

Dynamic Link Library (DLL) *e* *Thread Local Storage (TLS)*

- *Jeffrey Richter, Christophe Nasarre, **Windows via C/C++**, Fifth Edition, Microsoft Press, 2008 [cap. 19, 20 e 21]*
- *Microsoft, **Microsoft Developer's Network (MSDN)***

Dynamic Link Library
DLL

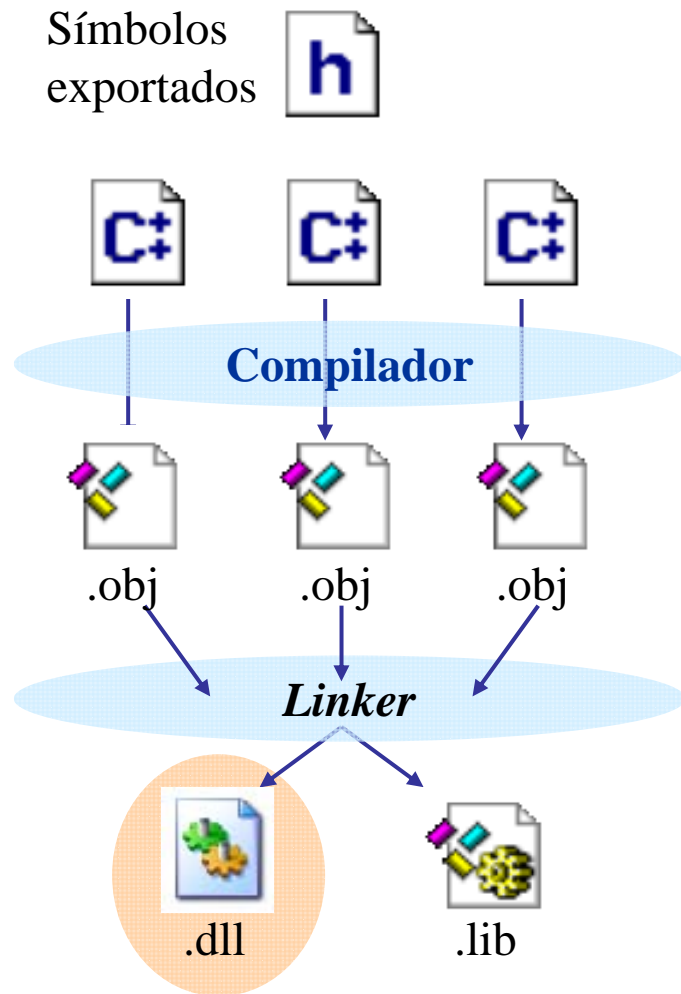
Dynamic Link Library (DLL)

- Parte Fundamental do Windows desde a primeira versão
- Biblioteca de Funcionalidades
- “Static Linking”
 - Ocorre na geração do Programa
 - Ex.:
 - Object Library: LIBC.LIB
 - Import Library: KERNEL32.LIB, USER32.LIB, GDI32.LIB
- “Dinamic Linking”
 - Ocorre em Run Time
 - Ex.:
 - Sistema:
 - KERNEL32.DLL – Gestão de memória, processos e tarefas;
 - USER32.DLL – interface com o utilizador;
 - GDI32.DLL – componente gráfica;
 - Drivers: MOUSE.DRV, Printer, Video
 - Resource: Font Files (.FON), Clipart

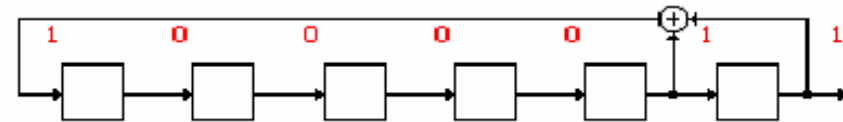
Dynamic Link Library (DLL)

- **Carregada apenas quando é necessária por uma aplicação**
- **Se tiver extensão .DLL o Windows pode carregá-la automaticamente, senão o carregamento tem de ser explícito (usando LoadLibrary)**
- **O ficheiro da biblioteca DLL deve estar situado (ordem de pesquisa):**
 - Na directoria onde reside o ficheiro executável;
 - Na directoria do Windows devolvida por GetSystemDirectory (e.g. c:\windows\system32)
 - Na directoria system 16-bits (e.g. c:\windows\system)
 - Na directoria do Windows devolvida por the GetWindowsDirectory (e.g. c:\windows);
 - Na directoria corrente;
 - Nas das directorias da variável de ambiente PATH.
- **Vantagens das DLLs**
 - Uma vez carregada por um processo passa a fazer parte deste;
 - Uma única cópia em memória do código e dados da DLL, partilhada entre os vários processos que a utilizam (poupa memória e acelera o carregamento);
 - É possível actualizar o programa de forma modular, sem ter de fazer o recompilação de toda a aplicação;
 - A aplicação pode ser configurada em tempo de execução, i.e. apenas entregar os módulos necessários (e.g. Drivers, Conversores, actualizações) ou escolher em tempo de execução qual a melhor DLL a usar;
 - Podem ser escritas / usadas em várias linguagens de programação.
 - ...

Construção de uma DLL



Exemplo: Pretende-se implementar a função `Rand62()` que gera uma sequência periódica de números Inteiros pseudo aleatórios entre 1 e 63 usando o algoritmo *Linear Recursive Sequence Generator* (LRSG) usando o polinómio gerador $X^6 + X + 1$.

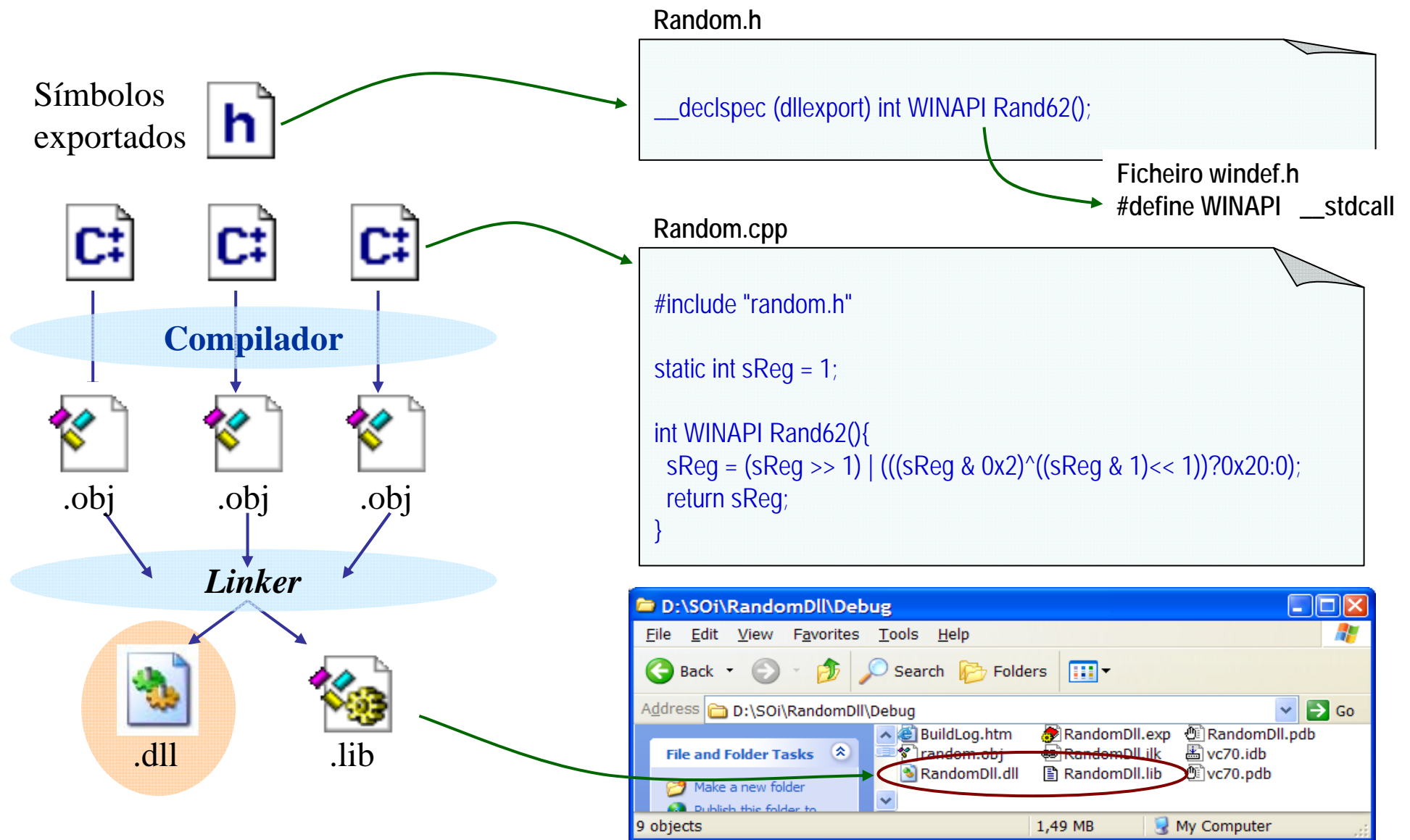


Para a realização do algoritmo em ambiente mono-thread tem-se:

```
...
static int sReg = 1;
```

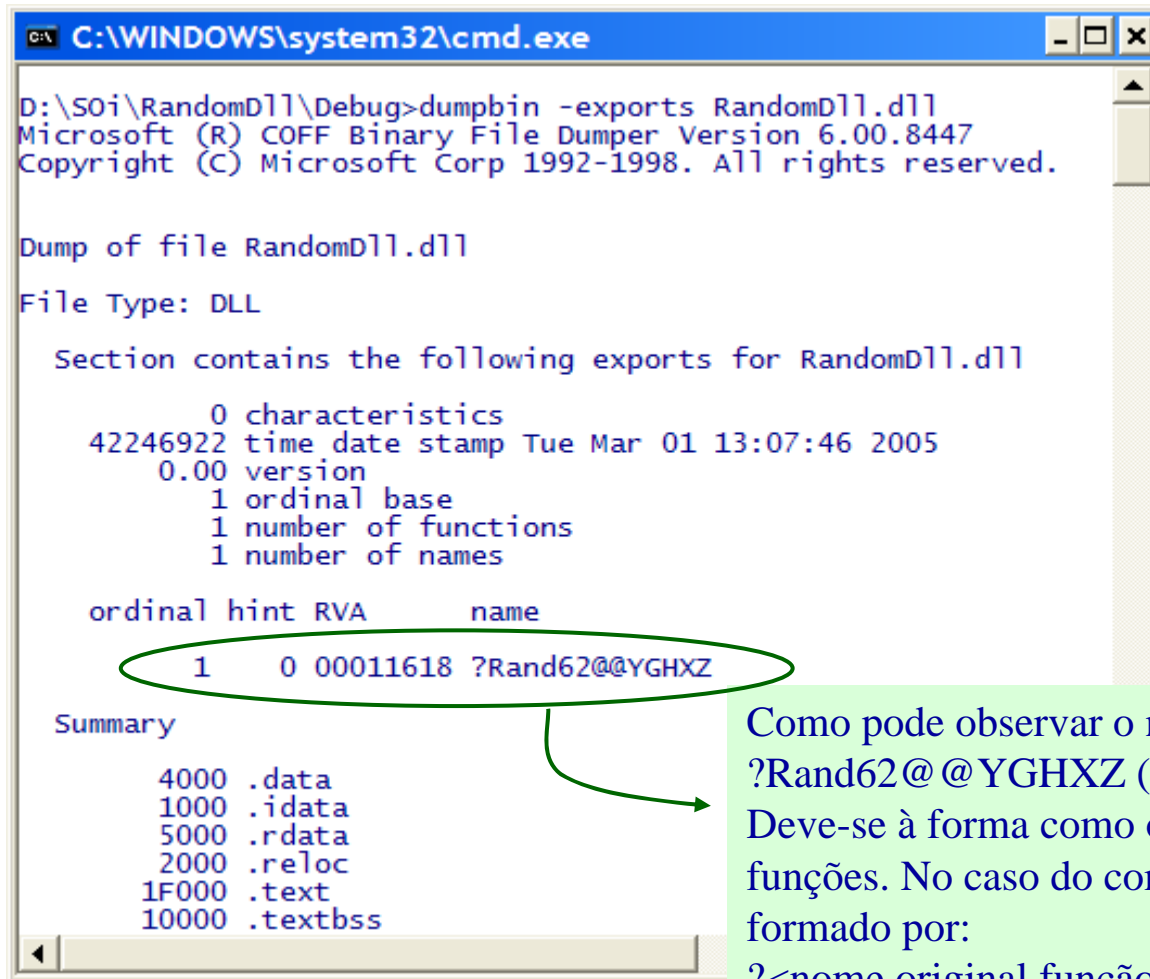
```
int Rand62(){
    sReg = (sReg >> 1) | (((sReg & 0x2)^((sReg & 1)<< 1))>0x20:0);
    return sReg;
}
...
```

Construção de uma DLL



Construção de uma DLL

Utilizando o utilitário dumpbin com a opção `-exports` podemos observar a tabela de símbolos exportados: `dumpbin -exports RandomDll.dll`



```
C:\WINDOWS\system32\cmd.exe
D:\SOI\RandomDll\Debug>dumpbin -exports RandomDll.dll
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file RandomDll.dll
File Type: DLL

Section contains the following exports for RandomDll.dll

  0 characteristics
42246922 time date stamp Tue Mar 01 13:07:46 2005
  0.00 version
  1 ordinal base
  1 number of functions
  1 number of names

ordinal hint RVA      name
1       0 00011618 ?Rand62@@YGHXZ

Summary
 4000 .data
 1000 .idata
 5000 .rdata
 2000 .reloc
1F000 .text
10000 .textbss
```

Se desenvolvermos a DLL e as aplicações que a utilizam no Visual Studio não existe nenhum problema. No entanto se a DLL for utilizada em aplicações desenvolvidas em ferramentas de desenvolvimento de outros fabricantes existem mais algumas definições a ter em conta.

