

## **Trabalho Prático I**

**Alunos:** Felipe Melo & Raul Camizão

**Disciplinas:** Algoritmo e Estruturas de Dados & Introdução a Ciência da Computação

**Curso:** Ciência da Computação

### **Introdução**

O homem contemporâneo está familiarizado com a base 10 (decimal), no dia-a-dia, já os computadores atuais trabalham exclusivamente com a base 2 (binário), assim é preciso fazer conversões entre estas bases quando se pretende inserir algum valor para ser processado pelo computador.

Em calculadoras e computadores, a conversão de números para o binário para a operação interna, e depois, para o decimal na visualização é um processo interno pré-programado para ser feito por ela. O ponto a ser entendido aqui é que internamente ela faz tudo em binário, em outras palavras: ela converte o que foi digitado para binário, faz o cálculo, converte o resultado para decimal e apresenta o resultado.

No entanto quando se está escrevendo um programa é normal a introdução de valores no meio do código, e em muitas situações a digitação de códigos binários é muito complicada/longa para o programador, então existem outros códigos que facilitam a digitação, na prática é muito utilizada a base 8 (octal), e a base 16 (hexadecimal), ambas derivadas da base 2 (note que estas bases facilitam a digitação somente, de qualquer forma ao ser compilado toda e qualquer base usada para escrever o programa é convertida para base 2 para que o valor seja usado pelo processador).

Desta forma este Trabalho Prático teve como objetivo, implementar um Conversor de Bases em Linguagem C, que executa conversões de números reais entre as bases **decimal**, **binária**, **octal** e **hexadecimal**.

### **Solução do Problema**

Para a resolução do problema proposto pelo trabalho prático, foi implementado um algoritmo que primeiramente recebe uma cadeia de caractere (sempre contendo o sinal e o valor), o número correspondente à base de origem e o número correspondente à base de destino.

Feitas todas as verificações do conteúdo da *stdin* (existência do sinal, existência do ponto, verificação de inclusão do valor digitado ao conjunto de caracteres

pertencentes à base de origem , condição de parada do programa e etc.) e retornando pela *stderr* a mensagem de erro, o programa chama a função que transforma os caracteres no seu valor referente a base em que se está trabalhando. Isto é necessário pois cada valor do tipo char, possui um valor para representá-lo como um número inteiro .

Como pode ser visto neste fragmento da tabela ASCII:

## ASCII Table

Dec	Char
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9

Para um valor octal, como por exemplo: 246.0, se fossemos jogar algarismo por algarismo no somatório de conversão para base decimal que veremos a frente, teríamos  $50 \cdot 10^2 + 52 \cdot 10^1 + 54 \cdot 10^0$ . O que resultaria em um valor incoerente com o objetivo do trabalho. Desta forma é necessário sempre subtrair 48 do valor inteiro do char, para obtermos o valor correto do algarismo. Por outro lado, para o caso particular da base hexadecimal que utiliza os caracteres de 'A' a 'F', foi atribuído o valor específico correspondente ao mesmo.

Por meio de uma função que retorna ao programa o índice em que se encontra o '.' (ponto) que separa a parte inteira da fracionária, é possível separar a parte inteira da fracionária, atribuindo as ao tipo *int* e *float* respectivamente.

Valor:	+	1	0	<div style="border: 1px solid black; padding: 2px 5px;">.</div>	1	
Índice:		0	1	2	<div style="border: 1px solid black; padding: 2px 5px;">3</div>	4

Aplicando então o algoritmo de conversão para a base decimal, primeiramente para a parte inteira:

```

1  InicioAlgoritmo
2
3  Variaveis
4  inteiro: i, exp, soma, parte_inteira;
5  exp <- posição_do_ponto -2; //desconsidera o ponto e o sinal
6  soma <- 0;
7
8  Inicio
9  para i, de 1 a posição_do_ponto, faça:
10     soma <- soma + n[i] x 10^exp ;
11     exp = exp - 1;
12 fimpara
13
14 parte_inteira <- soma;
15
16 FimAlgoritmo

```

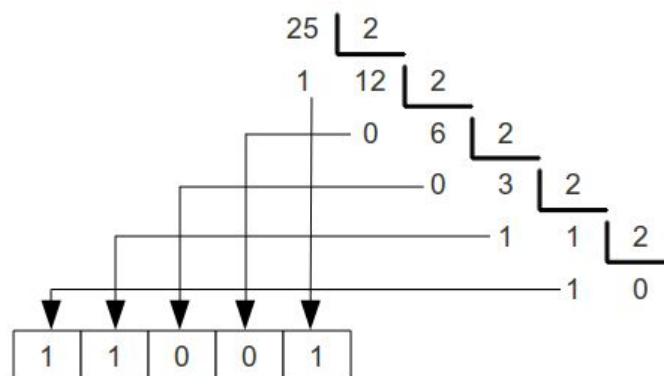
Depois a parte fracionária:

```
1  InicioAlgoritmo
2
3  Variaveis
4  inteiro: i, exp;
5  real: soma, parte_fracionaria;
6  exp <- -1;
7  soma <- 0;
8
9  Inicio
10 para i, de posição_do_ponto +1 a num_digitos, faça:
11     soma <- soma + n[i] x 10^exp ;
12     exp = exp - 1;
13 fimpara
14
15 parte_fracionaria <- soma;
16
17 FimAlgoritmo
```

Obtido a conversão do conteúdo numérico da *string* para a base decimal, se a base de destino era a decimal, somamos a parte inteira com a parte fracionária e *loop* concluído.

