Strings multibyte

Video desta aula

Localização

Em computação, o termo locale designa um conjunto de parâmetros que definem a "localização", ou seja as preferências de linguagem de um sistema, como a língua usada nas mensagens do sistema, a codificação de caracteres e formatos de informações usuais (números, data/hora, moeda, etc).

No sistema Linux, por exemplo, o comando `locale` informa os parâmetros locais em uso. No exemplo abaixo o sistema usa como língua o Português brasileiro e como codificação padrão o UTF-8:

```
$ locale
LANG=pt_BR.utf8
LANGUAGE=pt_BR.utf8
LC_CTYPE="pt_BR.utf8"
LC_NUMERIC="pt_BR.utf8"
LC_TIME="pt_BR.utf8"
...
```

O parâmetro de localização mais importante para um programa em C é LC_CTYPE (*character type*), pois ele define o conjunto de caracteres e afeta o comportamento de funções como `printf` e `scanf`.

Um programa em C pode consultar ou modificar os parâmetros de *locale* do SO através da função `setlocale()`:

locale.c

```c
#include <stdio.h>
#include <locale.h>

int main()
{
  char *locale ;

  // obtém o LC_CTYPE atual do programa
  locale = setlocale (LC_CTYPE, NULL) ;
  printf ("Current locale is %s\n", locale) ;

  // ajusta o LC_TYPE do programa para o default do SO
  locale = setlocale (LC_CTYPE, "") ;
  if (locale)
    printf ("Current locale is %s\n", locale) ;
  else
    fprintf (stderr, "Can't set the specified locale\n") ;

  // ajusta o LC_TYPE do programa para "pt_BR.iso88591"
  locale = setlocale (LC_CTYPE, "pt_BR.iso88591") ;
  if (locale)
    printf ("Current locale is %s\n", locale) ;
  else
```

```
    fprintf (stderr, "Can't set the specified locale\n") ;
}
```

Se a função `setlocale()` for chamada com uma string vazia (“”), a configuração de localização do programa é feita com base nas variáveis de ambiente providas pelo sistema operacional. Então é recomendável **sempre chamar essa função** no início de programas que manipulam caracteres não-ASCII.

Caracteres e strings em C

Caracteres ASCII

A linguagem C manipula caracteres codificados em ASCII sem dificuldade, usando variáveis do tipo `char`. Dessa forma, strings ASCII são meros vetores de caracteres terminados com um caractere nulo (`\0`).

Caracteres ISO-8859

Como as codificações ISO-8859-* usam apenas um byte por caractere, programas em C podem manipular caracteres em ISO sem dificuldade, usando variáveis do tipo `unsigned char` (para representar caracteres com código numérico entre 0 e 255).

Além disso, deve-se definir o *locale* do programa para garantir o funcionamento correto de funções como `toupper()`, `isalpha()`, etc. com os caracteres estendidos:

```
char *locale ;

locale = setlocale (LC_CTYPE, "pt_BR.ISO-8859-1") ;
```

Obviamente, o locale ISO-8859-1 deve estar disponível no sistema operacional (essa informação pode ser consultada com o comando `locale -a`). Além disso, se houver escrita na tela, o **terminal deve estar configurado** para usar caracteres ISO.

Caracteres UTF-8

As coisas mudam para as codificações multibyte, pois tipo `char` é insuficiente para armazenar caracteres em codificações multibyte. Por isso, alguns cuidados devem ser tomados ao definir e usar strings em UTF-8, por exemplo:

- Ao alocar memória para as strings, lembre-se que alguns caracteres podem ocupar mais de um byte.
- As funções de entrada/saída formatadas, como `printf`, `scanf` suas variantes, suportam UTF-8 sem modificações, basta chamar `setlocale()` no início do programa.
- O índice não corresponde mais necessariamente à posição de cada caractere na string. Por exemplo, `nome[3]` não corresponde necessariamente ao quarto caractere da string `nome`, caso ela esteja codificada em UTF-8.
- A função `strlen` sempre informa o **número de bytes** da string; para obter o número de caracteres, deve-se usar a função `mbstowcs` (*multi-byte-string-to-wide-character-string*), que retorna o número de caracteres da string.

Como regra geral, deve-se sempre consultar o manual para verificar se a função desejada funciona com strings multibyte.

O código abaixo apresenta um exemplo de programa que manipula strings em UTF-8:

char-utf8.c

```c
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <stdlib.h>

int main()
{
  char *frase = "Olá ɣ 诶  ˝ ;

  // ajusta a localização de acordo com o SO
  setlocale (LC_ALL, "") ;

  // conteúdos da string
  printf ("Frase           : %s\n",  frase) ;

  // número de caracteres usando strlen()
  printf ("strlen (frase)  : %ld\n", strlen(frase)) ;

  // número de caracteres usando mbstowcs()
  printf ("mbstowcs (frase): %ld\n", mbstowcs(NULL, frase, 0)) ;
}
```

Saída:

```
Frase           : Olá ɣ 诶
strlen (frase)  : 16
mbstowcs (frase): 9
```

Caracteres largos

O padrão C 90 introduziu o conceito de caracteres “largos”, ou seja, com mais de um byte. Ao contrário dos caracteres *multibyte*, os caracteres largos têm **sempre o mesmo tamanho**, geralmente 2 ou 4 bytes (depende da plataforma). Em Linux, um caractere largo ocupa 4 bytes e pode representar qualquer *code point* do padrão Unicode.