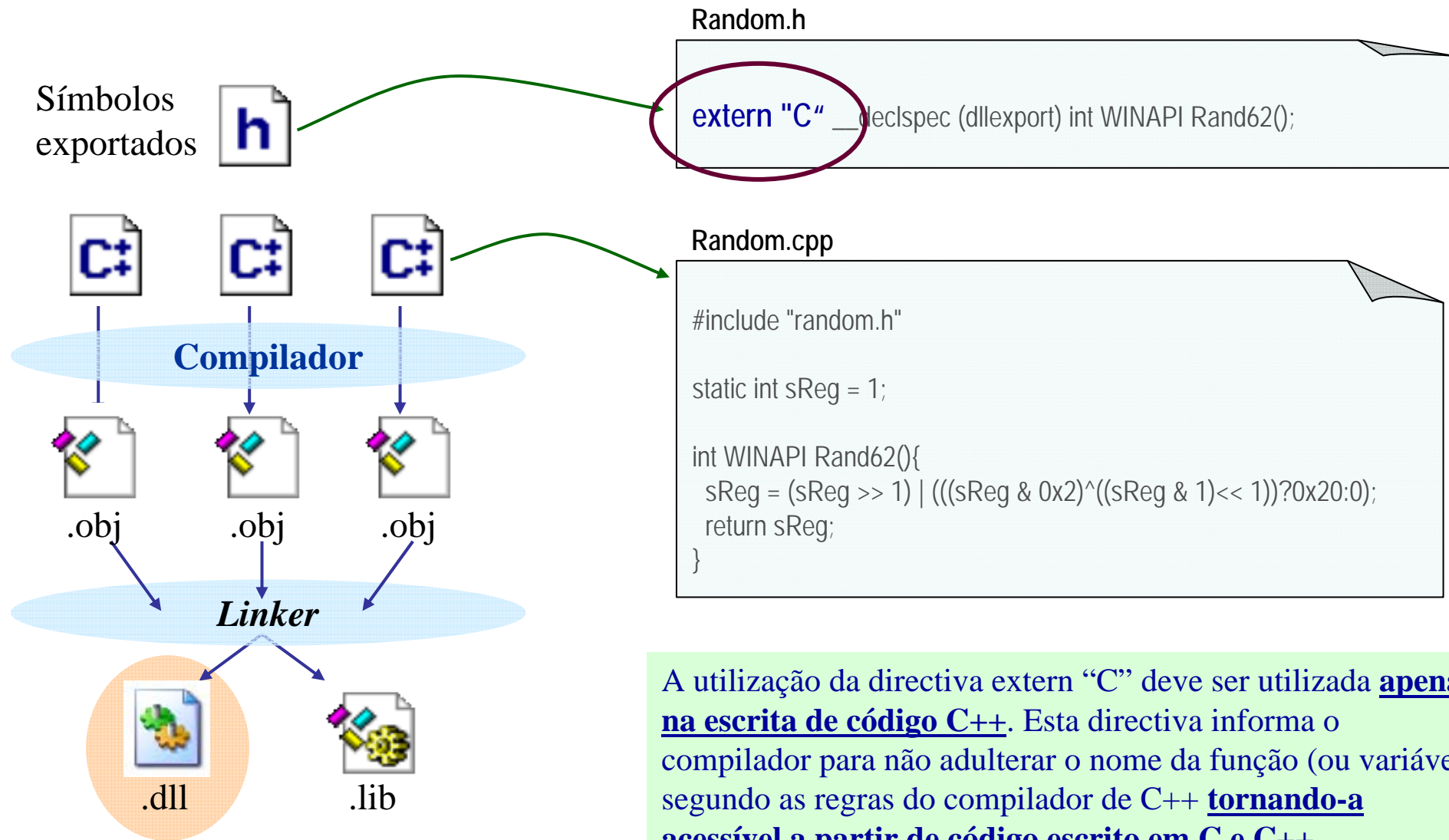
Como pode observar o nome da função exportada é:

`?Rand62@@YGHXZ` (*mangled names*)

Deve-se à forma como os compiladores C/C++ definem os nomes das funções. No caso do compilador da Microsoft o nome das funções é formado por:

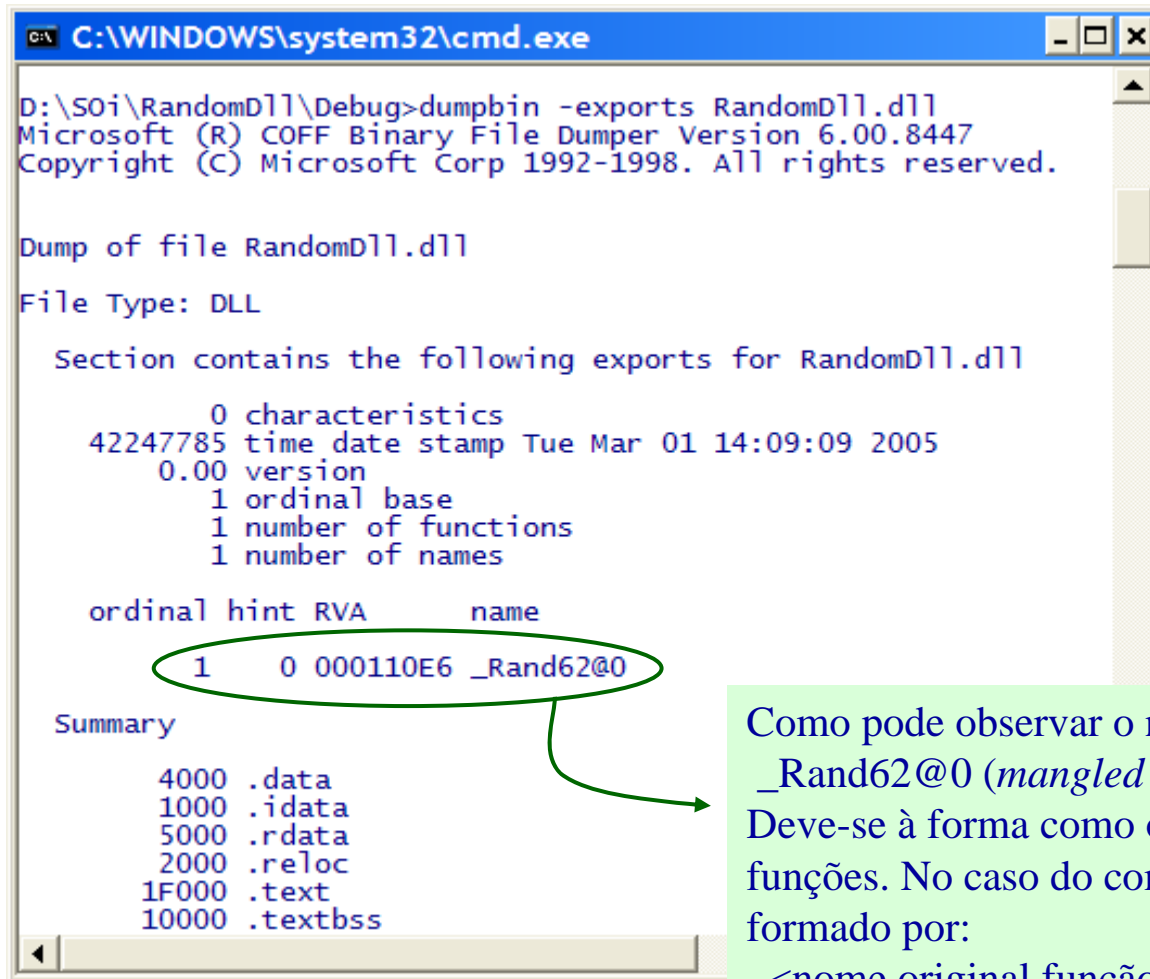
`?<nome original função>@@<informação sobre os argumentos>`

Construção de uma DLL



Construção de uma DLL

Utilizando, novamente, o utilitário dumpbin com a opção `-exports` podemos observar a tabela de símbolos exportados: `dumpbin -exports RandomDll.dll`



```
C:\WINDOWS\system32\cmd.exe
D:\SOi\RandomDll\Debug>dumpbin -exports RandomDll.dll
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file RandomDll.dll
File Type: DLL

Section contains the following exports for RandomDll.dll

 0 characteristics
42247785 time date stamp Tue Mar 01 14:09:09 2005
 0.00 version
 1 ordinal base
 1 number of functions
 1 number of names

ordinal hint RVA      name
1      0 000110E6 _Rand62@0

Summary
 4000 .data
 1000 .idata
 5000 .rdata
 2000 .reloc
1F000 .text
10000 .textbss
```

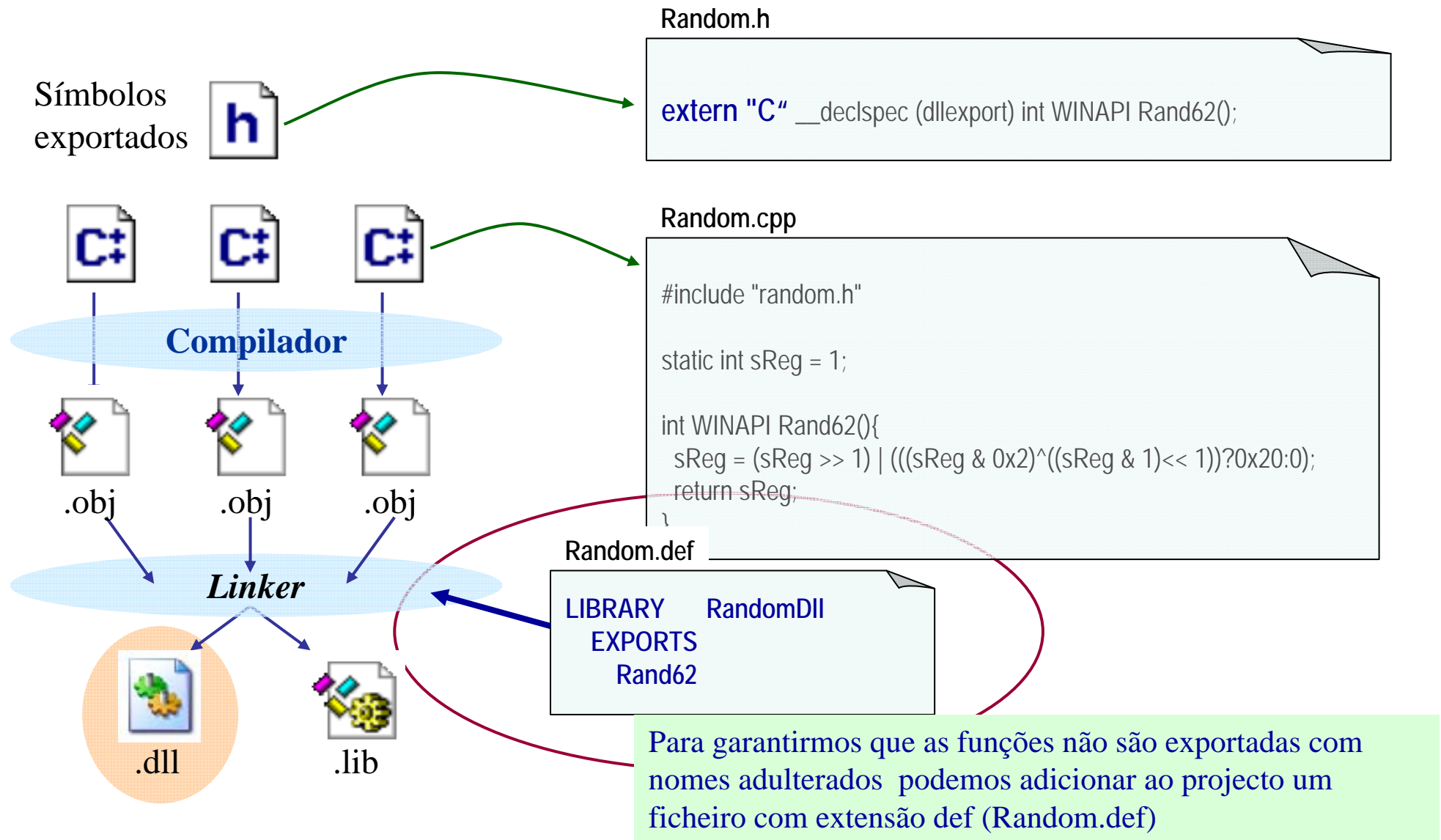
Como pode observar o nome da função exportada é:

`_Rand62@0` (*mangled names*)

Deve-se à forma como os compiladores C/C++ definem os nomes das funções. No caso do compilador da Microsoft o nome das funções C é formado por:

