Linguagens e Ambientes de Programação (2018/2019)

Teórica 15 (06/mai/2019)

Bibliotecas. Biblioteca padrão do C.
Tratamento de erros.
Funções com número variável de argumentos.
Input/Output.
Strings.
Output formatado.
Input formatado.

Funções em C.
O qualificador de tipo const.
Portabilidade - Independência da máquina.
Falhas de proteção da linguagem C.

Bibliotecas

Em diferentes linguagens de programação, existem diferentes filosofias relativamente à inclusão de funcionalidades específicas na linguagem.

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.

Esta aula vai ser integralmente dedicada a estudar algumas das funcionalidades da linguagem C que foram remetidas para a sua biblioteca.

Biblioteca padrão

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

A funcionalidade da biblioteca padrão está disponível através de um pouco mais de duas dezenas 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>, <thread.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 password codificadas.
- 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 <thread.h>
#include <crypt.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 -lthread -lcrypt -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

Essas operações funcionam com ficheiros de texto e ficheiros binários, exceto as últimas quatro operações que funcionam apenas com ficheiros de texto.

```
#include <stdio.h>

#define EOF    (-1)

typedef struct{...} FILE;

FILE *stdin, *stdout, *stderr;

FILE *fopen(char *, char *);
FILE *fclose(FILE *);
int fgetc(FILE *);
int fputc(int, FILE *);
int getchar(void);
int putchar(int);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

int fprintf(FILE *, char *, ...);
int fscanf(FILE *, char *, ...);
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);
```

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 de 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 entre funções.

```
#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.

Input/Output

Exemplo 1 - Cópia de ficheiro (de texto ou binário) byte a byte.

Note que estamos a abrir os dois ficheiros usando um modo "b" especial que indica processamento de dados binários e não de texto. Num processamento em modo "texto", podem ocorrer conversões de caracteres especiais. Por exemplo, no Windows, em modo "texto", a escrita de "\n" gera "\r\n" no ficheiro e a leitura de "\r\n" resulta num simples "\n".

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

bool copyFile(char* dest, char* orig) { /* Copia ficheiro, byte a byte. */
    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(orig, "rb")) == NULL )
        return false;
    if( (g = fopen(dest, "wb")) == NULL )
        return false;

    while( (c = fgetc(f)) != EOF )
        fputc(c, g);

    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( !copyFile(out, in) )
        fprintf(stderr, "Copy failed!\n");
    return 0;
}
```

Exemplo 2 - Cópia de ficheiro de texto linha a linha.

Repare que aqui deixámos de usar o "b".

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

#define MAX_LINE   1024
typedef char Line[MAX_LINE];

bool copyFile(char* dest, char* orig) { /* Copia ficheiro, linha a linha. */
    FILE *f, *g;
    Line s;

    if( (f = fopen(orig, "r")) == NULL )
        return false;
    if( (g = fopen(dest, "w")) == NULL )
        return false;

    while( fgets(s, MAX_LINE, f) != NULL ) // Lê e copia uma linha de cada vez
        fputs(s, g);

    fclose(f);
```

```
    fclose(g);
    return true;
}
```

Exemplo 3 - Cópia de ficheiro (de texto ou binário) em blocos de 1024 bytes.

Aqui usa-se o "b".

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

#define BUFFER_SIZE    1024

bool copyFile(char* dest, char* orig) { /* Copia ficheiro, bloco a bloco. */
    FILE *f, *g;
    char buffer[BUFFER_SIZE];
    int count;

    if( (f = fopen(orig, "rb")) == NULL )
        return false;
    if( (g = fopen(dest, "wb")) == NULL )
        return false;

    while( (count = fread(buffer, 1, BUFFER_SIZE, f)) > 0 ) {
        printf("%d\n", count);
        fwrite(buffer, 1, count, g);
    }
        printf("%d\n", count);

    fclose(f);
    fclose(g);
    return true;
}
```

Strings

Na linguagem C, as strings são simples vetores de carateres e cada string é terminada pelo caráter 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 caráter 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 caráter 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);
```

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 caráter 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 carateres 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 (forma exata) para fazer emparelhamento com o input, mas há duas exceções: (1) os carateres brancos (' ', '\t', '\n') emparelham com qualquer sequência de brancos; (2) os **especificadores de formato**, começados pelo caráter especial '%', causam a leitura de valores para os argumentos que estiverem colocados após a string de formatação.

Note que todos os argumentos que se seguem à string de formatação são argumentos de saída, sendo portanto 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; portanto, 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 nome da função, o tipo e nome dos argumentos e o tipo do resultado. Um exemplo:

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

As funções podem produzir efeitos laterais para além de calcular um resultado.

No caso extremo duma função que retorne **void**, ela não produz qualquer resultado e apenas efeitos laterais. Exemplo:

```
void outN(int n) {
    while( n-- > 0 )
        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 (por valor, claro) apontadores como argumentos (isto já foi estudado na secção sobre apontadores, numa aula passada).

