Sistemas Operativos 2

2021/22

***Encoding* de caracteres e suporte para unicode**

em Windows e Visual Studio

1

Tópicos

Programação portável com caracteres multi-byte

Bibliografia específica para este capítulo:

- http://www.codeproject.com/Articles/76252/What-are-TCHAR-WCHAR-LPSTR-LPWSTR-LPCTSTR-etc
- *Windows System Programming*, cap. 2

2

Caracteres e unicode em Windows com Visual Studio

Acerca de caracteres, *encoding* , Windows e Visual Studio

Caracteres de **8 bits (1 byte)**
- ANSI (bom para Inglês) → **`char`**

Multi-byte encoding (MCBS) → vários bytes por caracter
Duas variantes

- ***Fixo*** – Todos os caracteres têm o mesmo número de caracteres
  - Normalmente o cenário actual

- ***Variable byte*** / ***variable length*** – caracteres diferentes podem ocupar quantidades de bytes diferentes
  - Este cenário tem desvantagens
    - Exemplo: navegar para trás numa *string* torna-se bastante complexo

Caracteres e unicode em Windows com Visual Studio

Acerca de Caracteres, *encoding* , Windows e Visual Studio

Unicode
- Formato de caracteres *multi-byte* → **`wchar_t`**
- Adequado para todas as línguas
- Existem vários *encodings* possíveis
  - Windows: UTF-16 -> Caracteres de 16 bits
    - UTF-16 é, em teoria, um esquema *variable byte* que poderia usar até 4 bytes. Na implementação do Windows são 2 bytes

    Outros standards existentes (não necessariamente no Windows)
    - UTF-32 (4 bytes)
    - UTF-8 (também possível em Windows), comum na *Web*
      - » *1 byte para os primeiros 128 caracteres (os mais comuns) e 4 bytes para os restantes. Os primeiros 128 são iguais aos ASCII*
    - etc

Caracteres e unicode em Windows com Visual Studio

Acerca de Caracteres, *encoding* , Windows e Visual Studio

Considere-se uma variável *letra* que é um "caracter"

- Quanto é que ***sizeof(letra)*** será?
- A resposta é bastante mais complexa do que parece
  - Depende do *encoding* que está a ser usado
  - **Isto afecta a portabilidade do código fonte**
    - **Mesmo fixando o alvo como um determinado sistema operativo**

Objectivo: mantendo o domínio como sendo o Windows

Como ter código fonte compatível com situações 8 bits/16 bits de forma menos trabalhosa possível e que possa ser facilmente ser convertido entre 8 bits e 16 bits minimizando incoerências e bugs?

Caracteres e unicode em Windows com Visual Studio

Uso de 8 bits ou 16 bits nos programas

Primeira abordagem: usa-se ou *char* e *strETC* ou *wchar_t* e *wcsETC()*

8 bits	16 bits
`char letra;`	→ `wchar_t letra;`
`char nome[TAM];`	→ `wchar_t nome[TAM];`
`strETC(nome);`	→ `wcsETC(nome);`

Este código não é genérico e tem problemas de manutenção:

Existe a possibilidade da plataforma alvo poder nem estar a usar unicode de todo e pode ser necessário mudar entre 8 e 16 bits (recompilando a aplicação), mas essa tarefa implica uma adaptação manual do código, operação extensiva e que pode levar a introdução de bugs

→ **Pretende-se poder mudar o tipo de caracteres sem estar a mudar o código**

Caracteres e unicode em Windows com Visual Studio

Uso de 8 bits ou 16 bits nos programas

Abordagem melhor: nem *char* / *strETC* nem *wchar_t* e *wcsETC()* -> usar *TCHAR*

8 bits	16 bits
	`#include<TCHAR.h>`
`char letra;`	`--> TCHAR letra;`
`char nome[TAM];`	`--> TCHAR nome[TAM];`
`strETC(nome);`	`--> _tcsETC(nome);`