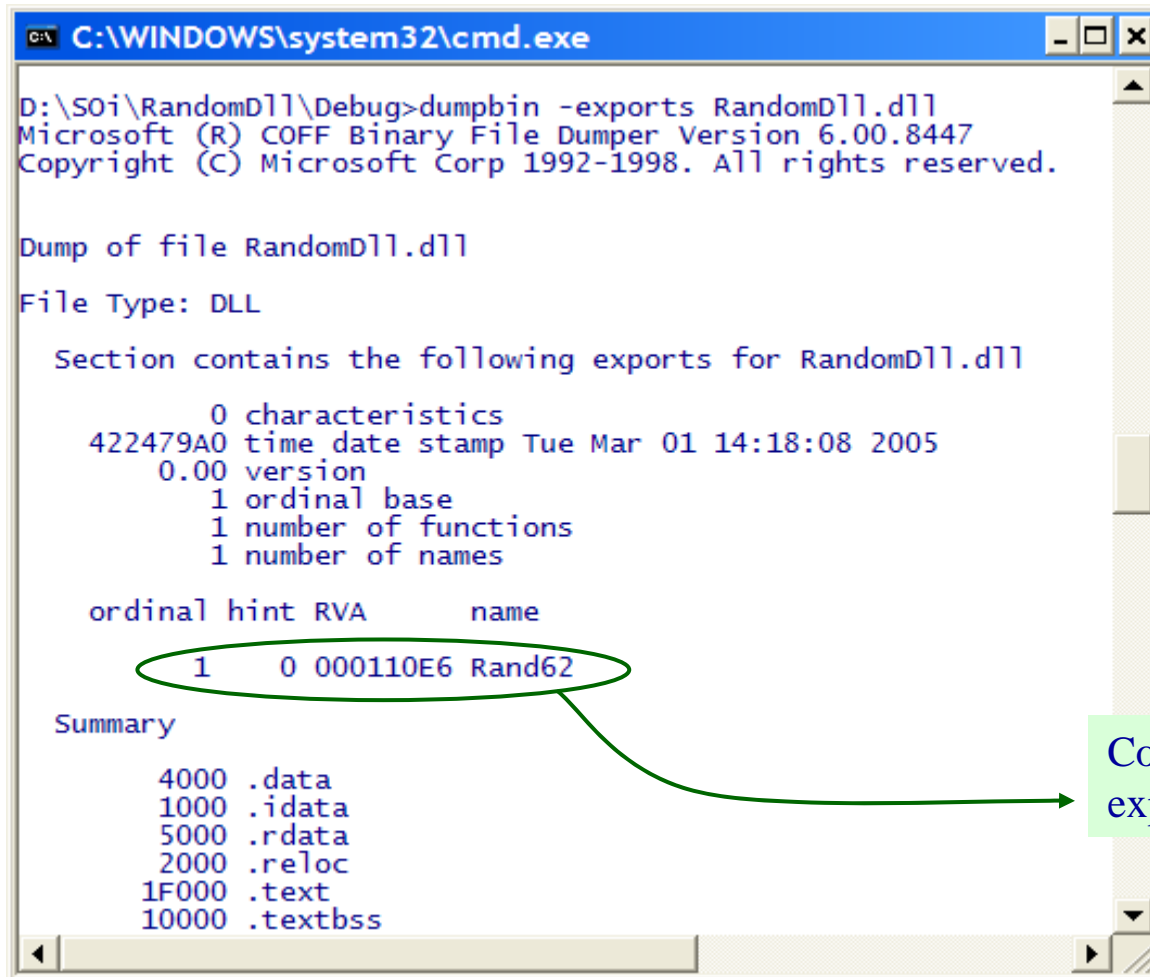
`<nome original função>@<dimensão (bytes) dos argumentos>`

Construção de uma DLL



Construção de uma DLL

Utilizando, novamente, o utilitário dumpbin com a opção `-exports` podemos observar a tabela de símbolos exportados: `dumpbin -exports RandomDll.dll`



```
C:\WINDOWS\system32\cmd.exe
D:\SOI\RandomDll\Debug>dumpbin -exports RandomDll.dll
Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file RandomDll.dll
File Type: DLL

Section contains the following exports for RandomDll.dll

   0 characteristics
422479A0 time date stamp Tue Mar 01 14:18:08 2005
   0.00 version
   1 ordinal base
   1 number of functions
   1 number of names

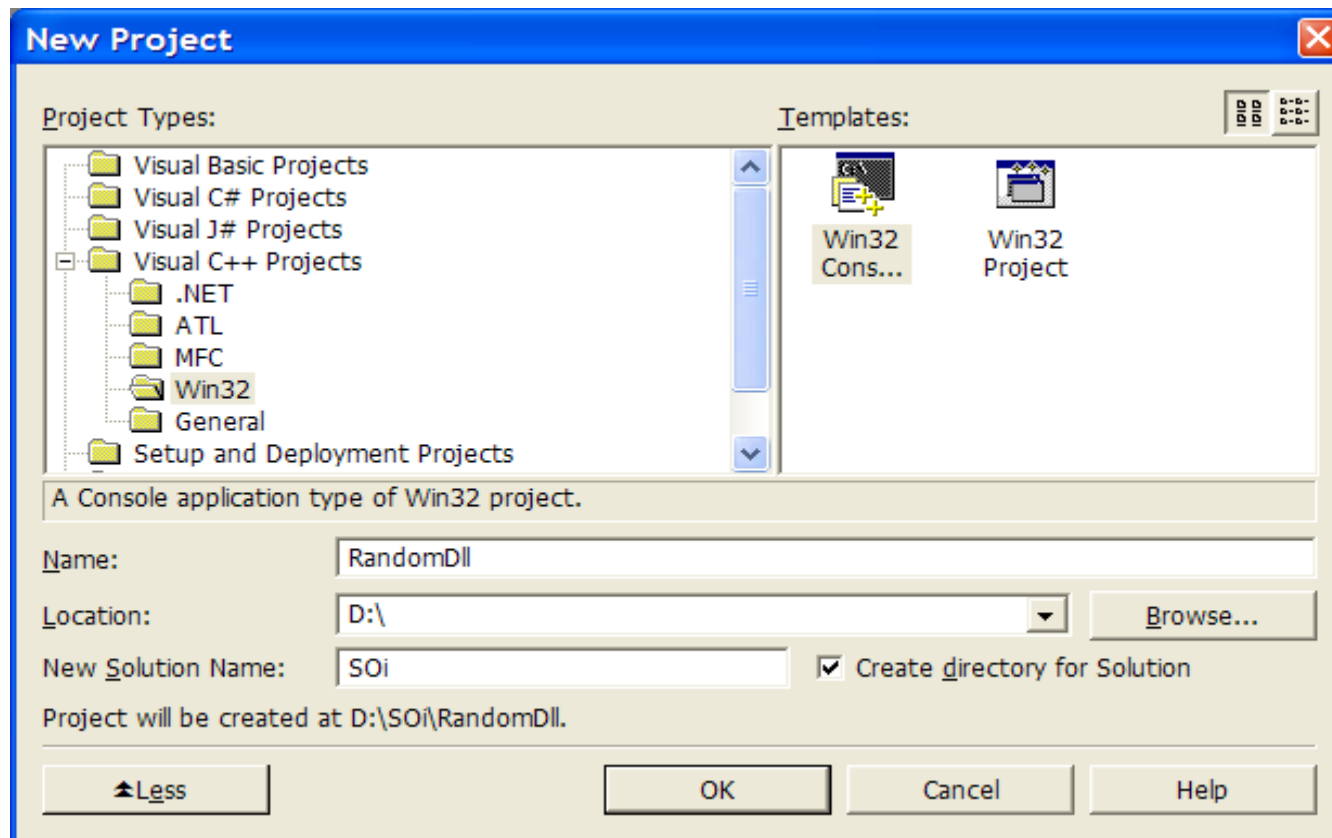
ordinal hint RVA      name
   1      0 000110E6 Rand62

Summary
 4000 .data
 1000 .idata
 5000 .rdata
 2000 .reloc
1F000 .text
10000 .textbss
```

Como pode observar o nome da função exportada agora é: Rand62

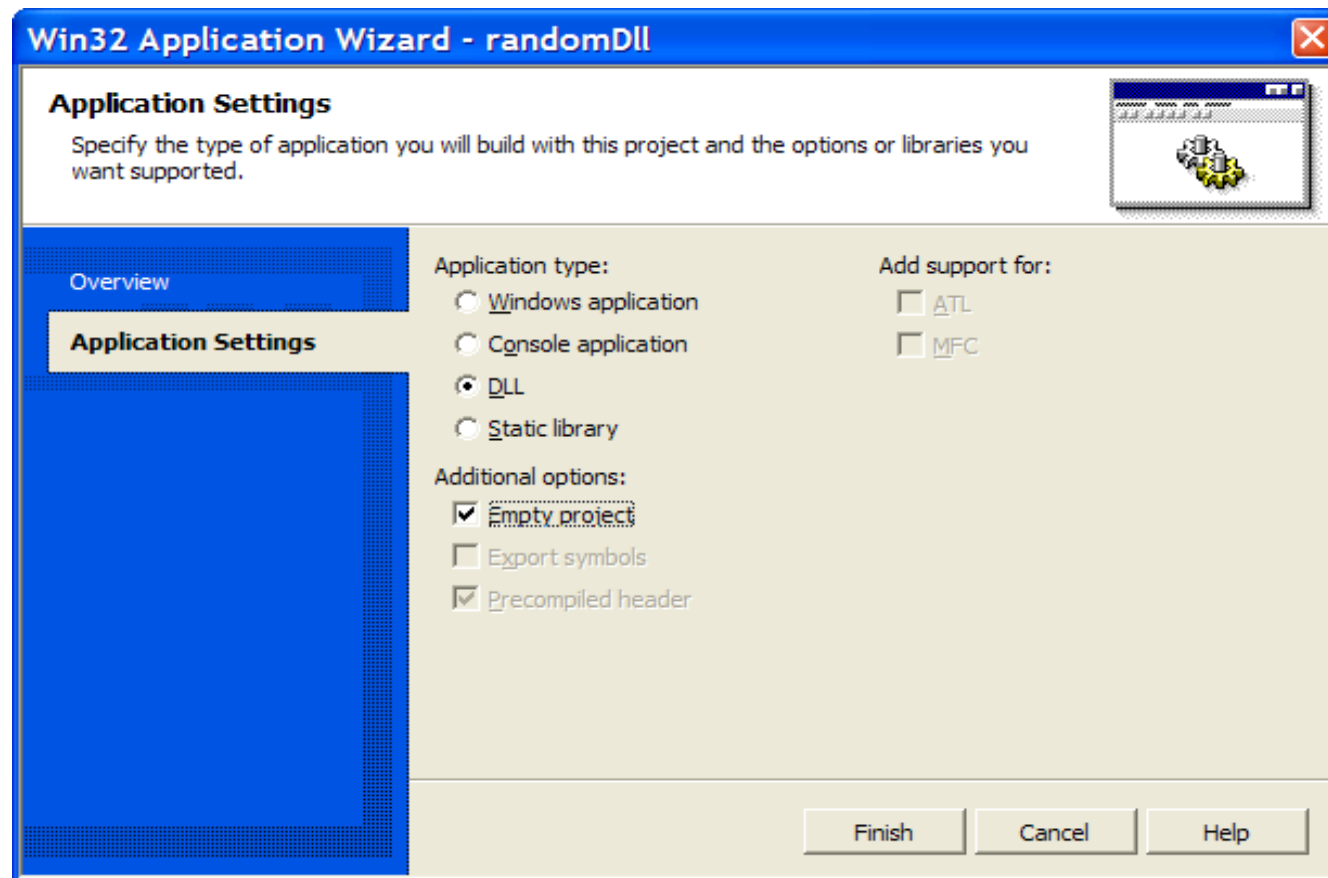
Construção de uma DLL – Visual Studio

No Visual Studio .Net 2003 criar um novo projecto do tipo Win32

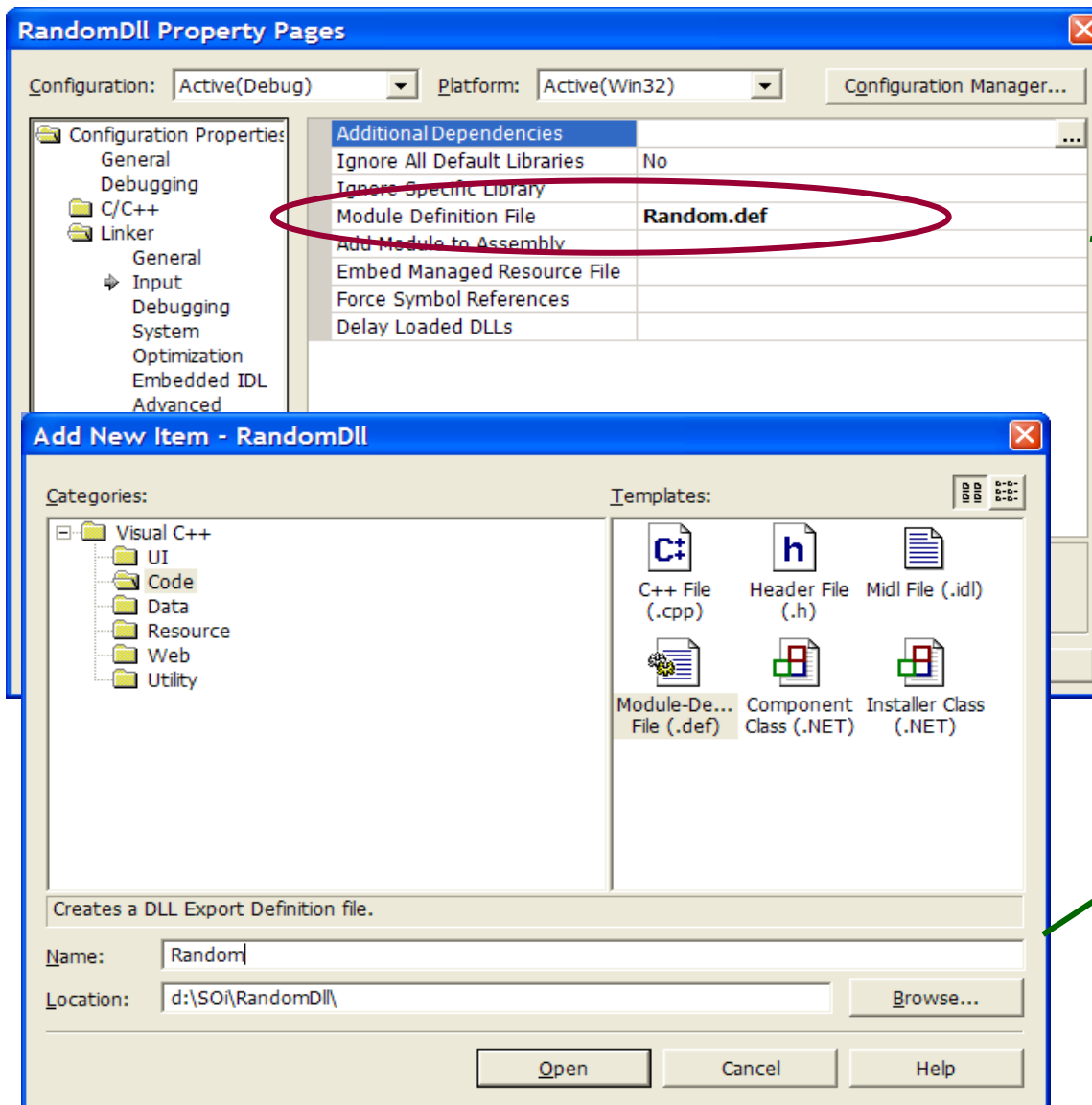


Construção de uma DLL – Visual Studio

Na janela de diálogo seguinte seleccionar *Applications Settings* e escolher DLL como o tipo da aplicação