Se não, devemos realizar mais uma conversão, primeiramente da parte inteira. Este método se baseia em **sucessivas divisões** do inteiro pela base de destino, guardando a cada divisão, o resto em um vetor, até que se obtenha zero da divisão do número inteiro pela base.

Como mostrado no exemplo prático a seguir:



Já para parte fracionária utilizamos o oposto da parte inteira, o método das **multiplicações sucessivas**. Esse método consiste em multiplicar a parte

fracionária do número desejado pela base para a qual se deseja converter, até que a mesma seja ZERO.

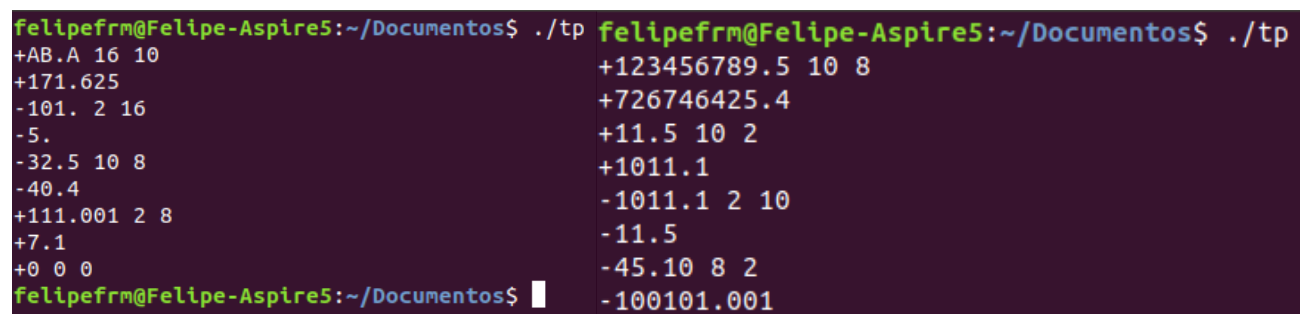
O número convertido para a base desejada é igual a concatenação de todas as partes inteiras obtidas nos resultados das multiplicações, que também são guardadas no mesmo vetor, porém com o ponto separando-os da parte inteira obtida pelo método das divisões.

Por fim, para podermos imprimir o resultado da conversão na tela, precisamos transformar o vetor em um vetor de caracteres (*string*), para isso deve-se realizar o processo inverso do que foi feito no início do programa, somar 48 ao valor do *char*. Nos casos particulares da hexadecimal, para os *char* de 10 a 15 devemos atribuir de 'A' a 'F', isto é necessário para obtermos uma saída satisfatória.

Se não fosse implementado este trecho de código ao programa, teríamos hipoteticamente o seguinte exemplo: se tivermos como entrada: +171.625 10 16, obteríamos na saída +1011.10, o que obviamente não queremos.

## Testes realizados

Os testes foram satisfatórios para os exemplos de entradas presentes na documentação de orientação do TP, e para várias outras entradas analisadas. segue a imagem da execução do ./tp:



```
felipefrm@Felipe-Aspire5:~/Documentos$ ./tp felipefrm@Felipe-Aspire5:~/Documentos$ ./tp
+AB.A 16 10 +123456789.5 10 8
+171.625 +726746425.4
-101. 2 16 +11.5 10 2
-5. +1011.1
-32.5 10 8 -1011.1 2 10
-40.4 -11.5
+111.001 2 8 -45.10 8 2
+7.1 -100101.001
+0 0 0
felipefrm@Felipe-Aspire5:~/Documentos$
```

Os testes foram feitos entre todas as bases (bin-dec, bin-oct, bin-hex, oct-bin ...), alternado entre números inteiros ou reais, sinal positivo ou negativo. Sempre conferindo o resultado da conversão com a “Calculadora” *BC* do terminal linux. Foi testado também, casos em que o número (incluindo sinal, e ponto) é maior que 50 dígitos, o programa nesta situação não realiza a conversão e acusa uma mensagem de erro, implementada justamente para estes casos.

### Tratamento de erros:

[illegible]

Lembrando que a mensagem de erro, sai pela *stderr*, a saída de erro padrão, que independente à *stdout*, a saída padrão. Logo, como pedido na documentação de orientação do trabalho prático, na *stdout* sai apenas o resultado da conversão e nada mais.

### Análise Valgrind:

```
felipefrm@Felipe-Aspire5: ~/Documentos/tp1
Arquivo Editar Ver Pesquisar Terminal Ajuda
felipefrm@Felipe-Aspire5:~/Documentos/tp1$ valgrind ./tp
==7156== Memcheck, a memory error detector
==7156== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7156== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7156== Command: ./tp
==7156==
+AB.A 16 10
+171.625
-101. 2 16
-5.
-32.5 10 8
-40.4
+111.001 2 8
+7.1
+0 0 0
==7156==
==7156== HEAP SUMMARY:
==7156==    in use at exit: 0 bytes in 0 blocks
==7156==    total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==7156==
==7156== All heap blocks were freed -- no leaks are possible
==7156==
==7156== For counts of detected and suppressed errors, rerun with: -v
==7156== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

O relatório do *Valgrind* informa que 1 bloco de memória foi alocado e devidamente liberado. Sendo assim, não há vazamento de memória durante a execução do programa.

## Referência Bibliográfica

As 10 conversões numéricas mais utilizadas na computação. Disponível em<[https://dicasdeprogramacao.com.br/as-10-conversoes-numericas-mais-utilizadas-na-computacao](https://dicasdeprogramacao.com.br/as-10-conversoes-numericas-mais-utilizadas-na-computacao/) />. Acesso em: 25 maio 2018.