Ponteiros

Baseado nos slides do Prof. Mauro.

Ponteiros

A memória do computador é uma sequência de bytes, na qual endereçamos cada um sequencialmente.

Como vimos, uma variável é uma região da memória que reservamos para armazenar um dado, por exemplo:

int a;
Reservamos 4 bytes (em geral) para armazenar um número inteiro.

char c;
Reservamos 1 byte para armazenar um caractere.

Ponteiros

Ponteiros são variáveis que armazenam um endereço de memória, por exemplo:

int *a;
Reservamos 4 bytes (máquina 32 bits) para armazenar um endereço de memória. Neste endereço de memória estará armazenado um número inteiro (em geral, 4 bytes).

char *c;
Reservamos 4 bytes (máquina 32 bits) para armazenar um endereço de memória. Neste endereço de memória estará armazenado um caractere (1 byte).

Ponteiros

Todas as variáveis tem um endereço, que podemos obter pelo operador & (não confundam com o &&), por exemplo:

```
int a;
int *b;

a = 10;
b = &a;
```

A variável **b** contém o endereço da variável **a**, que, por sua vez, contém o número 10.

Ponteiros

```
char c;
char *d;

c = 'a';
d = &c;
```

A variável **d** contém o endereço da variável **c**, que por sua vez contém o caractere a.

Observe que, quando declaramos um ponteiro, não estamos reservando espaço para armazenar um dado, estamos apenas reservando espaço da memória para armazenar um endereço.

Ponteiros

O valor especial NULL (precisamos da stdlib.h) significa que não apontamos para nenhum endereço. Por exemplo:

```
int *a;
a = NULL;
```

O ponteiro **a** está sendo declarado, mas **a** não aponta (neste momento) para nenhum endereço de memória.

Ponteiros

Importante: em C, o operador * tem diversas funções, dependendo do seu contexto:

Caso 1:

```
int a, b, c;

a = 10;
b = 20;
c = a * b;
```

O operador *, neste caso, significa multiplicação.

Ponteiros

Caso 2:

```
int *a;

a = NULL;
```

O operador *, neste caso, significa que queremos declarar um ponteiro.

Caso 3:

```
int a, *b;

b = &a;
*b = 10;
```

Neste caso, o “conteúdo” de **b** recebe 10.

Ponteiros

Exemplo:

```
int main() {
   int a, b, *c, d;

   a = 10;
   b = 20;
   c = &a;
   d = a + b;
   printf(“%d\n”, d);

   *c = 30;

   d = *c + b;
   printf(“%d”, d);

   d = a + b;
   printf(“%d”, d);
   return(0);
}
```

Ponteiros

Exemplo:

```
void troca(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
   return;
}

int main() {
   int a, b;

   a = 10;
   b = 20;
   printf(“%d e %d\n”, a, b);

   troca(&a, &b);
   printf(“%d e %d\n”, a, b);
   return(0);
}
```

Ponteiros

Assim como não faz sentido deixarmos de inicializar uma variável comum:

```
int a;
printf(“%d\n”, a);
```

Também não faz sentido utilizar um ponteiro sem inicializá-lo:

```
int *a;
printf(“%d\n”, *a);
```

Ponteiros

Temos que tomar um cuidado adicional com ponteiros: ele pode ter sido inicializado, mas inicializado com o valor NULL, logo não podemos fazer:

```
int *a;

a = NULL;
printf("%d", *a);
```

É muito comum o uso de *"if"* para verificar se um ponteiro possui o valor NULL ou não, para só então utilizá-lo.

Alocação dinâmica

Podemos fazer um ponteiro “apontar” para qualquer variável comum:

```
int a, *p;

a = 10;
p = &a;
```

Alocação dinâmica

Também podemos “apontar” para um bloco de memória que reservamos, por exemplo:

```
int main() {
   int *a;

   a = (int*) malloc( sizeof(int) );
   ...
   free(a);
   a = NULL;
   return(0);
}
```

Alocação dinâmica

Cuidados:

- Quando usamos memória alocada dinamicamente, não podemos esquecer de desalocar estas memórias!
  - Dica: cada malloc tem que ter um free correspondente!

- Depois que desalocamos um bloco de memória, não podemos mais utilizá-lo, embora o ponteiro ainda contenha o endereço da memória utilizada!
  - Dica: depois que chamar free(ponteiro), faça: ponteiro=NULL;

Vetores

Podemos alocar blocos de qualquer tamanho:

Exemplo 1:

```
int main() {
   int a[100];

   a[50] = 1;
   return(0);
}
```

Vetores

Exemplo 2:

```
int main() {
{
   int *a;

   a = (int*) malloc( sizeof(int)*100 );

   a[50] = 1;
   free(a);
   a = NULL;
   return(0);
}
```

Vetores

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

void ler(int n, int *vetor) {
   int i;
   for(i=0;i<n;i++)
      scanf(“%d “, &vetor[i]);
   return;
}
void imprimir(int n, int *vetor) {
   int i;
   for(i=0;i<n;i++)
      printf(“%d “, vetor[i]);
   printf(“\n”);
   return;
}
int main() {
   int *a;

   a = (int*) malloc( sizeof(int)*100 );
   if (a == NULL) {
       return(-1);
   }
   ler(100, a);
   imprimir(100, a);

   free(a);
   a = NULL;
   return(0);
}
```

Matrizes

Exemplo 1:

```
int main() {
   int a[100][30];

   a[50] [5] = 1;
   return(0);
}
```

Matrizes

Exemplo 2:

```
#include <stdlib.h>
int main() {
  int **a, i;

  a = (int**) malloc( sizeof(int*)*100 );
  if ( a == NULL )
    return(-1);

  for(i=0; i<100; i++) {
    a[i] = (int*) malloc( sizeof(int)*30 );
    if ( a[i] == NULL )
       return(-1);
  }

  a[50][5] = 1;
  for(i=0; i<100; i++) {
     free(a[i]);
     a[i] = NULL;
  }
  free(a);
  a = NULL;
  return(0);
}
```