Construção de uma DLL – Visual Studio



Para adicionar o ficheiro .def ao projecto deve-se ter o cuidado de verificar se nas configurações do projecto, relativas ao *linker*, existe a referência a esse ficheiro.

Esta definição é adicionada automaticamente se utilizarmos o visual studio para a criação do ficheiro .def (Project->Add New Item e escolher um ficheiro do tipo .def)

Carregamento de DLL

O Carregamento de uma DLL pode ser feito de duas maneiras:

- **Carregamento automático ou implícito**

- O programa é compilado e *ligado* com o **.LIB** usado para gerar a **DLL**.
- A biblioteca é carregada automaticamente em memória quando a aplicação corre (desde que o ficheiro da biblioteca tenha terminação **.DLL**). Ex.:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom) ; // Ligado com GDI32.LIB
```

- **Carregamento explícito**

- Usando as funções **LoadLibrary**, **GetProcAddress**, **FreeLibrary** .
- A biblioteca apenas é carregada e libertada a pedido do programa.
- Útil quando apenas sabemos o nome da biblioteca em tempo de execução. Ex.:

```
HANDLE hLibrary;  
PFNRECT pfnRectangle;
```

```
hLibrary = LoadLibrary(TEXT("GDI32.DLL"));  
pfnRectangle = (PFNRECT) GetProcAddress(hLibrary, TEXT("Rectangle"));  
pfnRectangle(hdc, xLeft, yTop, xRight, yBottom);  
FreeLibrary(hLibrary);
```

Carregamento de DLL - implícito

Construção da DLL

Símbolos
exportados



Compilador



.obj

.obj

.obj

Linker



.dll



.lib

Construção do EXE



Símbolos
importados



Compilador



.obj

.obj

.obj

Linker



.exe

Carregamento de DLL - implícito em C++

RandomImport.h

```
#include <windows.h>
extern "C" __declspec (dllimport) int WINAPI Rand62();
```

GeraAleatorios.cpp

```
#include "randomImport.h"
#include <stdio.h>

void main()
{
    for (int i=0; i<12; i++) printf("%d\n", Rand62());
    getchar();
}
```

Construção do EXE

 Símbolos importados



Compilador



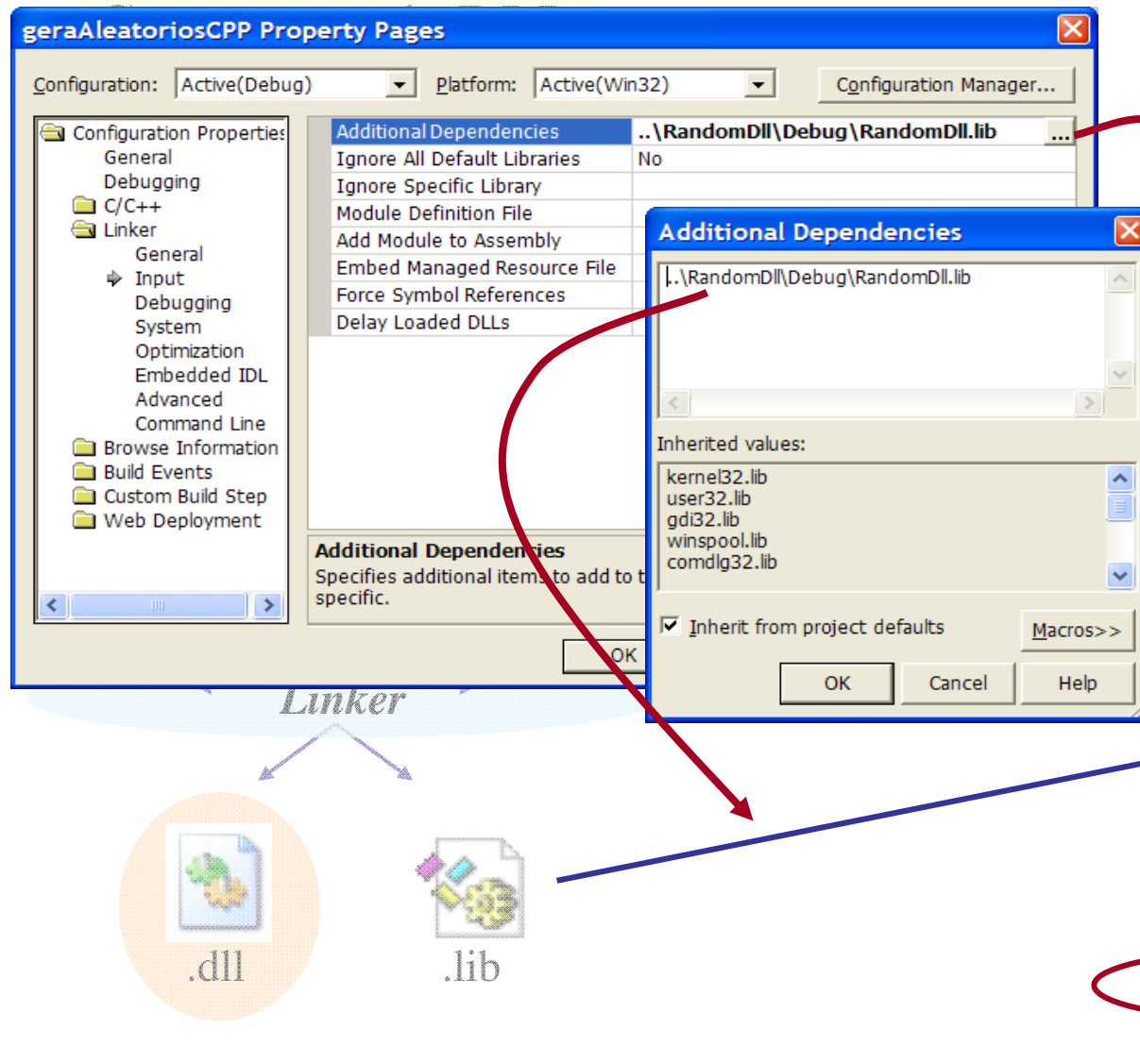
Linker

 .exe

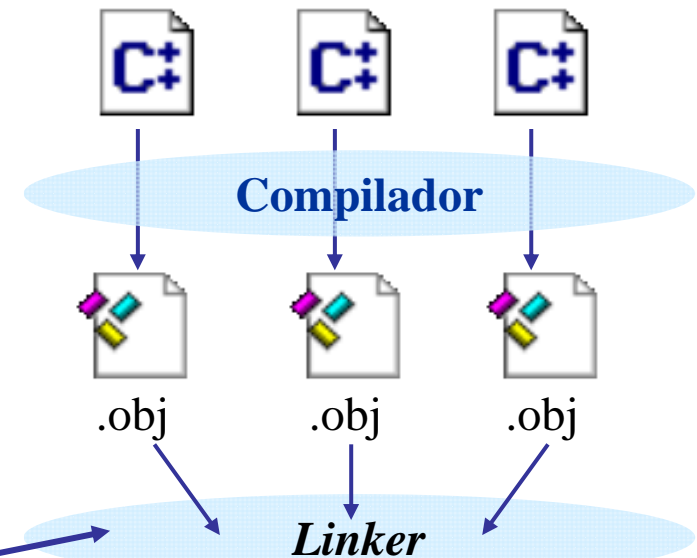
Nota: A DLL pode ter sido compilada em C ou C++. No caso de ter sido compilada em C++ foi utilizada a declaração extern "C".

Carregamento de DLL - implícito em C++

Construção do EXE



Símbolos importados



Carregamento de DLL - implícito em C

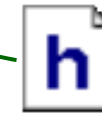
RandomImport.h

```
#include <windows.h>  
extern "C" __declspec (dllimport) int WINAPI Rand62();
```

GeraAleatorios.c

```
#include "randomImport.h"  
#include <stdio.h>  
  
void main()  
{  
    for (int i=0; i<12; i++) printf("%d\n", Rand62());  
    getchar();  
}
```

Construção do EXE



Símbolos importados



Compilador



.obj

.obj

.obj

Linker



.exe

Nota: A declaração do símbolo importado não pode conter a declaração extern "C", uma vez que **não é** uma declaração válida para o compilador de C

Carregamento de DLL – implícito: Ficheiro único de definições

```
#ifndef __randomdll__
#define __randomdll__

#ifdef __cplusplus
extern "C" {
#endif // __cplusplus

#include <windows.h>
#define EXPORT __declspec (dllexport)
#define IMPORT __declspec (dllimport)

#ifdef randomdll_build

EXPORT int WINAPI Rand62();

#else // Definições de carregamento implícito

IMPORT int WINAPI Rand62();

#endif // randomdll_build

#ifdef __cplusplus
} //extern "C" {
#endif // __cplusplus

#endif // __randomdll__
```

Quando se cria uma DLL deve-se definir o ficheiro de definições (*header File*) contendo as declarações de:

1. Exportações a ser utilizado no código fonte da DLL:

- Variáveis (tipo e nomes) a exportar
- Funções (protótipos) a exportar
- Símbolos e estruturas utilizados nas funções e variáveis exportadas

2. Importações a ser utilizado pelo código que utilize a DLL

A existência de um único ficheiro de definições facilita a manutenção

Carregamento de DLL - explícito

Construção da DLL

Símbolos
exportados



Compilador



.obj

.obj

.obj

Linker



.dll



.lib

Construção do EXE



Símbolos
importados



Compilador



.obj

.obj

.obj

Linker



.exe

A aplicação carrega
explicitamente a DLL através
da função `loadlibrary`

Carregamento de DLL - explícito

Funções utilizadas no carregamento explícito

```
HMODULE LoadLibrary( LPCTSTR lpFileName );
```

```
FARPROC GetProcAddress( HMODULE hModule,  
                        LPCSTR lpProcName );
```

```
BOOL FreeLibrary( HMODULE hModule );
```

Carregamento de DLL – explícito: Ficheiro único de definições

```
#ifndef __randomdll__
#define __randomdll__

#ifdef __cplusplus
extern "C" {
#endif // __cplusplus
#include <windows.h>
#define EXPORT __declspec (dllexport)
#define IMPORT __declspec (dllimport)

#ifdef randomdll_build
    EXPORT int WINAPI Rand62();
#else // Definições de carregamento implícito
    IMPORT int WINAPI Rand62();
#endif // randomdll_build

//Definições para carregamento explícito com a função LoadLibray:
typedef int (WINAPI *TpRand62)();

#ifdef __cplusplus
} //extern "C" {
#endif // __cplusplus

#endif // __randomdll__
```

O ficheiro de definições da DLL deve conter as definições de tipos para os símbolos exportados que serão utilizados quando se optar pelo carregamento explícito da DLL

Carregamento de DLL – explícito: exemplo

```
#include "..\randomDll\random.h"
#include <stdio.h>

void main() {
    TpRand62 pfuncRand62 = NULL;

    HINSTANCE hRandomDll = LoadLibrary("RandomDll");
    if (hRandomDll == NULL) {
        printf("Erro ao carregar a DLL: %d\n", GetLastError());
        getchar(); return;
    }
    printf("Biblioteca RandomDll carregada\n");

    pfuncRand62 = (TpRand62)GetProcAddress(hRandomDll, "Rand62");
    if (pfuncRand62 == NULL) {
        printf("Erro ao determinar endereço função Rand62: %d\n", GetLastError());
        getchar(); return;
    }
    printf("Endereço da função Rand62 determinado\n");

    for (int i=0; i<12; i++) printf("%d\n", pfuncRand62());

    FreeLibrary(hRandomDll);
}
```


Biblioteca de Resources

- Contém recursos embora possam, também, conter funções.
- No caso de não terem funções (mais comum), não geram .LIB e tem de ser carregadas explicitamente.
- Ex.: Font files, Clipart files, Icon Files, Textos dos diálogos numa dada língua, etc.

