Apoio à Ficha de Trabalho N.º 4

Versão 2024/24

Objectivos: Funções; ponteiros, *Arrays*; *Strings*;
Passagem de parâmetros "por valor" e "valor da referência" (normalmente denominado como "por referência").

Exercício Introdutório

Antes de relembrar o conceito de ponteiro, e como este conceito está já muito perto do hardware da máquina, será conveniente apresentar a Arquitectura de von Neumann. É certo que, para cerca de metade dos estudantes, estes conceitos já terão sido leccionados em TC, mas, muito possivelmente, a muitos deles, "passaram-lhes ao lado". Para os restantes, obviamente, poucos ou nenhuns conhecimentos terão a esse nível, pois ainda não foram leccionados em TC.
Assim, se talvez a maioria já saiba que os valores das variáveis são guardados em memória, muitos deles ainda não perceberam como são lá guardados. Ou seja, que quando se declara uma variável, o compilador se encarrega de atribuir um conjunto de bytes de memória a essa variável (sendo o seu número ditado pelo tipo da variável) e cria uma tabela onde a cada variável é associado o seu endereço inicial, para saber onde irá guardar o dado ou lê-lo, quando o programa o especificar.
Além disso e, porventura algo que mais dificulta a compreensão, é o facto de para a maioria dos estudantes, a memória é algo abstracto que guarda dados, mas não têm o conhecimento de como realmente o dado fica guardado fisicamente. Isso é ainda mais se complica quando se fala em endereços de memória, já que desconhecem o que é e para que serve, porque não têm qq. percepção da sua implementação física.
Agora, tratando-se de ponteiros, o problema ainda mais se agrava, já que, possivelmente, se muitos terão aprendido na aula Teórica, o conceito de ponteiro, ou seja, que quando se cria um ponteiro, está a criar-se uma variável, mas que já não vai guardar o valor da variável, mas sim um novo endereço de memória que pode guardar o valor associado ao ponteiro ou, ainda, um outro endereço de outro ponteiro e assim sucessivamente, mas, mais uma vez depois não conseguem aplicar o conceito, porque lhes falta a ligação dele à realidade física do funcionamento da memória e, especialmente o conceito de endereço.
Assim, talvez seja boa ideia, mostrar a arquitectura de Von Neumann e explicar o que é a memória e como se colocam ou leem lá os dados e como se especifica que byte de memória se vai ler ou escrever e o que se pode lá colocar.
Podem encontrar na pasta onde está este documento, um ficheiro "Arquitectura_VonNeumann", com alguns diapositivos de TC, que podem ajudar a dar uma breve panorâmica acerca dela: os princípios em que assenta e uma mostra diagramática da própria arquitectura, com especial ênfase na componente memória e endereços.

Assim, num browser, ir a (https://pythontutor.com/) que, apesar do seu nome, aceita também código em **Python**, **JavaScript**, **C**, **C++** e **Java**, e usando o bloco de código infra, efectuar a execução do código linha a linha e ir explicando o que vai acontecendo: na caixa de texto onde aparecem os outputs e na parte do ecrã onde vai sendo desenhado o diagrama dos dados (as variáveis) e respectivos endereços.

Será importante que alterem a visualização dos dados a serem apresentados em cada rectângulo do diagrama, para ser lá mostrado o 1.º endereço da memória onde reside.

```
void aplica1(int x)
{
    x=10;
    printf("%d\n",x);
}


void aplica2(int *x)
{
    *x=10;
}


void main()
{
     int n;
     n=5;
     aplica1(n);
     printf("%d\n",n);
     aplica2(&n);
     printf("%d\n",n);
}
```

Explicar o que significa o *x no parâmetro da função aplica2 e a diferença para o x na função aplica1. Mostrar as diferenças entre uma e outra chamada das funções no main.
Perguntar aos estudantes o que será então mostrado no ecrã.