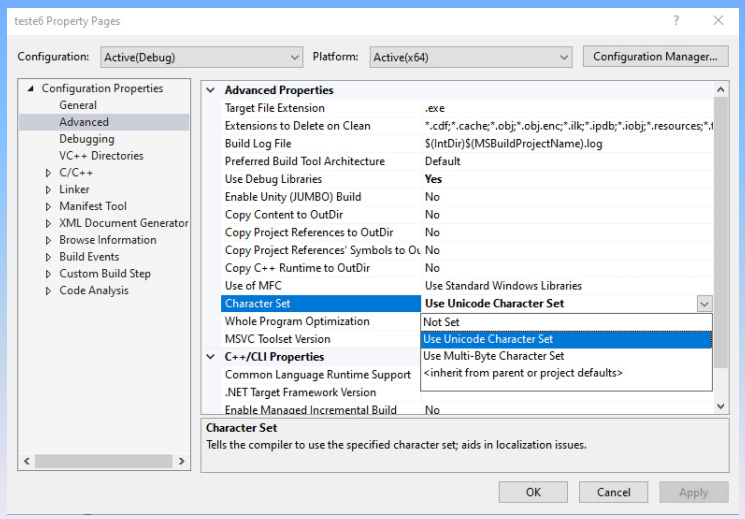
TCHAR é uma macro que se traduz em

- **char** ← se o projecto estiver configurado como ANSI/*Multi-byte charset* (MCBS)
- **wchar_t** ← se o projecto estiver configurado como Unicode

7

Caracteres e unicode em Windows com Visual Studio

Visual Studio → Propriedades do projeto → *Advanced*



8

Caracteres e unicode em Windows com Visual Studio

TCHAR → É uma macro (atenção: já existe)

```
#ifdef _UNICODE
    typedef wchar_t TCHAR;
#else
    typedef char TCHAR;
#endif
```

Atenção

- **TCHAR** transforma-se em **char** ou **wchar_t** (conforme "o define" em vigor)
- **char** e **wchar_t** são "definitivos" e já não se transformam em nada:
  - **char** é sempre **char**
  - **wchar_t** é sempre **wchar_t**

#define UNICODE e **#define _UNICODE** antes de **#include <windows.h>**
(O IDE normalmente trata da definição deste símbolo e não é preciso defini-lo)

Caracteres e unicode em Windows com Visual Studio

Funções a usar quando se usam **TCHAR**

Em vez de

```
strlen(...) / wcslen(...)
strcpy(...) / wcscpy(...)
strETC(...) / wcsETC(...)
```

Usar

```
_tcslen(...)
_tcscpy(...)
_tcsETC(...)
```

Importante

A conversão de nomes nem sempre é óbvia. Nem sempre é **str***XXX* → **_tcs***XXX*
→ **Consultar sempre o manual / referência**

Nas aplicações consola habituais

Usar	`_tmain(int argc, TCHAR * argv[])`
Em vez de	`main(int argc, char * argv[])`

Caracteres e unicode em Windows com Visual Studio

Funções **_tcsETC** → São macros

Exemplo para **_tcslen**

```
#ifdef _UNICODE
    #define _tcslen wcslen
#else
    #define _tcslen strlen
#endif
```

Atenção

- As funções **str***XXX* e **wcs***XXX* não são intermutáveis:
  - **str***XXX* recebem e manipulam caracteres de 8 bits (sempre)
  - **wcs***XXX* recebem e manipulam caracteres de 16 bits (sempre)

Caracteres e unicode em Windows com Visual Studio

O API Win32 contém a *maioria* das funções de mais alto nível em duplicado

- Uma versão para caracteres de 8 bits e outra para caracteres de 16 bits (tipicamente terminadas com ***A*** e ***W*** respectivamente)
- Para a maioria dos casos existe uma macro que esconde os pormenores de codificação de caracteres (nem ***A*** nem ***W***) e se traduz para a versão correcta automaticamente (tal como TCHAR)

Exemplo:

Função **CreateFile** - abre/cria ficheiros

O que se encontra nos .h que já existem:

```
#ifdef UNICODE
    #define CreateFile  CreateFileW
#else
    #define CreateFile  CreateFileA
#endif
```

Idem para muitas outras funções

Caracteres e unicode em Windows com Visual Studio

Para manter o máximo de portabilidade do código fonte:

- Evitar menções e uso explícito de tamanho de caracteres
- Usar as macro que são convertidas para as funções adequadas consoante o tamanho de caracter em uso

No caso do API Win32

- Usar as funções sem especificar **A** (8 bits) nem **W** (16 bits)
- Exemplo: **`CreateFile`** em vez de **`CreateFileA`** ou **`CreateFileW`**

No caso das funções da biblioteca C

- Usar as versões **`_tcs`** ou equivalente
- Exemplo: **`_tcslen`** em vez de **`strlen`** (8 bits) ou **`_wcslen`** (16 bits)

Atenção

- Nem todas as funções do API tem um equivalente em A ou W
- Nem todos os pares de funções com A/W têm uma versão genérica (sem A/W)

13

Caracteres e unicode em Windows com Visual Studio

E no caso de programas em C++?
Aplica-se a mesma lógica

Existem ambos **`cout`** (8 bits) e **`wcout`** (16 bits)
Mas o melhor é usar **`_tcout`**

```
#include <iostream>
#include <tchar.h>

#if defined(UNICODE)
    #define _tcout std::wcout
#else
    #define _tcout std::cout
#endif
```


14

Caracteres e unicode em Windows com Visual Studio

Como lidar com constantes de caracteres (strings explicitas no código)?