```
// Biblioteca MYBITMAPLIB.DLL
```

```
MYBITMAPLIB.RC (excerpts)
```

```
//Microsoft Developer Studio generated resource script.
```

```
#include "resource.h"
```

```
#include "afxres.h"
```

```
////////////////////////////////////
```

```
// Bitmap
```

```
1          BITMAP DISCARDABLE "bitmap1.bmp"
2          BITMAP DISCARDABLE "bitmap2.bmp"
3          BITMAP DISCARDABLE "bitmap3.bmp"
4          BITMAP DISCARDABLE "bitmap4.bmp"
5          BITMAP DISCARDABLE "bitmap5.bmp"
```

```
// Program TEST_MYBITMAPLIB.CPP
```

```
HINSTANCE hLibrary ;
```

```
HBITMAP hBitmap ;
```

```
hLibrary = LoadLibrary (TEXT ("BITLIB.DLL"))
```

```
...
```

```
if (hLibrary) {
```

```
    hBitmap = LoadBitmap (hLibrary, MAKEINTRESOURCE (2)) ;
```

```
    if (hBitmap) {
```

```
        DrawBitmap (hdc, 0, 0, hBitmap) ;
```

```
        DeleteObject (hBitmap) ;
```

```
    }
```

```
}
```

```
...
```

```
if (hLibrary)
```

```
    FreeLibrary (hLibrary) ;
```

Resumo - Garantir o sucesso de integração da DLL noutras aplicações ou DLL's

- Especificar, explicitamente, a convenção de chamadas às funções exportadas: `__cdecl` (usado por omissão nos compiladores C/C++); `__stdcall` (sinónimos `APIENTRY`, `CALLBACK`, `WINAPI`).
- Fazer o ficheiro de include compatível com os compiladores de C e de C++:

```
#ifdef __cplusplus
    extern "C" {
#endif

. . .
#ifdef __cplusplus
    } //extern "C" {
#endif
```
- Simplificar a importação das variáveis e funções da DLL fornecendo um ficheiro de include que especifique `__declspec(dllexport)`:

```
#define EXPORT __declspec (dllexport)
#define IMPORT __declspec (dllimport)
#ifdef dll_build
    EXPORT void WINAPI aFunction();
#else
    IMPORT void WINAPI aFunction();
#endif
```
- O símbolo `dll_build` só será definido para compilação do código fonte da DLL.
- Fornecer definições de tipos de apontadores para funções (typedef) para facilitar a evocação das funções quando a DLL for carregada implicitamente (`Loadlibrary` e `GetProcAddress`).
- Adicionar um ficheiro de definições ao projecto da DLL para definir os nome dos símbolos exportados sem que estes apareçam alterados pelo compilador (evitar *mangled names*).

DLL – Função *Entry-Point DllMain*

```
BOOL WINAPI DllMain ( HINSTANCE hinstDLL,  
                      DWORD      fdwReason,  
                      LPVOID      lpReserved );
```

hinstDLL: Handle da instância da DLL em memória

fdwReason: Indica a razão da função DllMain ter sido chamada:

DLL_PROCESS_ATTACH

DLL_THREAD_ATTACH

DLL_THREAD_DETACH

DLL_PROCESS_DETACH

lpReserved: Reservado pelo sistema

- Esta função pode não ser definida, neste caso o *linker* cria uma que apenas faz o retorno do valor TRUE
- O valor de retorno de DllMain apenas é importante se o fdwReason tiver o valor DLL_PROCESS_ATTACH. Nesse caso, e se a função retornar FALSE, o Windows impede a DLL de ser carregada

DLL – Função *Entry-Point DllMain*

Parâmetro `fdwReason`

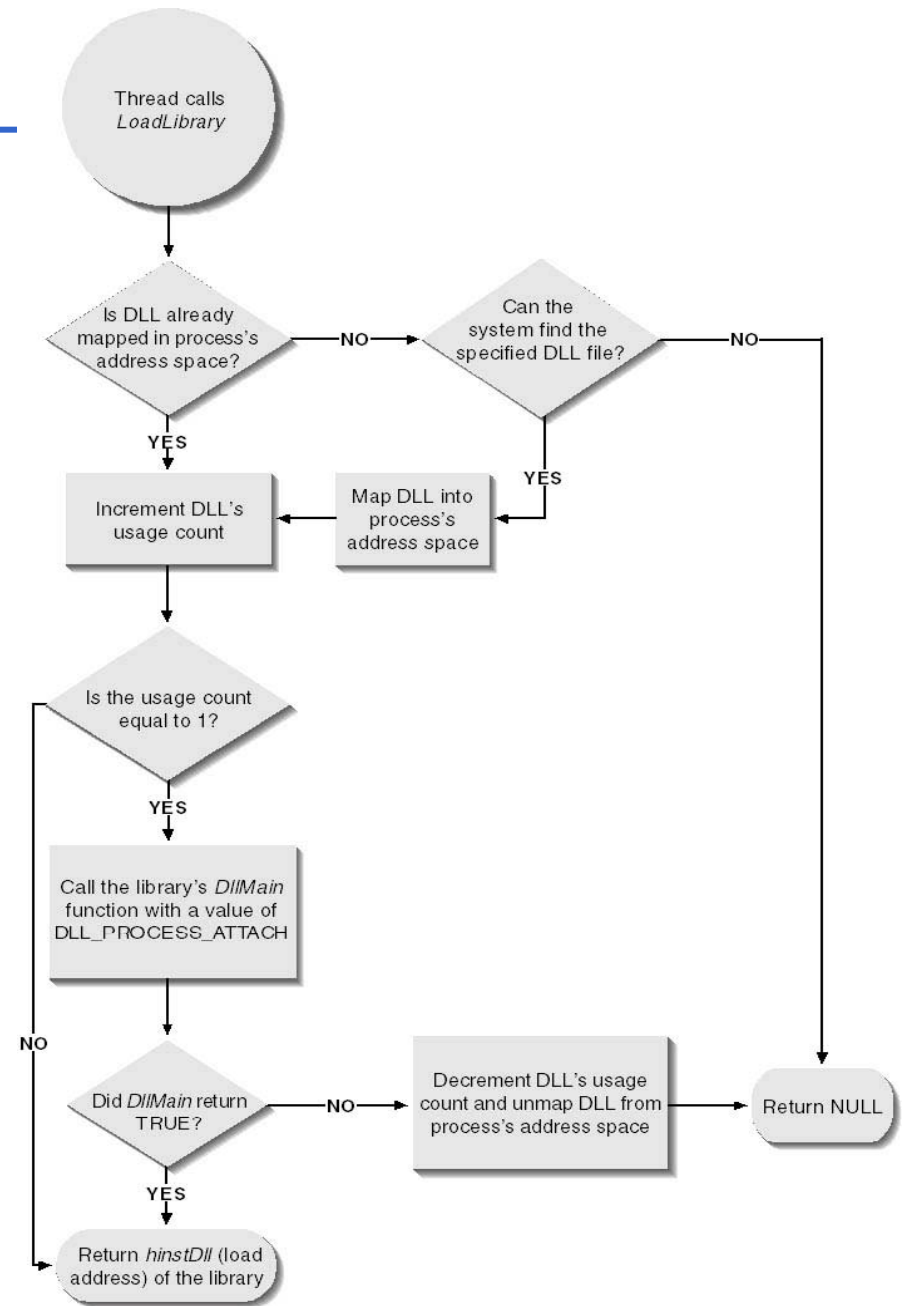
- `DLL_PROCESS_ATTACH`
 - Quando o código de um processo que foi construído com a LIB da DLL é carregado ou então quando se invoca a função `LoadLibrary`.
 - `DLL_THREAD_ATTACH`
 - Quando um *thread* é criado no processo onde a DLL está mapeada. **Atenção:** Se na altura em que a DLL é mapeada para um processo este já tem *threads* criados, então **DllMain** não é chamada para estes.
 - `DLL_THREAD_DETACH`
 - Quando um *thread* existente no processo onde a DLL está mapeada termina, retornando da sua função ou chamando `ExitThread`.
 - `DLL_PROCESS_DETACH`
 - Quando a DLL vai ser removida do mapeamento do processo.
-
- No caso de um processo gerar um grande número de *threads*, podemos não querer que **DllMain** seja chamada inúmeras vezes com a razão `DLL_THREAD_ATTACH` e `DLL_THREAD_DETACH`. Nesse caso podemos usar a função
`DisableThreadLibraryCalls(HMODULE hModule);`
 - **Atenção:** As funções `TerminateThread` e `TerminateProcess` não chamam **DllMain** com `DLL_THREAD_DETACH` ou `DLL_PROCESS_DETACH`.

DLL – Função *Entry-Point DllMain*

