != NULL ; l = l->next )
        if( (*f)(l->data) ) {
            *res = l->data ;
            return true ;
        }
    return false ;
}
```

Pode ser testada assim:

```
bool MyF(Data d) {
    return d < 0 ;
}

int main(void) {
    Data d ;
    List l = ListNew(5, 1.2, 2.6, 3.7, -47.1, 5.0) ;
    if( ListSearch(l, &MyF, &d) )
        printf("---> %lf\n", d) ;
    else
        printf("No found\n") ;
    return 0 ;
}
```

---

---

# Inseguranças da Linguagem C

A linguagem C contém diversas inseguranças:

- Não validação dos limites nos acessos a arrays;
- Não validação dos argumentos em chamadas de funções com um número variável de argumentos;
- Permite manipulação de memória a baixo nível através de apontadores;
- As regras de conversão automática de tipo fazem com que todas quase todas as expressões estruturalmente válidas sejam aceites;
- Nas atribuições pode haver perda de precisão dos valores.

## Ferramentas que ajudam a reduzir a insegurança

Para detetar problemas, chamar o compilador de C com todos os warnings ligados ajuda (cc -Wall), mas em geral recomenda-se a utilização duma ferramenta especializada para detetar código duvidoso, antes de o submeter ao compilador de C.

Em 1979 foi criado um verificador chamado **lint** que passou a ser distribuído com todas as versões do Unix.

Uma versão melhorada, atualmente em uso, chama-se **splint**:

```
$ man splint
splint(1)
splint(1)

NAME
       splint - A tool for statically checking C programs

SYNOPSIS
       splint [options]

DESCRIPTION
       Splint  is a tool for statically checking C programs for security vulnerabilities
and com-
       mon programming mistakes. With minimal effort, Splint can be used as a  better
lint(1).If
       additional  effort is invested adding annotations to programs, Splint can perform
stronger
       checks than can be done  by  any  standard  lint.   For  full  documentation,
please  see
       http://www.splint.org.  This man page only covers a few of the available options.
```

Uma situação dramática em C e C++ é quando, numa fase adiantada de desenvolvimento, um programa grande começa a rebentar com problemas de acesso à memória. Quando não há a mínima ideia sobre qual possa ser a origem do problema, a ferramenta **valgrind** pode ser a salvação. Permite executar o programa executável dentro duma máquina virtual que valida todos os acessos à memória, conseguindo detetar:

- Acessos a variáveis não inicializadas,
- Acessos a blocos de memória fora dos seus limites,
- Acessos a blocos de memória que já não estão em uso por se ter feito o `free` deles,
- Outros tipos de usos errados das funções do módulo `malloc`.

```
VALGRIND(1)                                Release 3.4.0
VALGRIND(1)

NAME
       valgrind - a suite of tools for debugging and profiling programs

SYNOPSIS
       valgrind [[valgrind] [options]] [your-program] [[your-program-options]]

DESCRIPTION
       Valgrind is a flexible program for debugging and profiling Linux executables. It
consists
       of a core, which provides a synthetic CPU in software, and a series of "tools",
each of
       which is a debugging or profiling tool. The architecture is modular, so that new
tools can
       be created easily and without disturbing the existing structure.

       This manual page covers only basic usage and options. For more comprehensive
information,
       please see the HTML documentation on your system:
/usr/share/doc/valgrind/html/index.html,
       or online: http://www.valgrind.org/docs/manual/index.html.

INVOCATION
       Valgrind is typically invoked as follows:

           valgrind program args
```

Para usar o Valgring, faça assim: Compile o programa da seguinte forma:

```
gcc -g -O0 -o prog prog.c
```

e depois corra o programa assim:

```
valgrind ./prog
```