

# Sistemas de Operação (2018/2019)

## Ficha 1

**Q1.** Considere o programa `hello.c`:

```
#include <stdio.h>

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

Veja o que acontece ao código quando corre apenas o pré-processador de C:

```
$ gcc -E hello.c
```

Para inspecionar o código assembly Intel x86 produzido para o programa pode fazer:

```
$ gcc -S hello.c
```

e procure um ficheiro com o nome `hello.s`.

Execute os seguintes comandos e observe o que acontece:

```
$ gcc hello.c
$ gcc -o hello hello.c
$ gcc -Wall -o hello hello.c
```

Sempre que compile um programa em C deverá utilizar a opção `-Wall` para que o compilador sinalize todas as situações (*warnings*) que, não sendo impeditivas da geração de um ficheiro binário executável, poderão indiciar problemas. Deve tratá-los como se de erros de compilação se tratassem.

Para utilizar um *debugger*, um programa auxiliar que permite a execução passo a passo do binário executável, indicando a instrução no programa fonte a ser executada, deverá compilar com a opção `-g`. Experimente:

```
$ gcc -g -o hello hello.c
$ gdb hello
gdb> break main
gdb> run
gdb> next
gdb> ...
```

**Q2.** Considere o seguinte programa `trig.c` que pré-calcula as funções trigonométricas  $\sin x$  e  $\cos x$  para ângulos inteiros em graus entre 0 e 360 (evitando assim chamar posteriormente as funções respectivas da biblioteca matemática).

```
#include <stdio.h>

#define START      0
#define ONE_TURN   360

double cos_table[ONE_TURN];
double sin_table[ONE_TURN];

void build_tables() {
    int i;
    for (i = START; i < ONE_TURN; i++) {
        sin_table[i] = sin(M_PI * i / 180.0);
        cos_table[i] = cos(M_PI * i / 180.0);
    }
}

double sin_degrees(int angle) {
    return sin_table[angle % ONE_TURN];
}

double cos_degrees(int angle) {
    return cos_table[angle % ONE_TURN];
}

int main() {
    build_tables();
    printf("sin(20) = %f\n", sin_degrees(20));
    printf("cos(80) = %f\n", cos_degrees(425));
    printf("tan(60) = %f\n", sin_degrees(60) / cos_degrees(60));
    return 0;
}
```

Compile o programa com o comando: `gcc -Wall -o trig trig.c`. O compilador queixa-se de algo? Consegue perceber do quê? (nota importante: leia as mensagens de erro com atenção).

Corrija o erro e volte a compilar o programa com o mesmo comando. O compilador volta a queixar-se? Desta vez qual é o problema? Como o pode resolver? (sugestão: faça `man sin` ou `man cos`).

**Q3.** Considere o seguinte programa, `pointers1.c`, que pretende exemplificar alguns aspectos da utilização de apontadores, nomeadamente dos operadores `&` (endereço de) e `*` (conteúdo de endereço).

```
int main() {
    int i, j, *p, *q;
    i = 5;
    p = &i;
    *p = 7;
    j = 3;
    p = &j;
    q = p;
    p = &i;
    *q = 2;
    return 0;
}
```

Compile-o com o comando: `gcc -Wall -o pointers1 pointers1.c` e veja o que acontece às variáveis acrescentando a linha seguinte em diferentes pontos do programa:

```
printf("i=%d, j=%d, p=%p, q=%p\n", i, j, p, q);
```

Faça um desenho representando a memória do sistema e represente a criação das variáveis `i`, `j`, `p` e `q` e siga o resto do programa alterando os valores das mesmas no dito desenho.

**Q4.** Considere o programa `char_array.c` que percorre um array de caracteres:

```
#include <stdio.h>

int main() {
    int i;
    char msg[] = "Hello World";
    for (i = 0; i < sizeof(msg); i++) {
        printf("%c <--> %c\n", msg[i], *(msg + i));
    }
    return 0;
}
```

Compile-o e execute-o? Como explica o resultado? A variável `msg` comporta-se como se tivesse que tipo? Cada incremento de `i` corresponde a quantos bytes?

Considere agora o programa `int_array.c`.

```
#include <stdio.h>

int main() {
    int i;
    int primes[] = {2, 3, 5, 7, 11};
    for (i = 0; i < sizeof(primes)/sizeof(int); i++) {
        printf("%d <--> %d\n", primes[i], *(primes + i));
    }
    return 0;
}
```

Compile-o e execute-o? Como explica o resultado? A variável `primes` comporta-se como se tivesse que tipo? Cada incremento de `i` corresponde a quantos bytes?

**Q5.** Considere os seguintes programas, `call_by_value.c`:

```
void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
}

int main() {
    int n1 = 1;
    int n2 = 2;
    swap(n1, n2);
    printf("n1: %d n2: %d\n", n1, n2);
    return 0;
}
```

e `call_by_reference.c`:

```
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main() {
    int n1 = 1;
    int n2 = 2;
    swap(&n1, &n2);
    printf("n1: %d n2: %d\n", n1, n2);
    return 0;
}
```

Faça um desenho representando a memória do sistema e represente a criação das variáveis `n1`, `n2`, `p1` e `p2` e siga o resto do programa alterando os valores das mesmas no dito desenho.

Consegue perceber a diferença entre os dois programas? Porque é que no segundo os valores de `n1` e `n2` são trocados, ao contrário do que acontece com no primeiro caso?

**Q6.** Considere os programas `bad_pointer.c`:

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

int* get_int() {
    int i = 2;
    return &i;
}

int use_int(int *p) {
    int j = 3;
    return j + *p;
}

int main() {
    int* p = get_int();
    printf("integer = %d\n", *p);
    int k = use_int(p);
    printf("integer = %d\n", k);
    return 0;
}
```

e `good_pointer.c`:

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

int* get_int() {
    int* p = (int*)malloc(sizeof(int));
    *p = 2;
    return p;
}

int use_int(int *p) {
    int j = 3;
```

```

    return j + *p;
}

int main() {
    int* p = get_int();
    printf("integer = %d\n", *p);
    int k = use_int(p);
    printf("integer = %d\n", k);
    return 0;
}

```

Compile-os e execute-os. O que aconteceu? Consegue perceber o que se passa?

Recompile o programa `gcc -g -o bad_pointer bad_pointer.c` e corra-o no `gdb`.

```

$ gdb bad_pointer
gdb> break main
gdb> run
gdb> step
gdb> ENTER
gdb> ...

```

Onde ocorre o erro? Porquê?

**Nota.** Há vários cenários que podem dar origem a erros no acesso à memória durante a execução de um programa. Estes são normalmente sinalizados pelo sistema operativo como **segmentation fault** ou **bus error** e resultam na interrupção abrupta da execução do programa. Os cenários mais comuns resultam da desreferenciação (aplicação do operador `*`) a um apontador inválido. No seguinte bloco de código mostram-se três situações típicas:

```

/*
 * Null Pointer:
 * o endereço NULL não é válido
 */
char *p1 = NULL;
...
char c1 = *p1; /* erro em runtime */

/*
 * Wild Pointer:
 * p2 não foi inicializado, normalmente conterá
 * um endereço inválido
 */

```

```
char *p2;
...
char  c2 = *p2;  /* erro em runtime */

/*
 * Dangling Pointer:
 * um apontador que deixou de ser válido
 */
char *p3 = (char*)malloc(sizeof(char));
...
free(p3);
...
char  c3 = *p3;  /* erro em runtime */
```