Caracteres largos e strings largas são definidos pelo tipo `wchar_t`:

char-wide.c

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>

int main ()
{
  wchar_t c ;         // um caractere largo
  wchar_t *s ;        // ponteiro para uma string larga
```

```
  c = L'a' ;          // caractere constante largo
  s = L"equação" ;    // string constante larga

  // ajusta a localização de acordo com o SO
  setlocale (LC_ALL,"");

  // escrita de caracteres largos
  printf ("O caractere [%lc] tem %ld bytes\n", c, sizeof (c)) ;

  // escrita de strings largas
  printf ("A string [%ls] tem %ld caracteres\n", s, wcslen (s)) ;
}
```

Saída:

```
O caractere [a] tem 4 bytes
A string [equação] tem 7 caracteres
```

Várias funções são definidas pelo padrão POSIX para manipular caracteres e strings largas. Elas geralmente estão presentes na LibC.

Algumas diferenças entre strings largas e strings multibyte UTF-8:

- Uma string larga é terminada pelo caractere largo nulo `L'\0'`, enquanto string comuns e UTF-8 são terminadas por um caractere nulo com um byte `'\0'`.
- Em uma string larga, o número de campos equivale ao número de caracteres, por isso `s[10]` sempre é o 11º caractere da string, independente do conteúdo, o que não ocorre em UTF-8.
- Uma string larga ocupa mais memória que uma string multibyte, pois todos os seus caracteres ocupam o mesmo número de bytes, independente de seu *code point*.

Caracteres largos são empregados na implementação de aplicações que manipulam muitas strings, como editores de texto. Por exemplo, o ambiente de execução da linguagem Python (que é implementado em C) usa caracteres largos para armazenar strings.

O código abaixo exemplifica e compara algumas operações usando strings largas e strings UTF-8. Ele gera diversos avisos (*warnings*) ao compilar, devidos às chamadas de funções inadequadas:

wide-utf8.c

```
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <stdlib.h>
#include <wchar.h>

int main()
{
  int i ;
  char    *frase1 =  "Olá ɣ 诶   ″ ;
  wchar_t *frase2 = L"Olá ɣ 诶   ″ ;

  // ajusta a localização de acordo com o SO
  setlocale (LC_ALL,"");
```

```
  // conteúdos das strings
  printf ("Frase 1          : %s\n",  frase1) ;
  printf ("Frase 2          : %ls\n", frase2) ;

  // tamanho em bytes
  printf ("sizeof (char)    : %ld\n", sizeof(char)) ;
  printf ("sizeof (wchar_t) : %ld\n", sizeof(wchar_t)) ;

  // número de caracteres usando strlen()
  printf ("strlen (frase1)  : %ld\n", strlen(frase1)) ;
  printf ("strlen (frase2)  : %ld\n", strlen(frase2)) ;  // incorreto

  // número de caracteres usando wcslen()
  printf ("wcslen (frase1)  : %ld\n", wcslen(frase1)) ;  // incorreto
  printf ("wcslen (frase2)  : %ld\n", wcslen(frase2)) ;

  // número de caracteres usando mbstowcs()
  printf ("mbstowcs (frase1): %ld\n", mbstowcs(NULL, frase1, 0)) ;
  printf ("mbstowcs (frase2): %ld\n", mbstowcs(NULL, frase2, 0)) ; //
incorreto

  // percurso por índice, string estreita (narrow)
  printf ("Frase1: ") ;
  for (i = 0; i < strlen(frase1); i++)
    printf ("[%c] ", frase1[i]) ;
  printf ("\n") ;

  // percurso por índice, string larga (wide)
  printf ("Frase2: ") ;
  for (i = 0; i < wcslen(frase2); i++)
    printf ("[%lc] ", frase2[i]) ;
  printf ("\n") ;
}
```

Ao executar, este programa gera:

```
Current locale is pt_BR.UTF-8

Frase 1          : Olá ɣ 诶
Frase 2          : Olá ɣ 诶

sizeof (char)    : 1
sizeof (wchar_t) : 4

strlen (frase1)  : 16
strlen (frase2)  : 1     // incorreto

wcslen (frase1)  : 4     // incorreto
wcslen (frase2)  : 9

mbstowcs (frase1): 9
mbstowcs (frase2): 1     // incorreto

Frase1: [O] [l] [] [] [ ] [] [] [ ] [] [] [] [ ] [] [] [] []
Frase2: [O] [l] [á] [ ] [ɣ] [ ] [诶] [ ] [ ] [ ]
```

Mais informações

Bibliotecas para UTF-8:

- https://en.wikipedia.org/wiki/International_Components_for_Unicode
- https://developer.gnome.org/glib/2.62/glib-Unicode-Manipulation.html
- https://juliastrings.github.io/utf8proc/

Exercícios

1. escreva um programa em C para converter um texto em ISO-8859-1 para ASCII, substituindo as letras acentuadas e cedilha por seus equivalentes sem acento.
2. escreva um programa em C para converter um texto em ISO-8859-1 para UTF-8.
3. escreva uma função `char* utf8strn(char* s, int n)` que devolva um ponteiro para a posição do n-ésimo **caractere** da string `s`, que está codificada em UTF-8.
4. escreva um programa C que imprima as tabelas ASCII e ISO-8859-1.