- `"etcetc"` é sempre uma sequência de caracteres de 8 bits

Para fazer o compilador entender (≠ "*converter*") a *string* anterior como sequência de caracteres de 16 bits?

- **`L"etcetc"`**

No entanto, o uso ou omissão do prefixo L torna o código **não-genérico**.

Para recuperar a compatibilidade e portabilidade:

-> Usar sempre **_T** (ou **TEXT**) em vez de **L** ou omissão de **L**
(**_T** converte-se para **L** ou para nada, consoante UNICODE esteja ou não definido)

Caracteres e unicode em Windows com Visual Studio

_T → É uma macro

```
#ifdef _UNICODE
  #define _T(c) L##c
  #define TEXT(c) L##c
#else
  #define _T(c) c
  #define TEXT(c) c
#endif
```

Atenção

- _T e TEXT **não são** operadores de conversão – **não convertem** dados de um tipo para outro. Apenas **interpretam** dados no código e actuam apenas **no momento de compilação**.

Funções secure e não-secure

Funções *não secure*

- Muitas das funções para tratamento de texto são apontadas como não secure
- Dependendo das configurações do compilador, o uso destas funções poderá impedir a obtenção do executável
- Mensagem de erro ou aviso possível (para a função *scanf*)

  error C4996: 'scanf': This function or variable may be unsafe
- Esta questão **não está relacionada** com caracteres *multi-byte*/Unicode, nas aproveita-se o assunto para abordar a questão

Funções secure e não-secure

Funções não seguras

Qual é a origem do problema: por que não são seguras?

- Qualquer função que trabalhe com buffer, dispondo-se a lá colocar ou obter informação sem controlo de quantos bytes vai ler ***pode ultrapassar a dimensão do buffer***
  - Na leitura de informação (ler do buffer): poderá ler informação de outras variáveis (que estão a seguir ao buffer)
    - Pode ser usado por um *hacker* para obter dados internos ao processo
  - Na escrita de informação (escrever no buffer): poderá escrever por cima de outras variáveis
    - Pode ser usado por hackers para corromper dados de outras variáveis, desviar a execução do programa para para código injectado (reescrevendo o registo na pilha que tem o endereço de retorno da função – "*stack smashing*")

Funções secure e não-secure

Funções não seguras

Exemplo

```
char buffer[30];
scanf("%s",buffer;
```

Problema: O que impede a função de ultrapassar os 30 caracteres do buffer? **-> Nada**

1ª solução (fraca): usar a formatação adequada

```
scanf("%29s", buffer);
```

-> A solução é fraca porque o programador não é obrigado a usar a formatação e na maioria das vezes não o faz.

Funções secure e não-secure

Funções seguras

Solução adequada: usar a ***versão segura*** da função pretendida

Exemplo do *scanf*

scanf -> scanf_s

```
char buffer[30];
scanf_s("%s", buffer, 30);
```

-> Aqui, o programador identifica de uma forma mais explicita e menos sujeita a erros no parâmetro adicional imediatamente seguinte qual o máximo de caracteres a ler

Pode continuar a usar o formato %*nn*s

Funções secure e não-secure

Funções seguras – caracteres *multi-byte*/Unicode

(Continuando o exemplo do *scanf)*

scanf -> scanf_s

-> Esta questão não está relacionada com caracteres *multibyte*/Unicode e existem versões para wchar_t e agnósticas

Versão wchar_t

```
wchar_t buffer[30];
wscanf_s(L"%29s", buffer, 30);
```

-> A vermelho as diferenças: *wide char*

Versão agnóstica (char/wchar_t conforme configuração do projeto

```
TCHAR buffer[30];
_tscanf_s(_T("%29"), buffer, 30);
```


Funções secure e não-secure

Recordar:

- A questão não secure / secure não está especificamente associada a tratamento de caracteres, mas este cenário é um onde mais se nota este problema
  - Existem versões secure para as funções habituais
- No caso específico de tratamento de texto, a existência de alternativas secure tanto se aplica ao caso de *char* como *wchar_t* e existem funções para ambos os casos
  - Neste caso, é melhor usar as versões **agnósticas (genéricas)** pelas razões já expostas nos slides anteriores
- Em algumas fontes bibliográficas, é usado o termo "safe" em vez de "secure", sendo a questão e assunto os mesmos.