Ficha de Trabalho n.º 3

Ponteiros

Ponteiro é uma variável que guarda o endereço de memória de outra variável (para onde aponta).

- declaração de *nome_ponteiro* como um ponteiro que aponta para uma variável do tipo *tipo*:

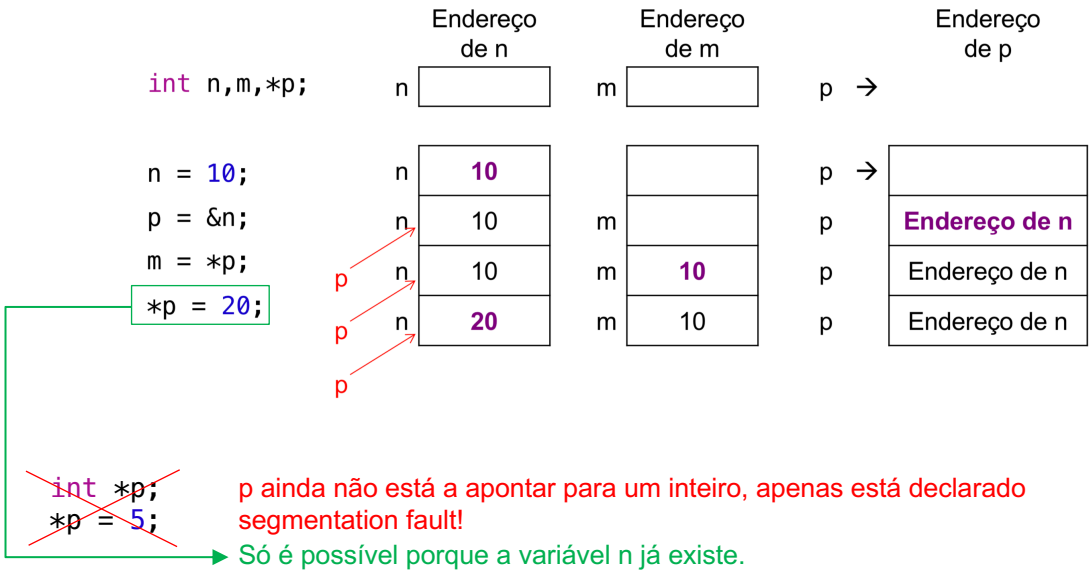
  `<tipo> * <nome_ponteiro>`

  NOTA: não reserva espaço de memória, apenas espaço para um endereço de memória.
- conteúdo da zona de memória guardada pelo ponteiro *nome_ponteiro* (valor para o qual *nome_ponteiro* aponta)

  `*<nome_ponteiro>`
- endereço de memória de *nome_ponteiro*

  `&<nome_ponteiro>`
- inicialização do ponteiro *nome_ponteiro* como NULL, ou seja, a apontar para nada

  `*<nome_ponteiro> = NULL`




```
int *p;
*p = 5;
```

p ainda não está a apontar para um inteiro, apenas está declarado
segmentation fault!
Só é possível porque a variável n já existe.

```
int n,*p;
p = &n;
*p = 5;
printf("n = %i",n);
```

ou

```
int *pont;
pont = (int *)malloc(sizeof(int));
*pont = 5;
printf("valor = %i\n",*pont);
free(pont);
```

Ver funções da biblioteca `stdlib.h`

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

1. Considere o seguinte conjunto de instruções de um programa.

```
int a = 2, b = 3;
int *p, *q;

p = &a;
q = &b;
```

a. Indique o valor das expressões apresentadas

i. `p == &a`

ii. `*p - *q`

iii. `**&p`

iv. `*p+1`

v. `*(q-2)`

b. Teste e analise o resultado da seguinte instrução

```
printf("%p %u %u %d %d %d %d\n",p,p,&p,*p+4,**&p,5**p,**&p+6);
```

2. Elabore um programa que mostre o número de bytes ocupados por cada tipo de variável (char, int, float), assim como o tamanho (em bytes) de um ponteiro para cada um desses tipos.

NOTA: Use o operador `sizeof()`.