Eis mais um exemplo duma função com parâmetro de saída, em que esse parâmetro é usado de forma sofisticada em diversos pontos do corpo da função - tente perceber bem o exemplo, pois ele está cheio de detalhes subtis e instrutivos:

```
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 ;
}
```

Já sabemos duma aula anterior que o nome dum vetor é sempre um apontador constante para a sua primeira posição. Este facto determina o que acontece quando se passa um vetor como argumento para uma função. O que realmente é passado é o tal apontador. Logo, um argumento-vetor é um parâmetro de saída (ou se entrada-saída, conforme a forma como for usado).

Outra consequência dos vetores serem simplesmente passados como um apontadores é o facto da função não saber o tamanho do vetor recebido. Por causa disso, o programador geralmente passa o tamanho do vetor num argumento suplementar. Repare que isto até é uma vantagem, pois assim a mesma função consegue processar vetores de diferentes tamanhos.

```
void write_values(int vector[], int n)
{
    for( int i = 0 ; i < n ; i++ )
        printf("vector[%d] = %d", i, vector[i]);
}

int main(void) {
    int  a[] = {1;2;3;4;5;6;7;8;9;10} ;
```

```
    write_values(a, 10);
    return 0;
}
```

Antes da norma C89, valores de tipos registo ou tipos união não podiam ser passados diretamente como argumentos de funções; só se permitia passar apontadores para eles.

Validação dos parâmetros das funções

Antes da norma C89, não era feita verificação do número e tipo dos argumentos nas chamadas das funções. O programa compilava e tudo parecia estar bem até o programa começar a correr!?

No C89, a verificação do número e tipo dos argumentos passou a ser possível, embora não obrigatória. Para validar todas as chamadas duma função, o programador precisa de garantir que o cabeçalho dessa função é conhecido nos ponto da chamada. No caso de chamada para a frente ou chamadas a partir de outros módulos, é preciso que apareça o respetivo protótipo (cabeçalho de função, sem corpo) antes da chamada.

No C99, a verificação do número e tipo dos argumentos passou a ser obrigatória, ou seja passou a ser obrigatorio escrever protótipos nas chamadas para a frente ou chamadas a partir de outros módulos.

No seguinte exemplo, se retirarmos o protótipo, o programa deixa de poder ser compilado num compilador que seja estritamente C99:

```
#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 ;
}
```

Funções sem argumentos

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

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 ;
}
```

Mas note que a possibilidade de passar função por parâmetro não é tão poderosa quanto se poderia esperar, porque em C não existe aninhamento de função, mas só funções globais.

O qualificador de tipo const

O qualificador **const** pode ser usado nos tipos da linguagem C para indicar que determinadas variáveis ou determinadas zonas de memória não podem ser alterados depois de inicializado.

Há duas vantagens em usar este qualificador:

- Aumenta a segurança dos programas, permitindo ao compilador detetar algumas escritas inválidas na memória, antes do programa correr;
- Os programas tornam-se mais informativos para quem os lê.

O qualificador **const** é de utilização muito flexível. Veja os seguintes exemplos:

```
int main(void) {
    const int i = 3 ;      // Constante inteira
    const int *pt ;        // Apontador para inteiro constante
    int const *pt ;        // Apontador para inteiro constante
    int *const pt ;        // Apontador constante para inteiro
    int const *const pt ;  // Apontador constante para inteiro constante
    int *const *pt ;       // Apontador para apontador constante para inteiro
}
```

A leitura destes tipos nem sempre é simples. Veja as regras:

- Quando **const** é a primeira palavra do tipo, o primeiro elemento desse tipo é considerado constante.
- Quando **const** não é a primeira palavra do tipo, o que ocorrer imediatamente à esquerda de const é considerado constante.

Um caso particular importante: se um argumento duma função tem o atributo **const** (e.g. "const int a") isso significa que se trata dum parâmetro de entrada que não pode ser modificado no corpo da função.

Portabilidade - Independência da máquina

Apesar da possibilidade de usar diretamente os recursos duma máquina, a linguagem C não está comprometida com nenhuma arquitetura particular e até encoraja a escrita de programas portáveis, ou seja programas que funcionam em qualquer máquina.

Mas a portabilidade não se obtém automaticamente e o programador tem de se preocupar com essa questão.

Problema 1: tamanho dos valores

O tamanho dos valores de diversos tipos variam de implementação para implementação e a norma da linguagem especifica apenas valores mínimos. Por exemplo, uma variável de tipo `int` precisa de ter um mínimo de 16 bits (mas tem muitas vezes 32 bits) e uma variável de tipo `long` precisa de ter um mínimo de 32 bits (mas tem muitas vezes 64 bits).

Nunca se deve assumir que um inteiro ou um long tem um tamanho particular. Se for preciso conhecer o tamanho, então ele deve ser obtido através da função `sizeof`.

Para não se ter de pensar no tamanho dos valores, por vezes usam-se os seguintes tipos de biblioteca:

- **size_t** - Tipo inteiro usado para representar o tamanho duma variável ou tipo, retornado por `sizeof`.
- **ptrdiff_t** - Tipo inteiro usado para representar a diferença entre dois apontadores que referem partes diferentes do mesmo array.
- **off_t** - Tipo inteiro usado para representar um offset num ficheiro, retornado pela função `lseek`.
- **ssize_t** - Tipo inteiro usado para representar o número de bytes lidos ou escritos pelas funções `read` ou `write`.

Em todo o caso, sempre que o mínimo de 16 bits for suficiente, deve usar-se o tipo `int`, pois é normalmente o tipo inteiros mais eficiente da máquina.

O C99 introduziu um módulo com header `stdint.h` para ajudar nas situações em que precisarmos de inteiros com um número de bits conhecido à partida. Nesse ficheiro estão definidos os tipos `int8_t`, `int16_t`, `int32_t`, `int64_t`. Se precisarmos dum inteiro com tamanho suficiente para guardar um apontador, podemos usar o tipo `intptr_t`.

Problema 2: Arrumação dos campos nos registos

A arrumação dos campos num registo é dependente do compilador. Quando se enviam registos através da rede, dum programa para outro, convém implementar um mecanismo de serialização.

Problema 3: ordem dos bytes (Big endian/Little endian)

Os diversos bytes que constituem um inteiro podem ser guardados por duas ordens diferentes, consoante arquitetura do computador. Quando se enviam valores através da rede, convém usar um mecanismo de serialização que leve isso em conta.

Falhas de proteção da linguagem C

A linguagem C tem diversas falhas de proteção relativamente a erros de tipo:

- Não se validam os argumentos em chamadas de funções com um número variável de argumentos;
- Através duma `union`, um valor dum dado tipo pode ser visto como se tivesse outro tipo;
- Através de *casts* de apontadores, um valor dum dado tipo pode ser visto como se tivesse outro tipo.

A linguagem C também tem diversas falhas de proteção relativamente a erros de lógica básicos:

- Leitura de variáveis não inicializadas;
- As regras de conversão automática de tipo fazem com que todas quase todas as expressões estruturalmente válidas sejam aceites (ou seja, o programador escreve uma expressão sem sentido para ele, mas a linguagem atribui sempre um sentido a essa expressão);
- Nas atribuições pode haver perda de precisão dos valores;
- Não se validam os limites nos acessos a arrays;
- Os apontadores permitem manipulação irrestrita de memória;
- Na gestão das variáveis dinâmicas, possibilidade de *dangling pointers* e *memory leaks*.

Ferramentas que ajudam a aumentar a proteção em C

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

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
```

Exercício: Considere o seguinte programa em C. Este programa não tem erros sintáticos e portanto compila. No entanto tem diversos erros de lógica que os programadores fazem frequentemente.

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

void unnitialised(void) {
    printf("unnitialised start\n") ;
    int k ;
    printf("k = %d\n", k) ;
    printf("unnitialised end\n") ;
}

void outofbounds(void) {
    printf("outofbounds start\n") ;
    int i ;
    int *vect = malloc(10 * sizeof(int)) ;
    for( i = 0 ; i < 11 ; i++ )
        vect[i] = 12345 ;
    free(vect) ;
    printf("outofbounds end\n") ;
}

void invalidfree(void) {
    printf("invalidfree start\n") ;
    int *pt = (int *)&pt ;
    free(pt) ;
    printf("invalidfree end\n") ;
}

int main(void) {
    unnitialised() ;
    outofbounds() ;
    invalidfree() ;
    return 0 ;
}
```

- 1 - Depois descubra e explique todos os erros deste programa.
- 2 - Use a ferramenta Valgrind para descobrir automaticamente os erros anteriores.

#80