Instituto Superior de Engenharia de Lisboa

LEETC

Programação II

2024/25 – 1.º semestre letivo

Primeira Série de Exercícios

1. Exploração de operações *bitwise* e deslocamentos

Na utilização de inteiros com sinal a 32 bits são frequentes os valores que não necessitam de ocupar os 32 bits por terem uma sequência de bits a zero na parte baixa ou uma sequência de bits com o valor do sinal na parte alta. Tirando partido desta observação, propõe-se uma representação compacta, a 16 bits, com o seguinte formato:

- bits 0 a 10: codificam a parte significativa do valor, a 11 bits, incluindo o sinal;
- bits 11 a 15: codificam o deslocamento, indicando o número de bits a deslocar para a esquerda, de modo a posicionar a parte significativa na palavra a 32 bits.

A tabela seguinte contém exemplos de alguns valores acompanhados de uma codificação possível respetiva.

Valor a 32 bits	Deslocamento (5 bit)	Parte significativa (11 bit)	Codificação
0	0	000 0000 0000	0x0000
1	0	000 0000 0001	0x0001
2	0	000 0000 0010	0x0002
-1	0	111 1111 1111	0x07ff
-2	0	111 1111 1110	0x07fe
2048	2	010 0000 0000	0x1200
-2048	1	100 0000 0000	0x0c00

Há valores que podem ter várias representações equivalentes, por terem menos do que 11 bits significativos. Também há valores não codificáveis neste formato, por terem mais do que 11 bits significativos.

A função `encode`, abaixo, realiza a codificação do parâmetro `data` e devolve o respetivo código a 16 bits.

```
#define VAL_BITS 11

unsigned short encode( int data ){
      int shift = 0;
      int highBits;
      while( ( highBits = data >> VAL_BITS - 1 ) != 0 && highBits != ~0 ){
            data >>= 1;
            ++shift;
      }
      return shift << VAL_BITS | data & ~( ~0 << VAL_BITS ) ;
}
```

1.1. Escreva a função

```
int decode( unsigned short code );
```

que devolve o valor a 32 bits com sinal, reproduzido a partir do código passado no parâmetro `code`.

1.2. Escreva um programa de teste para as funções anteriores, que receba valores inteiros do *standard input* e, para cada valor, apresente no *standard output* a representação a 16 bits codificada com o formato especificado, exibida em hexadecimal, e o valor a 32 bits reproduzido a partir dela.

Ensaie com diversos valores, incluindo codificáveis e não codificáveis; verifique os resultados da descodificação.

1.3. Modifique a função `encode`, adicionando o parâmetro `valid` para a produzir uma indicação de validade do código gerado.

```
unsigned short encode( int data, int *valid );
```

A função deve afetar a variável apontada por `valid` com: 1, em caso de sucesso; 0, no caso oposto.

1.4. Modifique o programa de teste, adicionando a apresentação de uma mensagem de aviso quando a função `encode` indica resultado inválido.

2. Manipulação de *strings*

Pretende-se o processamento de *strings* contendo linhas de texto provenientes de um ficheiro. Cada linha é formada por uma sequência de campos, separados por ponto-e-vírgula ';'. Cada campo pode conter qualquer sequência de caracteres, incluindo alfabéticos, numéricos, espaços e *tab*, como no exemplo seguinte.

```
primeiro campo;; terceiro campo \t;  ; palavras   do quinto campo 1234\n
```

O processamento a realizar envolve a separação dos campos, uniformização dos seus espaçamentos e a comparação de palavras de forma insensível a maiúsculas ou minúsculas.

2.1. Escreva a função

```
char *splitField(char *str);
```

que separa um campo localizado no início da *string* `str`, substituindo o respetivo separador (';') pelo terminador de *string*. Retorna um ponteiro para o campo seguinte, indicando o endereço do caráter após o separador. Quando o campo processado é o último, não existindo mais nenhum separador até ao terminador de *string*, a função retorna NULL para indicar que não há mais campos.

Considere, por exemplo, a utilização desta função no troço de programa seguinte:

```
char s[] = "primeiro;; terceiro \t;  ; palavras   do quinto \n";
char *p1 = s, *p2;
do{
    p2 = splitField( p1 );
    printf( "{%s}", p1 );
}while( (p1 = p2) != NULL );
```

Na execução deste exemplo, o resultado em *standard output* seria:

```
{primeiro}{}{ terceiro      }{  }{ palavras   do quinto
}
```

Note que a última chaveta aparece na linha de baixo, devido à existência do caráter '`\n`'.

2.2. Escreva um programa de teste que, por cada linha recebida de *standard input*, aplique a função `splitField` e apresente em *standard output* os campos identificados, usando o formato descrito no exemplo anterior.