Registar a resposta no quadro e copiar o programa para o Pyton Tutor e executar o código linha a linha e chamar a atenção para o que vai aparecer no diagrama de memória mostrado, com especial relevo para a seta que vai aparecer a ligar o ponteiro X a um novo bloco de memória.
Depois mostrar a que acontece quando se usa a varável x não função aplica1 e o ponteiro x, na função aplica2.

Como já se disse numa ficha de apoio anterior, esta aplicação pode igualmente ser usada como um *debugger* bastante interessante, ainda que sejam reconhecidas algumas limitações, como não permitir instruções de entrada de dados (tipo getchar(), scanf(), etc.), além da dimensão do programa não poder ser elevada (referem cerca de 3000 linhas), mas indicam que o objectivo da aplicação é mostrar a execução de programas que caibam num quadro (um professor está a mostrar um programa para explicar um qualquer conceito, como uma simples declaração de uma variável e depois atribuir-lhe um valor). Como na execução aparecerá um bloco que mostra a variável e depois o valor (na instrução de atribuição) a visualização permite, imediatamente, a compreensão por parte dos estudantes do significado de variável e, bem mais importante quando os conceitos a transmitir são mais complexos como blocos de dados alocados directamente (utilizando uma função *malloc* ou *calloc*) de que irão lembra-se amiúde, já daqui a algumas semanas.
Talvez um pouco antes, ouvirão falar de ponteiros ou apontadores (uma variável especial, pois não contém um dado, mas o valor do endereço de memória onde reside um variável ou bloco de dados). Neste caso, a utilização do Phyton Tutor poderá ter um valor inestimável para a sua compreensão.
Quando se tratar de um programa onde se utilize este mecanismo (ou seja, exista um ponteiro que irá "apontar" (indicar qual o endereço inicial, como se referiu acima) onde reside um dado ou bloco de

dados, a utilização desta aplicação será ainda de maior interesse: a apresentação visual mostrará uma seta a ligar o ponteiro e o endereço inicial do dado ou bloco de dados apontado. Assim, a seta e a variável de onde parte (o ponteiro) e o destino (o que é apontado), será visualizado com valores e até os endereços de memória respectivos, permitindo facilmente perceber quando existe um erro de lógica ou de falta de alguma instrução (normalmente, o esquecimento de um *malloc* ou este estar incorrecto).

1 - a) Implemente uma função que permita trocar o valor de duas variáveis. Teste a função num pequeno programa para o efeito.

Passagem de parâmetros por valor (ex. 1, solução 1 - errada)

Para o prog.:

```
void main()
{
    int a1, a2;
    a1=10
    a2=20;
}
```

Em memória, ter-se-á:

int a1, a2

1000	a2	
1003	a1	

Quando se executa o programa, a memória ficará:

int a1, a2

1000	a2	20
1003	a1	10

Quando se invocar a função trocaE, de protótipo: void trocaE(int x1, int x2),

```
void trocaE(int x1, int x2)       /* Funçao que deveria trocar o valor de
duas variáveis mas está errada */
{
  int tmp;
  tmp = x1;
  x1 = x2;
  x2 = tmp;
}
```

ao ser executada a instrução no main(), trocaE(a1, a2), o que acontece é que é criado um espaço de memória para a função e teremos, em memória:

4F0	tmp	
500	x2	
503	x1	
1000	a2	20
1003	a1	10

Como já sabemos, a passagem por valor, copia os valores de a1 e a2, para os correspondentes parâmetros x1 e x2.

4F0	tmp	
500	x2	20
503	x1	10
1000	a2	20
1003	a1	10

```
void trocaE(int x1, int x2)        /* Funçao que deveria trocar o valor de
duas variáveis mas está errada */
{
  int tmp;
  tmp = x1;
  x1 = x2;

  x2 = tmp;
}
```

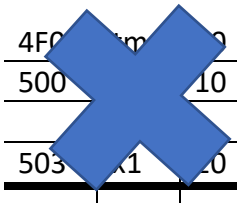
Quando a função trocaE inicia a sua execução, ficará, em memória:

4F0	tmp	10
500	x2	20
503	x1	10
1000	a2	20
1003	a1	10

Depois de executada, ter-se-á:

4F0	tmp	10
500	x2	10
503	x1	20
1000	a2	20
1003	a1	10

Realmente, é efectuada a troca, mas....
Ao terminar a execução, todo o ambiente associado à função trocaE(...) é destruído e só ficará o ambiente do main() com o valor das variáveis origem.



