Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 4

Arrays em C e apontadores

- O nome do array vale por um apontador constante:

```
int v[3];     // v tem tipo int*

v[0] = 1   ⇔   *v = 1

v[2] = 5;  ⇔   *(v+2) = 5;
```


Criar variáveis dinamicamente

- Criando dinamicamente variáveis:

```
void * malloc( int size )
```

- Criar um vetor de 3 inteiros:

```
int *v = (int*)malloc(3*sizeof(int));
```

- Usando como um vetor:

```
v[2] = 5;
```

- São libertadas (destruídas) com:

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


Exemplo

```
int v1[MAX] = { ..........};
```

- Criar uma copia só com os elementos >=0:

```
int npos = 0;
for (int i=0; i<MAX; i++ )
   if ( v1[i]>=0 ) npos++;

int *vpos = (int*)malloc(npos*sizeof(int));
for (int i=0, j=0; i<MAX; i++ )
   if ( v1[i]>=0 ) vpos[j++] = v1[i];
```


Arrays em C e funções

- Um parâmetro que é um array de dimensão pré-definida:

```
    void fun( int vet[3] ){ ...
```

NOTA:

```
int myvet[3];
fun( myvet );  ← O que é passado é o endereço!
```

- Mas podemos ter parâmetros arrays sem dimensão pré-definida:

```
void fun( int vet[] );
ou  void fun( int *vet );
```


Arrays em C e funções

- O nome do array representa o seu inicio
  - Se um argumento é um array, o valor que é passado é o do endereço!

```
int myvet[3];
fun( myvet );
```

  independentemente da declaração de fun:

```
void fun(int v[3]) vs. void fun(int *v)
```

- Porquê?
  - Consequências se passasse todo o array? Memória ocupada? Tempo de execução?

Parâmetros "ambíguos"

- Mas um array não é um objeto → não tem campo *length*!

```
void fun( int *vet );
```

  - Vet será um array de int, ou aponta só um int?
  - Fun deve ser chamada com um array? De que dimensão?

- Solução habitual: acompanhar o array com a sua dimensão:

```
void fun( int vet[], int vsize );
ou  void fun( int *vet, int vsize );
```

- E é vetor, matriz, etc...?
  - lêr documentação/comentários

Exemplo

- Somar todos os elementos de um vetor de inteiros

```
#include <stdio.h>

int sum( int v[], int vsize ) {
   int s = 0;
   for (int i=0; i<vsize; i++ )
      s = s+v[i];
   return s;
}
int main( ) {
   int myvect[] = { 1, 2, 5, 4, 2};
   printf("soma=%d\n", sum(myvect, 5) );
}
```


Strings em C

- Não existe tipo específico para strings.
- Convenção: uma sequência de chars terminada com '\0' (código ASCII 0)

```
char msg[]={'H','e','l','l','o','\0'};
char msg[]= "Hello";
```

- Mas há constantes! Exemplo: `"Hello"`

```
char *msg2 = "Hello";
```

  - msg2 é um apontador para a string **constante** "Hello"

Strings em C – string.h

- A biblioteca do C dispõe de várias funções para manipular *strings*, declaradas em string.h
- Exemplos:

```
int strnlen(char *str, int max);
char *strncpy(char *dst, char *src, int len);
char *strncat(char *dst, char *src, int len);
int strncmp(char *s1, char *s2, int len);
```

- E versões sem dimensão (sem max ou len):

```
strlen, strcpy, strcat, strcmp
```


Exemplos de utilização

```
char *str1;
char str2[] = "Hello";
char str3[100];

str1=str2;
str1[strlen(str1)-1] = 'u';

strcat( strcpy(str3,str1),str2 );

str1 == str2    vs.  strcmp(str1, str2)
```


Java ... Arrays

- Recorde:

```
int[] a = {1, 2, 3};
int[] b = a;
int[] c = {1, 2, 6};
b[2]=6;
```

Valores em a, b e c ?

```
a == b  vs.  b == c
        vs.  Arrays.equals(b, c)
```

Java ... Referencias de objetos

- Recorde:

```
class A { public int i; ......}
A a = new A();
A b, c;
a.i = 1;  b = a;  b.i++;
c = new A(); c.i = 2;
```

Valores em a, b e c?

a == b *vs.* **b == c**

Java ... String

- Recorde:

```
String s1 = "Hello";
String s2;

s2 = s1;    vs.  s2 = new String(s1);

s2 == s1    vs.  s2.equals( s1 )
```

Nota: as String são imutáveis (não podem ser alteradas)

Erros comuns

```
char *str1;
char str2[10];
```

- Usar apontadores não iniciados:

```
strcpy( str1, "Hello" );
```

- Esquecer a terminação '\0':

```
char str[5]="Hello"; ← sizeof("Hello") é 6!
        {'H','e','l','l','o','\0'}
```

- Aceder fora do array:

```
strcat( strcpy(str2, "Hello"), "World");
```


Mais erros comuns

```
char *str1; char str2[10];
```

- Copiar apontadores em vez de arrays (usar = em vez de strcpy ou do memcpy):

```
str2 = "hello";   ou   str1 = str2;
```

- Comparar apontadores em vez dos caracteres:

```
str1 == str2
```

em vez de `strcmp( str1, str2 )`

- Julgar que o tamanho do array é o da string e vice versa:

```
strcpy(str2, "Hello");
```

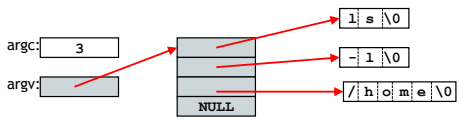
`strlen(str2)` não é 10. É 5!

Exemplo: main

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

- Argv contém uma cópia da linha de comando.
- Argv é um vetor de strings. Argc o número de strings. Exemplo:

```
$ ls  -l  /home
```



Escrever todos os seus argumentos:

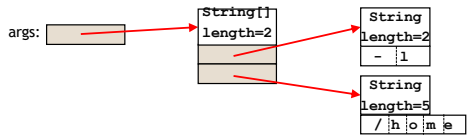
```
#include <stdio.h>

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

Exemplo: main em Java

`public static void main(String[] args)`

- Args contém uma cópia dos argumentos na linha de comando.
- Args representa um vetor de `String`. Exemplo:

```
$ java Ls  -l  /home
```



Escrever todos os seus argumentos:

```
class Prog {

 public static
 void main( String[] args ) {
   System.out.print("num args: ");
   System.out.println( args.length );
   for ( int i=0; i<args.length; i++ ) {
     System.out.printf("%d: %s\n", i, args[i] );
   }
 }
}
```