Universidade Federal de Santa Maria
Curso: Ciência da Computação
Disciplina: Laboratório de Programação II
Revisão: C
Professora: Juliana Kaizer Vizzotto

0.1 Ponteiros

Os *ints* guardam inteiros. Os *floats* guardam números de ponto flutuante. Os *chars* guardam caracteres. **Ponteiros** guardam endereços de memória. Anotamos o *endereço* de algo em uma variável ponteiros, para depois usar.

Um ponteiro também tem tipo. No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro **int** aponta para um inteiro, isto é, guarda o endereço de um inteiro.

0.1.1 Sintaxe

```
tipo_do_ponteiro *nome_da_variavel;
```

É o asterisco que o compilador saber que aquela variável não vai guardar um valor mais sim um endereço para aquele tipo especificado.

```
int *pt;
char *temp,*pt2;
```

O primeiro exemplo declaro um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda a variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. *O ponteiro deve ser inicializado antes de ser usado*. Para atribuir um valor ao ponteiro podemos utilizar o operador &, que retorna o endereço de uma variável:

```
int count=10;
int *pt;
pt=&count;
```

Podemos agora alterar o valor de `count` utilizando `pt`. Para tanto usamos o operador "inverso" do operador &, o operador `*`. A expressão `*pt` é equivalente ao próprio `count`. Isto significa que se quisermos mudar o valor de count para `12`, basta fazer `*pt=12`.

Operações que são usadas com ponteiros são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro. Está é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro `char*` ele anda um 1 byte na memória e se você incrementa um ponteiro `double` ele anda 8 bytes na memória. O decremento funciona de maneira semelhante.

```
p++;
p--;
```

Agora, para incrementar o *conteúdo* da variável apontada pelo ponteiro:

```
(*p)++;
```

0.1.2 Ponteiros e Vetores

Quando você declara uma matriz da seguinte forma:

```
tipo_da_variavel nome_da_variavel [tam1][tam2]...[tamN}
```

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz ($tam1 * tam2 * tamN * tamanho_do_tipo$). O compilador então aloca esse número de bytes em um espaço livre de memória. O `nome_da_variavel` é um *ponteiro para o tipo da variável matriz*. Tendo alocado na memória espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o primeiro elemento da matriz. Assim, a notação:

```
nome_da_variavel[indice]
```

é equivalente a:

```
*(nome_da_variavel+indice)
```

Assim como `*nome-da-variavel` é equivalente a `nome_da_variavel[0]`.

Mas note que um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa que não se consegue alterar o endereço que é apontado pelo "nome do vetor". Assim, o nome de um vetor é um ponteiro constante. Da mesma maneira que podemos indexar o nome de um vetor, também podemos indexar um ponteiro qualquer.

O principal cuidado ao se usar um ponteiro é: saiba sempre para onde o ponteiro está apontando!

0.1.3 Ponteiros para Funções

O C permite que acessemos variáveis e funções através de ponteiros. Podemos, por exemplo, passar uma função como argumento para outra função.

```
void PrintString(char *str, int (*func)());

main(void){
  char String [20] = "Alow";
  int (*p) ();
  p =puts;
  PrintString(String,p);
  return 0;
}

void PrintString(char *str,int (*func)()){
  (*func) (str);
}
```

0.2 Alocação Dinâmica

A alocação dinâmica permite ao programador criar variáveis em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado.

0.2.1 Malloc

A função `malloc` serve para alocar memória e tem o seguinte protótipo:

```
void *malloc(unsigned int num);
```

A função toma o número de bytes que queremos alocar (`num`), aloca na memória e retorna um ponteiro `void*` para o primeiro byte alocado. O ponteiro `void*` pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função `malloc` retorna um ponteiro nulo. Por exemplo:

```
main (void){
 int *p;
 int a=12;
 p=(int *) malloc(a*sizeof(int));
 if (!p){
     printf("** Erro: Memoria Insuficiente **");
     exit;
    }
 ...
 return 0;
}
```

Nesse exemplo, é alocada memória suficiente para se colocar `a` números inteiros. O operador `sizeof()` retorna o número d bytes de um inteiro. O ponteiro `void*` que `malloc()` retorna é convertido para um `*int` pelo cast e é atribuído a `p`.

0.2.2 Calloc

A função `calloc` também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc (unsigned int num,unsigned int size);
```

Essa função aloca uma quantidade de memória igual a `num *size`, isto é, aloca memória suficiente para uma matriz de `num` objetos de tamanho `size`. Retorna um ponteiro `*void` para o primeiro byte alocado. O ponteiro `*void` pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função `calloc` retorna um ponteiro nulo.

0.2.3 Realloc

A função `realloc` serve para realocar memória e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

A função modifica o tamanho da memória previamente alocada apontada por `*ptr` para aquele especificado por `num`. O valor de `num` pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque o `realloc` pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida.

0.2.4 Free

Quando alocamos memória dinamicamente é necessário que a liberemos quando ela não for mais necessária. Para isto existe a função `free`:

```
void free (void *p);
```

Basta então passar para `free()` o ponteiro que aponta para o início da memória alocada.

```
main (void){
 int *p;
 int a=12;
 p=(int *) malloc(a*sizeof(int));
 if (!p){
     printf("** Erro: Memoria Insuficiente **");
     exit;
    }
 ...
 free(p);
 ...
 return 0;
}
```