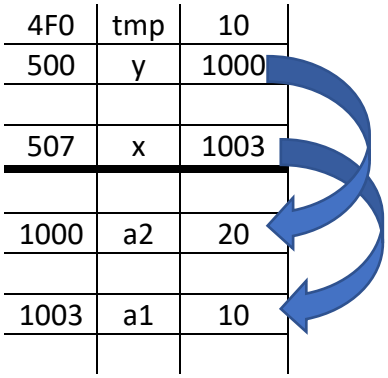
4F0	tmp	10
500	x2	10
503	x1	20
1000	a2	20
1003	a1	10

1-b) Implemente uma função que permita trocar o valor de duas variáveis. Teste a função num pequeno programa para o efeito.

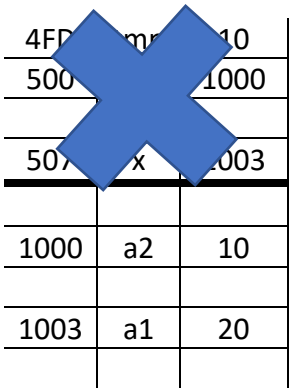
```
#include <stdio.h>

int leitura()    /* Funçao para leitura de um valor inteiro */
{
 int x;

  printf("Qual o valor? ");
  scanf("%d",&x);
  return(x);
}
```

```
void troca(int *x,int *y)   // Funçao que troca o valor de duas variáveis
{
  int tmp;
  tmp = *x;

  *x = *y;
  *y = tmp;
}
```

Quando esta função é executada, em memória fica :



Depois de executada a função, ter-se-á :




```
void main()
{
 int a1,a2;

 a1=leitura();
 a2=leitura();
 printf("\n  Antes de troca :\ta1= %d\ta2= d\n",a1,a2);

 trocaerrado(a1,a2);
 printf("\n Depois de troca :\ta1= %d\ta2= %d (não trocou)\n",a1,a2);

 troca(&a1,&a2);
 printf("\n Depois de troca :\ta1= %d\ta2= %d\n",a1,a2);
}
```

2 - Elabore um programa (não esqueça a respectiva função main()) que copie o conteúdo de uma string para outra, usando:
a) uma função para o efeito, utilizando o operador [];
b) uma função para o efeito, utilizando ponteiros.

```
// Exercício 2) Programa que copia o conteúdo de uma string para outra, usando:
// a) uma função para o efeito, utilizando o operador [];
// b) uma função para o efeito, utilizando ponteiros.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include<locale.h>
#define _CRT_SECURE_NO_WARNINGS

void strCopiaOpParentesis(char x[], char y[]){
    int i=0;
    while (x[i]!='\0')
    {
        y[i]=x[i];
        i++;
    }
    y[i]=x[i];
}

void strCopiaComPonteiros(char *x, char *y){
    while (*x!='\0')
    {
        *y=*x; x++; y++;
    }
    *y=*x;
}

int main(void)//  ::::::::::::::::::::::::::::::::
{
    setlocale(LC_ALL,"Portuguese");
    system("clear");
    char c;
    char str1[21],str2[21];
    int i;
    printf("\nEscreva uma frase (max. 20 caracteres): ");
    fgets(str1, sizeof(str1), stdin);

    printf("\n*** Parte I - Resolução, usando o operador [].....\n");
    printf("\nA string inserida foi: %s", str1);
    // solução, usando o operador []
    // mostra a string, caracter a caracter, separados por espaço
    i=0;
    printf("A string inserida (str1), mostrada caracter a caracter, separados por espaços
                                                                    foi:\n --> ");
    while(str1[i]!='\0'){
        printf("%c ", str1[i]);
        i++;
    }
      // invoca a função para efectuar a cópia
    strCopiaOpParentesis(str1, str2);
    // mostra a string str2, depois da cópia
    printf("A string str2, depois de efectuada a cópia é: %s\n", str2);

    printf("*** Parte II - Resolução, usando ponteiros.....\n\n");
    // solução usando ponteiros
    // mostra a string str1, caracter a caracter, separados por espaço
    char *x=str1;
    printf("A string inserida (str1), mostrada caracter a caracter, separados por espaços
                                                                    foi:\n --> ");
    while(*x!='\0'){
        printf("%c ", *x);
        x++;
    }
```

```
    // invoca a função para efectuar a cópia
    strCopiaComPonteiros(str1, str2);
    // mostra a string str2, depois da cópia
    printf("A string str2, depois de efectuada a cópia é: %s", str2);
    // mostra a string str2, depois da cópia, caracter a caracter, separados por espaço
    x=str2;
    printf("A string str2, depois de copiada, mostrada caracter a caracter,
                                                    separados por espaços foi:\n --> ");
    while(*x!='\0'){
        printf("%c ", *x);
        x++;
    }
    printf("\nDigite uma tecla para terminar...");
    c=getchar();
    return 0;
}
```