```
BOOL WINAPI DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved )
{
    BOOL Result = TRUE;
    switch (fdwReason) {
        case DLL_PROCESS_ATTACH:
            // A DLL está a ser ligada (mapped) no espaço de endereçamento do processo
            // Fazer iniciações de variáveis por processo aqui. Por exemplo criar as entradas TLS.
            // Se as iniciações falharem e se quiser fazer com que a DLL não seja carregada,
            // alterar Result para FALSE.
            break;
        case DLL_THREAD_ATTACH:
            // Está a ser criada uma thread. Fazer iniciações por thread aqui
            break;
        case DLL_THREAD_DETACH:
            // Um thread está a terminar a sua execução. Fazer cleanup por thread aqui
            break;
        case DLL_PROCESS_DETACH:
            // A DLL está a ser desligada (unmapped) do espaço de endereçamento do processo. Fazer
            // acções de terminação relativas a esta DLL antes de ser desligada de um processo
            break;
    }
    return Result; // Utilizado apenas para DLL_PROCESS_ATTACH
}
```

DLL – Função *Entry-Point DllMain*

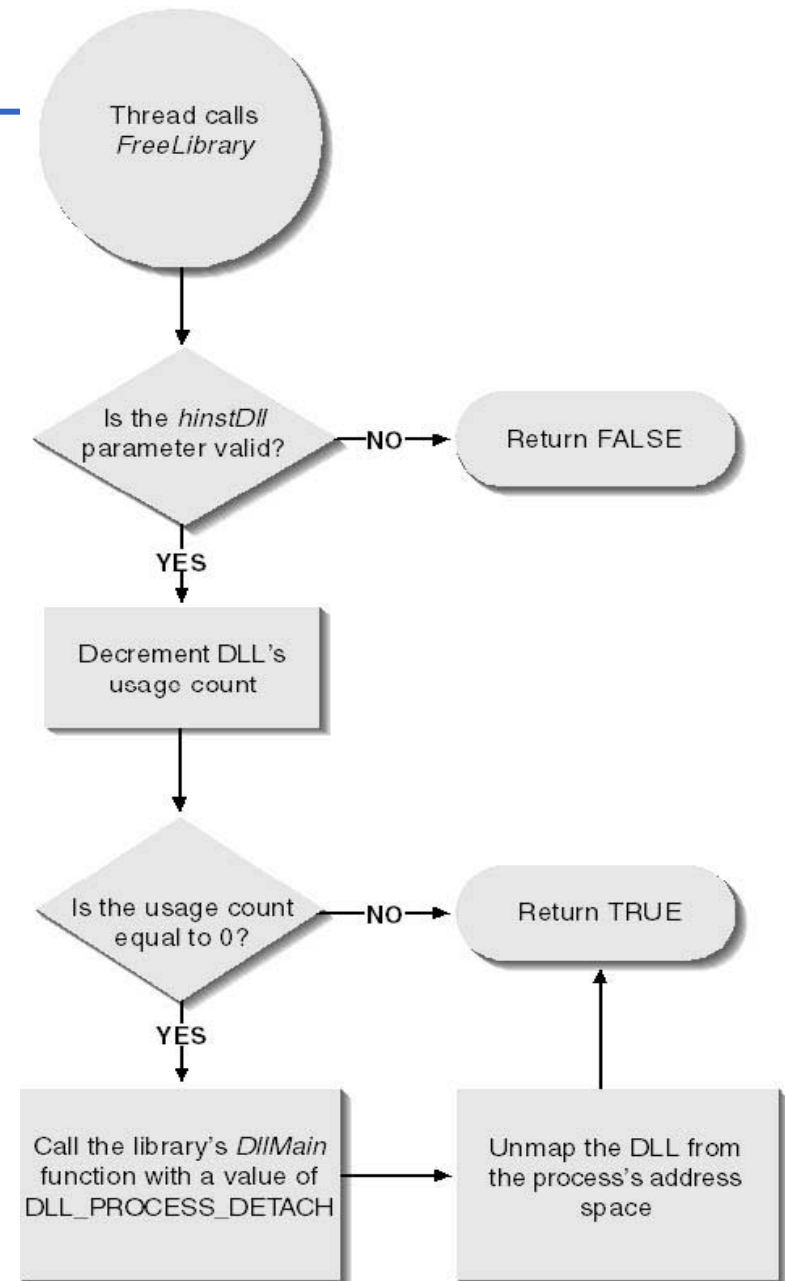
Passos realizados pelo sistema quando uma *thread* chama a função **LoadLibrary**



Fonte: Jeffrey Richter, *Programming Applications for Microsoft Windows*, 4th edition, cap. 19 e 20

DLL – Função *Entry-Point DllMain*

Passos realizados pelo sistema quando uma *thread* chama a função **FreeLibrary**



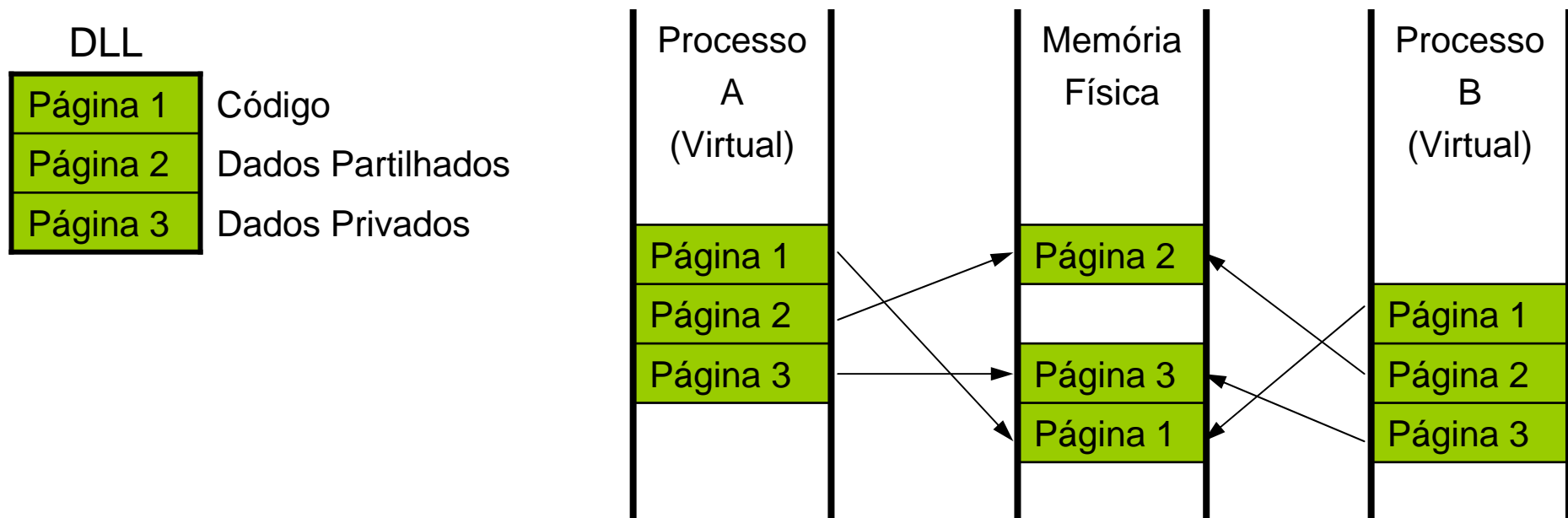
Fonte: Jeffrey Richter, *Programming Applications for Microsoft Windows*, 4th edition, cap. 19 e 20

DLL – Acesso à Função *Entry-Point DllMain*

- O Windows tem um MUTEX ligado a cada processo por forma a que cada *thread* desse processo tenha de adquirir o MUTEX antes desse *thread* poder chamar a função **DllMain**.
 - Se a função **DllMain** criar algum *thread*, esta função não pode ficar parada à espera de um sinal desse *thread* antes de continuar, senão entraremos num encravamento (o novo *thread* vai tentar chamar **DllMain** com a razão `DLL_THREAD_ATTACH` e vai ficar preso no MUTEX).
- Se a função **DllMain** precisar fazer alguma iniciação a variáveis de um segmento *shared*, é necessária a utilização de uma *flag* e um MUTEX para que esta apenas seja feita uma vez.
 - Essa iniciação é feita quando a função é chamada com a razão `DLL_PROCESS_ATTACH`. A *flag* destina-se a indicar que as variáveis foram já iniciadas. Pode-se dar o caso de dois processos chamarem **DllMain** com `DLL_PROCESS_ATTACH` ao mesmo tempo e ambos tentarem iniciar as variáveis; daí a necessidade do MUTEX.

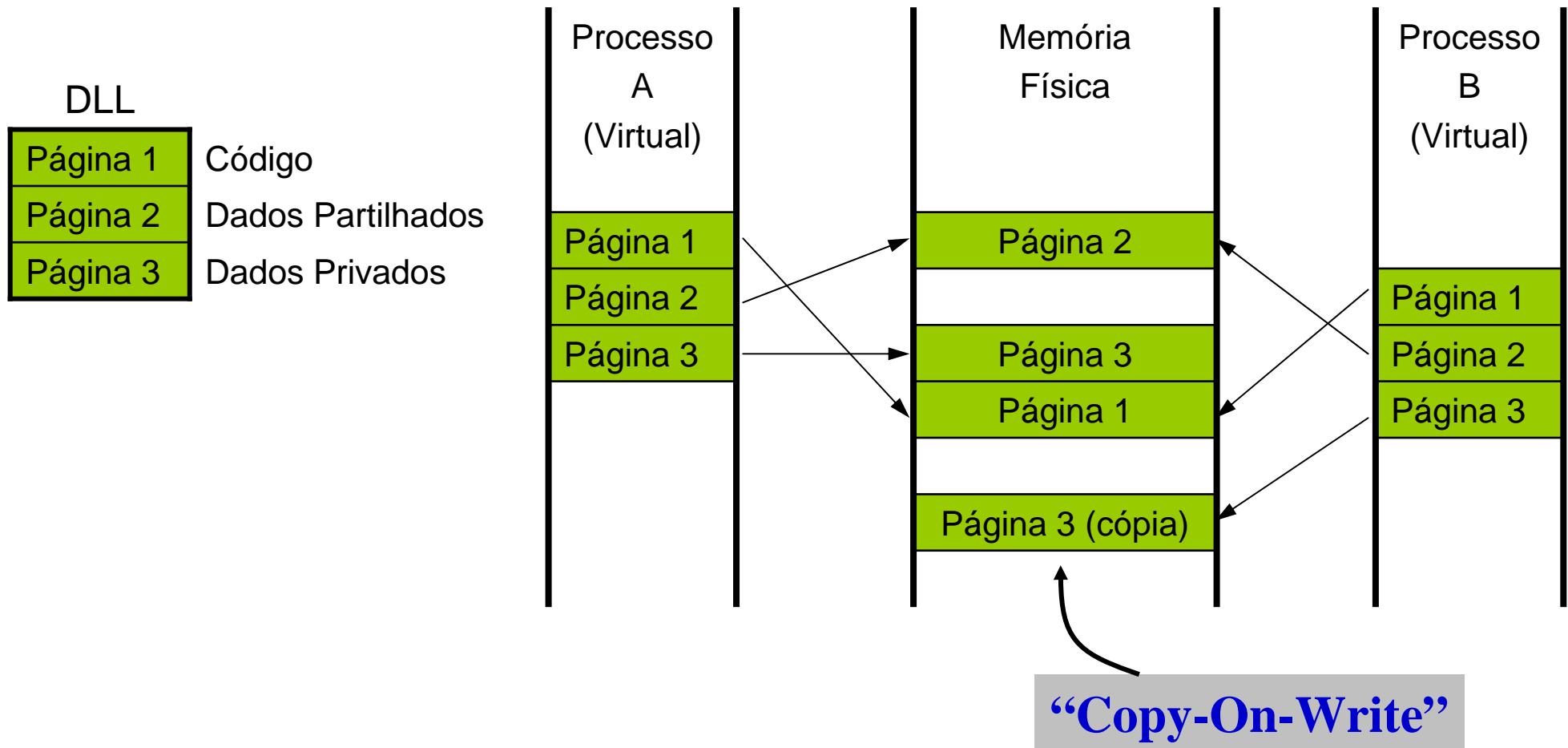
DLL - Espaço de Endereçamento do Processo

- Nas DLLs de 16 bits, todas as aplicações tinham acesso de leitura e escrita nas variáveis da DLL (all shared), existindo apenas uma cópia de código e dados em memória.
- Nas DLLs de 32 bits, apesar de também existir apenas uma cópia em memória, assim que alguma variável é escrita, a aplicação passa a ter a sua própria cópia das variáveis da DLL: Copy-On-Write



DLL - Espaço de Endereçamento do Processo

Se o processo B altera alguma das variáveis privadas da DLL, por exemplo na Página 3:



DLL - Espaço de Endereçamento do Processo

Gestão de DLLs pelo Windows:

- DLLs 16 bits: O Windows tem um contador de referências para a DLL
- DLLs 32 bits: O Windows tem dois contadores:
 - Um que conta o número de processos que referenciam a DLL
 - Um por cada processo, que conta o número de referências dentro desse processo

Tempo de Vida das variáveis da DLL:

- Segmentos *Privados*: Desde o início do carregamento da DLL pelo processo até o último *thread* deste processo libertar a DLL

Ex:

```
int iNotShared = 0;           // Variável privada da DLL
```

- Segmentos *Shared*: Desde o início do carregamento da DLL pelo primeiro processo até o último *thread* do último processo libertar a DLL

Ex:

```
#pragma data_seg ("shared")  
int iShared = 0;              // Variável shared da DLL  
#pragma data_seg()
```

Thread Local Storage

TLS

TLS - *Thread Local Storage*

- Na programação em ambiente *multithreaded*, por vezes temos necessidade de organizar as variáveis do programa contextualizadas aos vários fios de execução que compõem um dado programa. Um exemplo disso é a biblioteca de “*run time*” do C. Por exemplo, a função `rand()` e `srand()`, baseiam-se numa variável global estática para guardar a sua memória de estado necessária ao algoritmo de geração de números aleatórios. A sequência de números aleatórios obtida através das sucessivas chamadas da função `rand()` pode ser influenciada (ou até mesmo estragada) se tivermos mais do que um fio de execução em concorrência.
- Para resolver este tipo de problemas, o SO oferece um mecanismo para providenciar o armazenamento de informação local a um fio de execução sem ser no seu *stack*. Esse mecanismo é designado por **TLS – *Thread Local Storage***. Recorre-se a este tipo de memória quando:
 - Não é possível recorrer a variáveis locais ao *thread* para a satisfação dos objectivos em vista, por ex. não temos controlo sobre a criação e estruturação do código dos *threads*.
 - Temos necessidade de migrar código, inicialmente desenvolvido para ser executado em mono-tarefa, para ambiente multi-tarefa.

TLS - *Thread Local Storage*

- Quando se cria um *thread* o SO reserva um *array* com **TLS_MINIMUM_AVAILABLE** entradas (64) de apontadores iniciados a NULL. Estes *arrays* fazem parte dos recursos do processo que contém os *threads*. Contudo para que um *thread* possa usar uma entrada do seu *array* **TLS** tem que criar/activar o índice para essa entrada. As funções para manipulação do **TLS** são:
 - **DWORD TlsAlloc(VOID);** Devolve um índice de entrada no **TLS** do *thread*. Serve portanto para obter uma entrada no *array* **TLS**;
 - **BOOL TlsFree(DWORD dwTlsIndex);** Elimina uma entrada criada pelo TlsAlloc. A usar no fim da execução do *thread* ou então a partir do momento em que este já não necessite desta entrada no *array* **TLS**;
 - **BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);** Armazena um valor (*lpTlsValue*) na entrada do *array* **TLS** cujo índice (*dwTlsIndex*) se especifica no argumento. Este índice foi obtido através do TlsAlloc;
 - **LPVOID TlsGetValue(DWORD dwTlsIndex);** Permite o acesso ao valor previamente armazenado no *array* **TLS** através da função TlsSetValue.

TLS - *Thread Local Storage*

- Quando um *thread* cria uma entrada no **TLS** através da função **TlsAlloc**, o índice devolvido destina-se a ser armazenado numa variável global do programa. Esse índice pode agora ser usado por todos os *threads* para acederem à sua informação local. Note-se que embora o valor do índice seja o mesmo ele corresponde a locais diferentes para cada *thread*. Se forem criados novos *threads* eles poderão também usar o índice, a única diferença é que a informação que este contém é um **NULL**.
- Sendo assim, um índice de entrada no **TLS** comporta-se como se fosse uma indexação a duas dimensões: Para além do índice no *array* é também necessário indicar qual o ID do *thread* para que assim se localize o apontador armazenado. As funções **TlsSetValue** e **TlsGetValue** encarregam-se de executar este mecanismo.

Array **TLS** de um
Processo WINDOWS

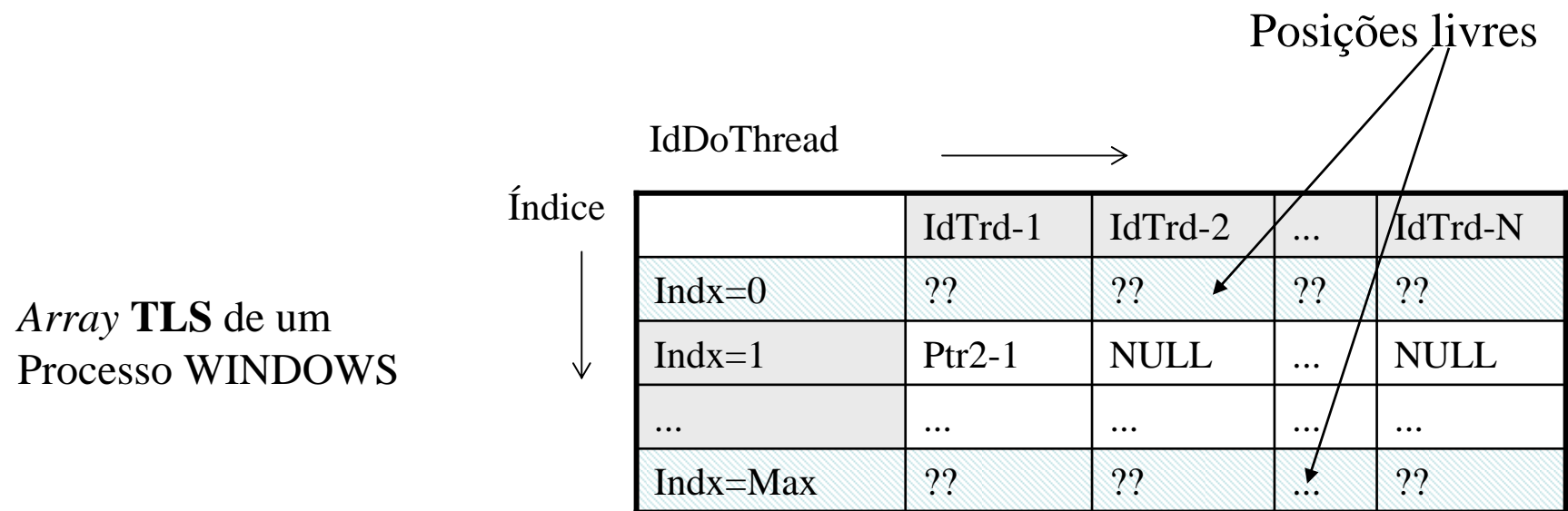
Índice

↓

	IdDoThread →				
		IdTrd-1	IdTrd-2	...	IdTrd-N
Indx=0		Ptr1-1	Ptr1-2	...	NULL
Indx=1		Ptr2-1	NULL	...	NULL
...	
Indx=Max		NULL	NULL	...	NULL

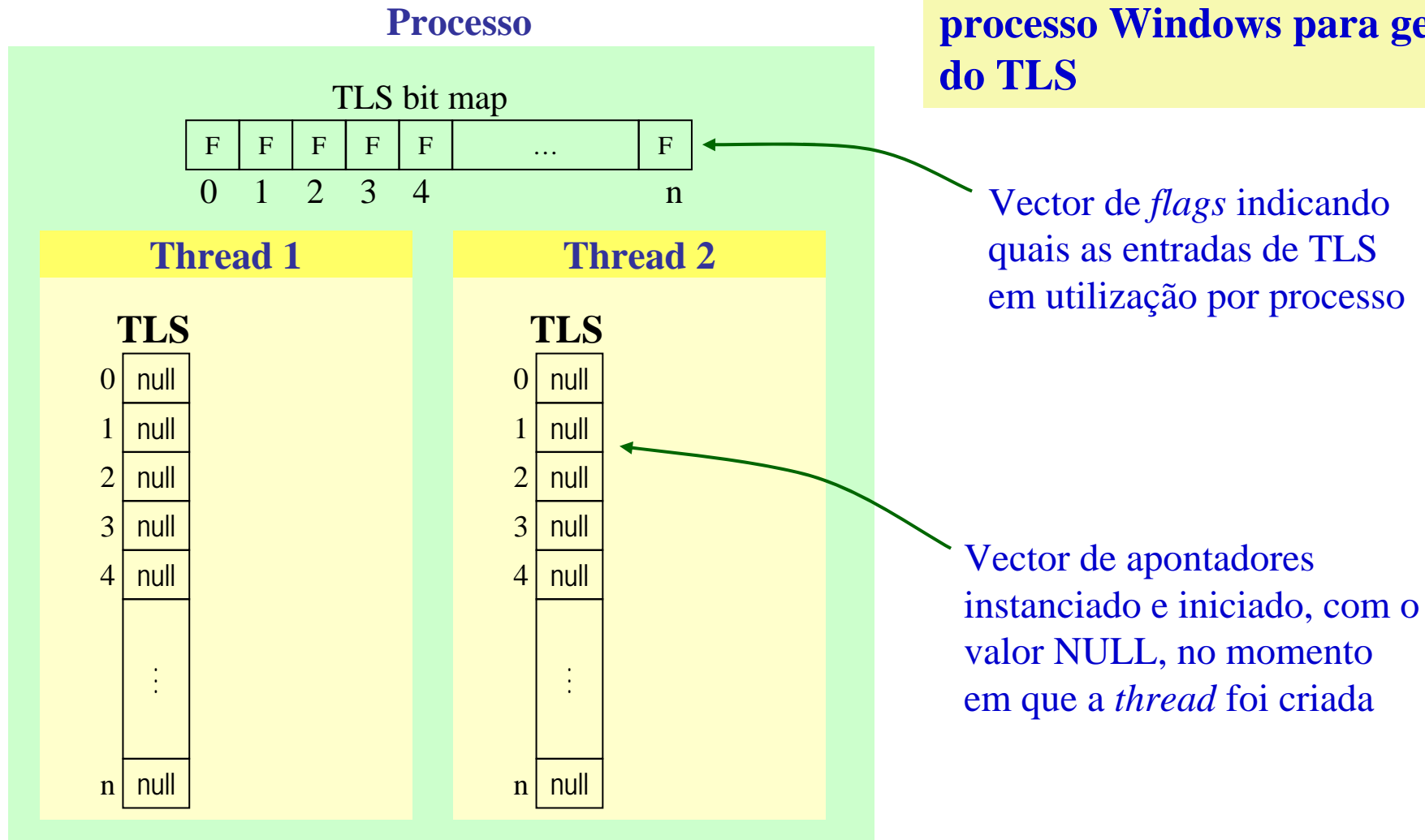
TLS - Thread Local Storage

- Sempre se se cria um *thread* é também inserida uma coluna iniciada a nulos no *array TLS*. De igual modo, sempre que termina um *thread* a coluna respectiva no *array TLS* é removida. Os índices do *array* que foram criados por um *thread*, através da função **TlsAlloc**, são geridos através de um mapa de bits (*bitmap*), que indica quais as entradas válidas para um dado processo. A função **Tlsfree** coloca o bit da entrada correspondente a *false*, fazendo com que daí para a frente a utilização desse índice resulte num erro de execução.



TLS - *Thread Local Storage*

Estruturas internas de um processo Windows para gestão do TLS



TLS - Thread Local Storage

Processo

TLS bit map

T	T	F	F	F	...	F
0	1	2	3	4		n

Thread 1

TLS

0	null
1	null
2	null
3	null
4	null
	⋮
n	null

Thread 2

TLS

0	null
1	null
2	null
3	null
4	null
	⋮
n	null

Exemplo de utilização

```
DWORD tlsIdx;
```

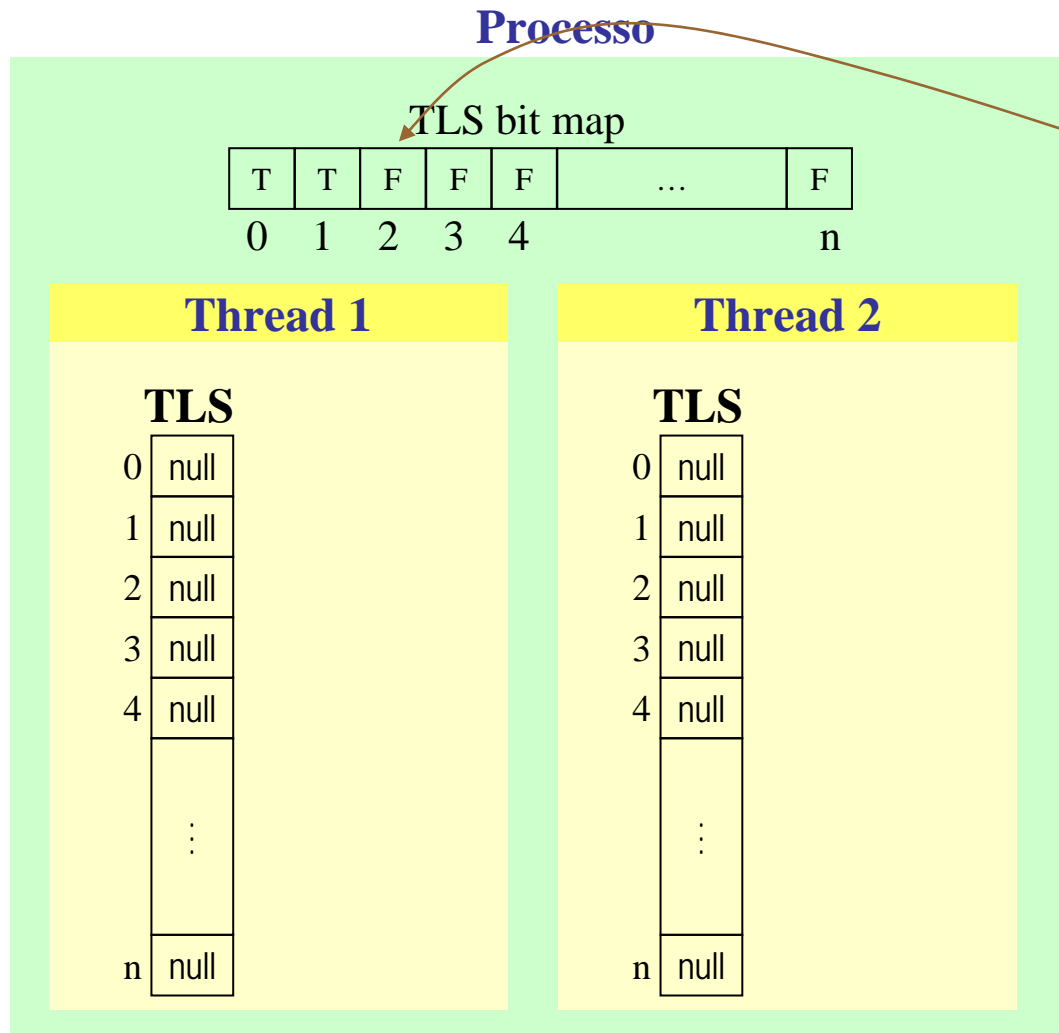
```
void main () {  
    tlsIdx = TlsAlloc();  
    ...  
}
```

```
thread1Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 3;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

```
thread2Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 5;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

TLS - Thread Local Storage

Exemplo de utilização



```
DWORD tlsIdx;
```

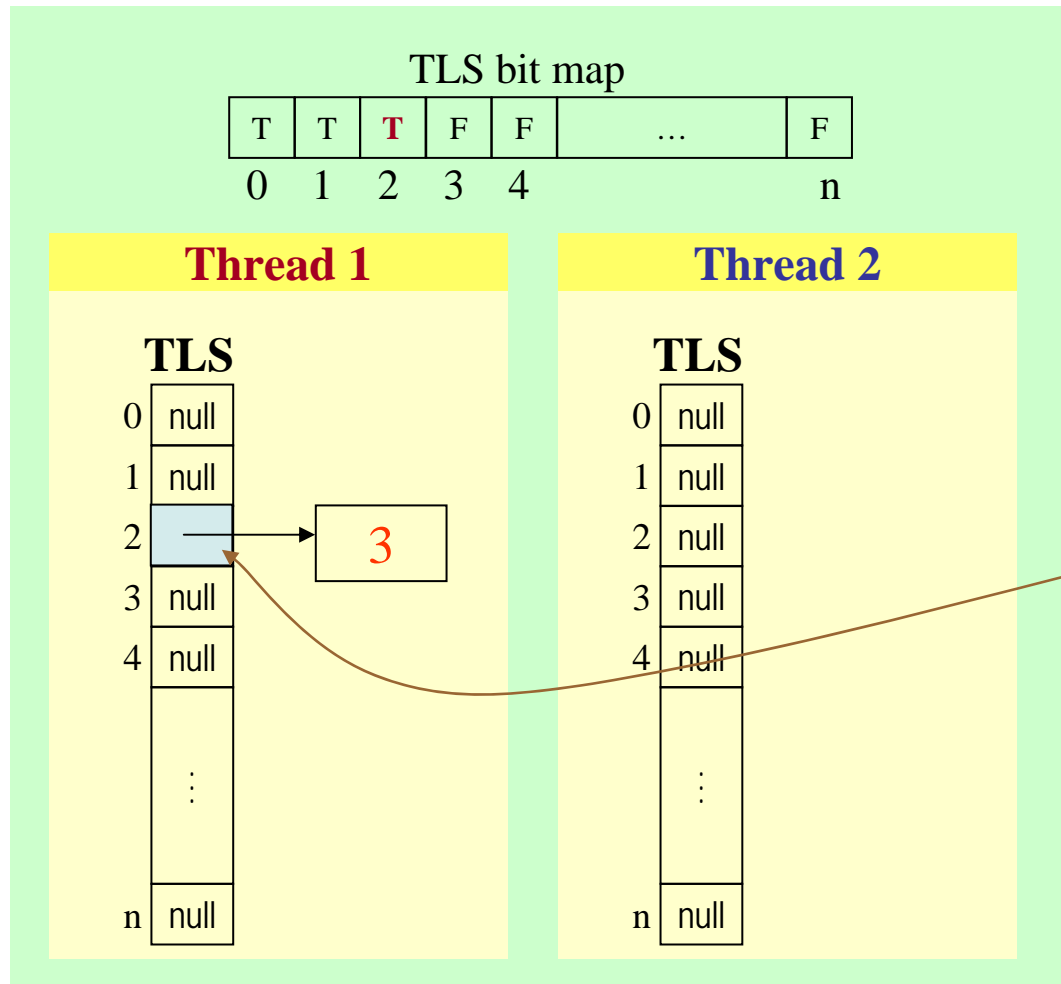
```
void main () {  
    tlsIdx = TlsAlloc();  
    ...  
}
```