Exemplo de apresentação:

```
  tipo |  tamanho (bytes)  |  tamanho do ponteiro (bytes)
+------+-------------------+------------------------------+
| char |                 1 |                            4 |
```

3. Considere as seguintes 3 versões da função *troca* e as 5 possibilidades de chamada de cada uma dessas funções. Dependo da forma como os dois elementos (função e chamada) são combinados, o resultado da sua execução poderá ser diferente.

```
int main()
{
      int a, b;
      unsigned long int i, j;

      i = &a;
      j = &b;
      // troca...
}
```

Funções

A

```
void troca(int *x, int *y)
{
      int aux;

      aux = *x;
      *x = *y;
      *y = aux;
}
```

B

```
void troca(int *x, int *y)
{
      int * aux;

      aux = x;
      x = y;
      y = aux;
}
```

C

```
void troca(int *x, int *y)
{
      int aux;

      aux = *x;
      x = *y;
      *y = aux;
}
```

Chamadas da função

I

```
int main()
{
      // ...
      troca(i,j);
}
```

II

```
int main()
{
      // ...
      troca(a,b);
}
```

III

```
int main()
{
      // ...
      troca(&i,&j);
}
```

IV

```
int main()
{
      // ...
      troca(*i,*j);
}
```

V

```
int main()
{
      // ...
      troca(*a,*b);
}
```

Considere as seguintes condições iniciais (antes de chamar a função *troca*) dos campos de memória do computador:

Nome de variável						*a*	*b*		*i*		*j*	
Endereço "físico"	…	5	…	12	…	1001	1002	…	2220	…	2224	…
Valor		1001		1002		5	12		1001		1002	

Indique quais os valores nos campos de memória 5, 12, 1001, 1002, 2220 e 2224 depois da execução de cada uma das 15 combinações possíveis do programa.

Por exemplo, combinando a função *A* com as instruções *I* obtém-se

Nome de variável						*a*	*b*		*i*		*j*	
Endereço "físico"	…	5	…	12	…	1001	1002	…	2220	…	2224	…
Valor		1001		1002		12	5		1001		1002	

Ponteiros, Tabelas e Strings

É possível realizar operações aritméticas com ponteiros: adicionar, subtrair, incrementar (avança no endereço de memória) e decrementar (recua no endereço de memória).
Quando se incrementa um ponteiro, ele avança o número de bytes (*sizeof(tipo de dados)*) que o seu tipo ocupa.
Estas operações podem ser úteis para manipular tabelas.

Quando se declara e atribui um ponteiro para uma tabela, ele aponta para o primeiro elemento da tabela.

```
int v[] = {10,20,30,40,50};
int * pv;

pv = &v[0]; //atribui a pv o endereco do primeiro elemento da tabela
printf("%i\n",*pv);
++pv; //ponteiro aponta para segundo elemento da tabela
printf("%i\n",*pv);
```

4. Usando ponteiros, elabore subprogramas que manipulem uma matriz de reais com dimensão 20×20:

a. Inicializar a matriz com o –1;

```
void inicializa_matrizA(float *pm)
```

b. Inicializar a matriz com valores de 1 a 400 por ordem crescente das linhas e das colunas;

```
void inicializa_matrizB(float *pm)
```

c. Colocar numa posição da matriz (*i* , *j*) o valor indicado pelo utilizador;

```
void atribui_valor_ij(float *pm, int i, int j)
```

d. Adicionar duas matrizes.

```
float * soma(float *pm1, float *pm2)
```

5. Usando ponteiros, elabore novas versões para as funções

a. *strlen* (para contar os caracteres de uma string)

existente:

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

nova versão:

```
int tamanho_str(char *ps)
```

b. *strncpy* (usada para copiar parte de uma string)

existente copia n caracteres de uma string:

```
char * stpncpy(char * dst, const char * src, size_t len)
```

nova versão deve copiar parte de uma string indicando a partir de onde pretende copiar e quantos caracteres:

```
char * copia_substr(char * src, int i, int len)
```

É ainda possível criar tabelas de ponteiros:

```
int * ptab[10];
ptab[0]=pv; //do exemplo anterior
printf("%i\n",*ptab[0]);
```