3 - a) Crie uma função strModifyParImpar que insira um caracter especificado nas posições par ou ímpares de um uma string.

Resolução:

```
/* Função que troca os elementos pares ou ímpares do vector str1 (string) pelo caracter c.
// Se ParOuImpar='p", troca o caracter existente nos elementos pares;
*/ se ParOuImpar='i', troca o caracter existente nos elementos ímpares.
void strModify(char *str1, char seParOuImpar, char c)
{
    int i=0;
    while(*(str1+i)!='\0')
    {
        switch (seParOuImpar)
        {
            case 'p': {if (i%2 == 0) *(str1+i)=c;}; break;
            case 'i': {if (i%2 == 1) *(str1+i)=c;}; break;
            default: printf("Indicação dos elementos a trocar inválida!\n"); return;
        }
        i++;
    }
}
```

b) Copie a função anterior e altere-a de forma a trocar os elementos múltiplos de um número a especificar (parâmetro m) pelo caracter c, criando a função strModifyM.

```
void strModifyM(char *str1, short m, char c)
{
    int i=0;
    while(*(str1+i)!='\0')
    {
        if(i%m==0)
            *(str1+i)=c;
        i++;
    }
}
```

c) A função anterior que se pretende mais genérica do que a criada na alínea a), poderá ser utilizada para substituir os elementos pares (especificando m=2), mas não os ímpares.
Copie a função anterior e altere-a de forma que já possa também substituir os números ímpares ou atendendo a um qualquer deslocamento (alterar o elemento de índice múltiplo de m + d (deslocamento)). P. ex., para substituir os números ímpares, bastaria especificar m=2 e d=1. A nova função deverá chamar-se strModifyMMaisD.

```
void strModifyMMaisD(char *str1, short m, char c, short d)
{
    int i=0;
    if (d>=m){
        printf("Deslocamento %d, inválido!\n");
```

```
        return;
    }

    while(*(str1+i)!='\0')
    {
        if(i%m==0){
            *(str1+i+d)=c;
            i+=d;
        }
        i++;
    }
}
```

Crie o main() que permita testar as três funções criadas.

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

int main()
{
    char vect[30]="Joao Francisco Silva";
    strModify(vect, 'i', '+');
    printf("Depois de executar a função strModify, a string é agora %s\n", vect);

    strcpy(vect, "Joao Francisco Silva");
    strModifyM(vect, 3, '+');
    printf("Depois de executar a função strModifyM, a string é agora %s\n", vect);

    strcpy(vect, "Joao Francisco Silva");
    strModifyMMaisD(vect, 2, '+', 1);
    printf("Depois de executar a função strModifyMMaisD, a string é agora %s\n", vect);
    return 0;
}
```