Propõe-se que leia de *standard input* com a função `fgets` da biblioteca normalizada. Admita que cada linha tem a dimensão máxima de 255 carateres.

Para efeitos de demonstração, deve criar ficheiros, contendo linhas com campos, e utilizá-los com redireccionamento de *input*.

2.3. Pretende-se uniformizar a separação entre palavras numa *string*, colocando um espaço apenas entre palavras, em substituição de qualquer sequência de separadores, espaço (' '), tabulação ('\t') ou mudança de linha ('\n'). Se houver separadores no início ou no final das *string*, devem ser eliminados.

Escreva a função

```
void separatorUnify( char str[] );
```

que, recebendo uma *string* no parâmetro `str`, aplica a formatação especificada e deixando a *string* processada no início da memória anteriormente ocupada pela *string* original (note que a nova forma tem sempre um número de carateres igual ou inferior à original.) São valorizadas as soluções que não

necessitem de memória temporária para o processamento. Pode realizar a modificação da *string* por manipulação direta ou usar os mecanismos declarados nos *header files* normalizados, nomeadamente `ctype.h` e `string.h`.

2.4. Adaptando o código realizado para a alínea 2.2, escreva uma segunda versão do programa de teste que, dispondo do texto separado pela função `splitField`, aplique a cada um dos campos a função `separatorUnify`, antes de o apresentar em *standard output*.
Se receber o texto do exemplo em 2.1, deve apresentar:
```
{primeiro}{}{terceiro}{}{palavras do quinto}
```

2.5. Escreva a função "*string compare ignoring case*"
```
int strcmp_ic( const char *str1, cont char *str1 );
```
que compara alfabeticamente, insensível a maiúsculas ou minúsculas, as *strings* indicadas pelos parâmetros. Retorna um valor negativo, zero ou positivo se, respetivamente, `str1` é alfabeticamente inferior, idêntica ou superior a `str2`. Deve utilizar as funcionalidades relacionadas com o controlo de maiúsculas e minúsculas, declaradas no *header file* `ctype.h`.

2.6. Escreva um programa de teste para a função `strcmp_ic`. O programa tem um argumento de linha de comando para indicar uma palavra a comparar; recebe sucessivas palavras de *standard input*, compara cada uma delas com a do argumento e indica o resultado da comparação em *standard output*.
Propõe-se que leia as palavras de *standard input* com a função `scanf` da biblioteca normalizada. Para efeitos de demonstração, deve criar ficheiros, contendo palavras com diversidade de maiúsculas e minúsculas, e utilizá-los através do redireccionamento de *input*. Por simplificação, assume-se o texto codificado em ASCII básico, pelo que não se pretende o processamento específico de letras acentuadas.

2.7. Escreva um programa de aplicação "`filter.c`" para filtrar linhas de texto, formadas por campos, com base no conteúdo de um dos campos, comparado com uma *string* através da função `strcmp_ic`.
O programa tem dois argumentos de linha de comando:
- O primeiro é um número, de valor 1 ou superior, para selecionar a posição do campo a comparar;
- O segundo é a *string* para ser comparada, em cada linha, com o campo selecionado.

A filtragem consiste em apresentar, em *standard output*, apenas o conteúdo das linhas em que do campo selecionado corresponde à *string* indicada no argumento de linha de comando; as restantes linhas são descartadas. Antes de realizar a comparação, a string de argumento e o campo indicado, de cada linha, devem ser uniformizados pela função `separatorUnify`.
As linhas selecionadas são apresentadas com o seu conteúdo original, sem modificações causadas pelo processamento dos campos.
O valor numérico do primeiro argumento pode ser obtido, da respetiva *string*, usando uma função de biblioteca, por exemplo `strtoul`. Quando ao conteúdo do segundo argumento de linha de comando, normalmente é apenas uma palavra; no entanto, pode ser inserida uma *string* com espaços se for delimitada por aspas.
Propõe-se que o programa leia o texto de *standard input* com a função `fgets` da biblioteca normalizada, assumindo a dimensão máxima de linha com 255 carateres.
Para efeitos de demonstração, deve criar ficheiros, contendo linhas com campos, e utilizá-los através do redireccionamento de *input*. Os campos devem permitir diversos cenários de teste, contendo:
- Texto diferente da palavra pretendida, para demonstrar que a comparação o distingue;
- A *string* pretendida com maiúsculas ou minúsculas, para demonstrar que a comparação aceita ambas;
- Carateres de espaçamento em diversos arranjos, para confirmar o efeito da uniformização.

ISEL, 2024.09.30