```
thread1Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 3;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

```
thread2Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 5;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

TLS - Thread Local Storage

Processo



Exemplo de utilização

```
DWORD tlsIdx; tlsIdx = 2
```

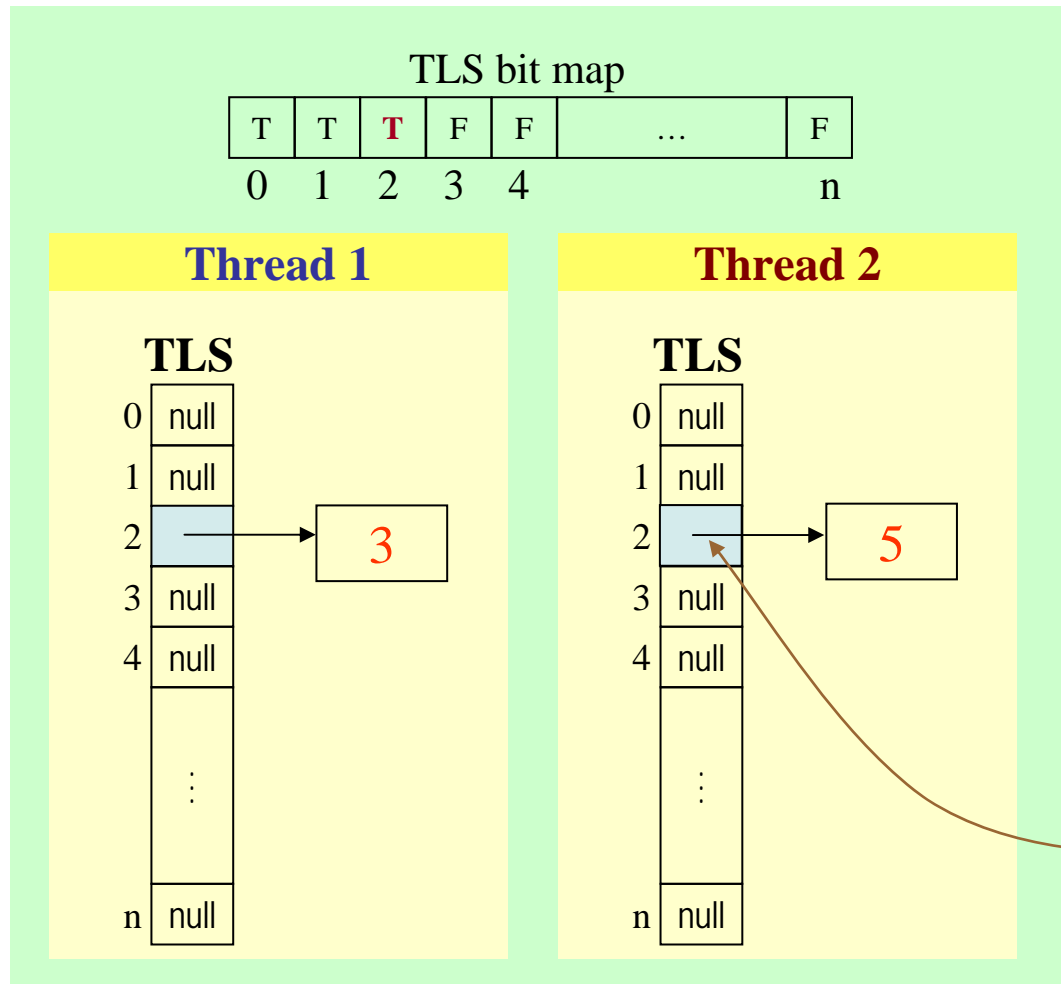
```
void main () {  
    tlsIdx = TlsAlloc();  
    ...  
}
```

```
thread1Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 3;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

```
thread2Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 5;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

TLS - Thread Local Storage

Processo



Exemplo de utilização

```
DWORD tlsIdx; ↗ tlsIdx = 2
```

```
void main () {  
    tlsIdx = TlsAlloc();  
    ...  
}
```

```
thread1Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 3;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

```
thread2Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 5;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

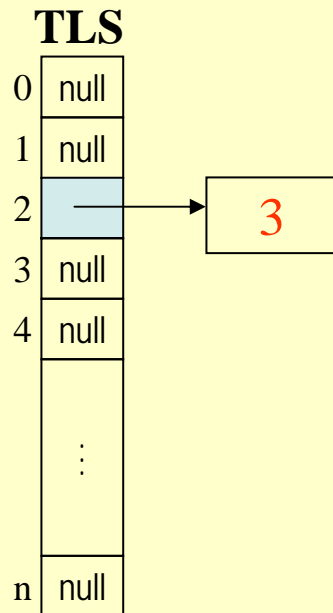
TLS - Thread Local Storage

Processo

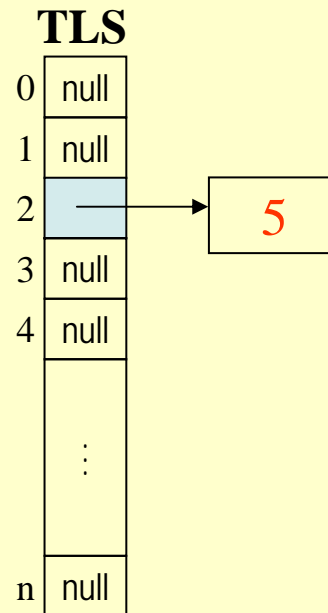
TLS bit map

T	T	T	F	F	...	F
0	1	2	3	4		n

Thread 1



Thread 2



Exemplo de utilização

DWORD tlsIdx; tlsIdx = 2

```
void main () {  
    tlsIdx = TlsAlloc();  
    ...  
}
```

```
thread1Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 3;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

```
thread2Proc() {  
    int *ptInt = new int[1];  
    *ptInt = 5;  
    TlsSetValue(tlsIdx, ptInt);  
    ...  
}
```

TLS – A função Rand62

Algoritmo em ambiente mono-thread

```
static int sReg = 1;

int MonoRand62(){
    sReg = (sReg >> 1) | (((sReg & 0x2)^((sReg & 1)<< 1))>0x20:0);
    return sReg;
}
```

A iniciação da variável *indxRnd* deve ser feita antes da criação dos *threads* que a utilizam, normalmente na função **main** ou **winmain**.

```
//Função de iniciação do módulo
void ModuleTlsInit(){
    if ((indxRnd = TlsAlloc()) == -1)
        FatalErrorSystem(" Falha no TlsAlloc!");
}
```

Algoritmo em ambiente Multi-thread

```
DWORD indxRnd=-1; // Índice da entrada TLS

int Rand62() {
    int* psReg = (int*)TlsGetValue(indxRnd); // Obter a entrada TLS

    // Se p==NULL então ainda não foi iniciada
    if( psReg==NULL) psReg = ThreadTlsInit ();

    // usar a entrada Tls
    *psReg = (*psReg >> 1) | (((*psReg & 0x2)^((*psReg & 1)<< 1))>0x20:0);
    return *psReg;
}
```

A entrada foi criada no *array* TLS e iniciada a NULL; cabe agora ao *thread* substituir o valor NULL por um apontador para os seus dados locais.

```
//Iniciação de uma entrada TLS para um thread
int *ThreadTlsInit () {
    // Espaço no heap para a informação local ao thread
    int *p = new int[1];
    *p = 1; // Valor inicial da variável
    if ( ! TlsSetValue(indxRnd, p) )
        FatalErrorSystem("TlsSetValue error!");
    return p;
}
```

TLS – A função Rand62

```
void main() {
    HANDLE thr[3];
    DWORD thld;

    // Criar uma entrada no TLS e guardá-la em indxRnd
    // Após a criação a entrada fica com o valor NULL para todos os threads.
    ModuleTlsInit();

    // Criar os threads. Falta teste de erro na criação dos threads
    thr[0]=chBEGINTHREADEX(NULL, 0, threadProc, "thread1.txt", 0, &thld);
    thr[1]=chBEGINTHREADEX(NULL, 0, threadProc, "thread2.txt", 0, &thld);
    thr[2]=chBEGINTHREADEX(NULL, 0, threadProc, "thread3.txt", 0, &thld);
    printf("TLS e thread criados. Esperar pelo termino dos threads.\n");

    //Esperar que os threads terminem
    WaitForMultipleObjects( 3,          // number of handles in array
                           thr,         // object-handle array
                           TRUE,        // wait option
                           INFINITE     // time-out interval
                           );

    //Destruir a entrada TLS
    ModuleTlsEnd();
    printf("Threads terminados e TLS libertado. \nVer o resultado nos ficheiros thread1.txt, ... \n");
}
```

```
//Função de iniciação do módulo
void ModuleTlsInit() {
    if ((indxRnd = TlsAlloc()) == -1)
        FatalErrorSystem(" Falha no TlsAlloc!");
}
```

```
//Função de Fecho do módulo
void ModuleTlsEnd() {
    if(indxRnd!=-1)
        if( !TlsFree(indxRnd) )
            FatalErrorSystem(" Falha no TlsFree!");
}
```


TLS – A função Rand62

Um exemplo de utilização da função Rand62 por um *thread* seria:

```
void threadProc(char* fname)
{
    FILE * fout = fopen(fname,"w"); //Ficheiro de output da sequência
    if (fout==NULL)
        FatalErrorSystem("Erro na criação do ficheiro!");
    int s;
    for(int i=0; i < 70; i++ ) {
        fprintf(fout, "i=%d ->%d\n", i, s=Rand62());
        Sleep(s);
    }

    //Fechar o ficheiro
    fclose(fout);

    //Libertar a memória do TLS
    ThreadTlsClose ();
}
```

//Término de uma entrada TLS

```
void ThreadTlsClose () {
    int* p = (int*)TlsGetValue(indxRnd); //Obter a entrada TLS
    if ( p ) {
        delete p;
        p=NULL;
        if ( !TlsSetValue(indxRnd, p) )
            FatalErrorSystem("TlsSetValue error!");
    }
}
```

TLS - *Thread Local Storage*

- Note-se que esta técnica é a adoptada pela Microsoft para resolver o problema da reentrância das funções da biblioteca standard do C/C++. Existem várias funções que têm necessidade de memorizar informação entre chamadas utilizando variáveis globais mas ao mesmo tempo *thread-safe*. Essas variáveis globais são armazenadas numa estrutura cujo endereço é armazenado numa entrada do TLS. Sendo assim, a criação de um *thread* necessita de criar e iniciar essa estrutura, razão por que se utiliza a macro **chBEGINTHREADEX** em vez da função **CreateThread** da API do Windows. Sugere-se que em modo *debug* faça o trace da execução do código correspondente à compilação da macro **chBEGINTHREADEX** para constatação do que se disse. Note ainda que esta macro também é responsável pela criação do *stack* de excepções para o código do *thread*.
- O recurso à memória TLS surge quando temos que construir módulos de código cujas funções são chamadas no contexto de *threads* de execução e não queremos que estas chamadas interfiram umas com as outras no caso de haver memória de estado global aos *threads*.
- Um caso onde este problema surge com muita frequência é na construção de bibliotecas DLL.

TLS – A DLL Random para ambiente multi-tarefa

```
BOOL WINAPIDllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved ) {  
    BOOL Result = TRUE;  
    switch (fdwReason) {  
        case DLL_PROCESS_ATTACH:  
            // Criar a entrada TLS  
            if ((indxRnd = TlsAlloc()) == -1) { Result = FALSE; break; }  
            //Atribuir espaço na entrada do thread corrente  
            Result = ThreadTlsInit() != NULL; break;  
  
        case DLL_THREAD_ATTACH:  
            //Atribuir espaço na entrada TLS do thread corrente  
            ThreadTlsInit(); break;  
  
        case DLL_THREAD_DETACH:  
            //Libertar espaço ocupado pelo thread na entrada TLS  
            ThreadTlsClose(); break;  
  
        case DLL_PROCESS_DETACH:  
            //Libertar espaço ocupado pelo thread na entrada TLS  
            ThreadTlsClose();  
            //Fechar a entrada de TLS no processo  
            if( !TlsFree(indxRnd) ) Result = FALSE;  
            break;  
    }  
    return Result; // Utilizado apenas para DLL_PROCESS_ATTACH  
}
```

Qual é a *thread* que executa este código?

TLS – A DLL Random para ambiente multi-tarefa

```
int WINAPI Rand62()
{
    int* psReg = (int*)TlsGetValue(indxRnd); // Obter a entrada TLS
    // Se p==NULL então ainda não foi iniciada
    if( psReg==NULL) {
        psReg = ThreadTlsInit();
        if ( psReg == NULL ) // Erro na iniciação do TLS
            return -1;
    }
    //usar a entrada Tls
    *psReg = (*psReg >> 1) | (((*psReg & 0x2)^(*psReg & 1)<< 1)?0x20:0);
    return *psReg;
}
```

Que *threads* podem ter a sua entrada de TLS a NULL considerando a definição da função `DllMain` apresentada?

```
//Iniciação de uma entrada TLS para um thread
int *ThreadTlsInit() {
    // Espaço no heap para a informação local ao thread
    int *p = new int[1];
    *p = 1; // Valor inicial da variável
    if (! TlsSetValue(indxRnd, p)) {
        delete [] p;
        return NULL;
    }
    return p;
}
```

```
//Término de uma entrada TLS
BOOL ThreadTlsClose() {
    int* p = (int*)TlsGetValue(indxRnd); //Obter a entrada TLS
    if ( p ) {
        delete p;
        p = NULL;
        if ( !TlsSetValue(indxRnd, p) ) return FALSE;
    }
    return TRUE;
}
```