# Threads in C++

**Livro Texto:** Multithreading Applications in Win 32 – Jim Beveridge, Robert Wiener

Threads são ideais para modelar o comportamento de diversos objetos do mundo real: os atores de um programa de simulação por exemplo. Se você está simulando o funcionamento de um banco, é interessante modelar cada caixa, cliente ou outros funcionários como objetos do tipo thread. Como se implementa isto ?

## Primeira tentativa:

```
class ThreadObject
{
public:
        void StartThread();
        virtual DWORD WINAPI ThreadFunc(LPVOID param);
private:
        HANDLE    m_hThread;
        DWORD     m_ThreadId;
};
```

A função ThreadFunc é virtual e assim um objeto real pode ser obtido derivando-se uma classe de ThreadObject e definindo-se a função ThreadFunc como desejado.

## Programa de exemplo:

```
/*
 * BadClass.cpp
 * Sample code for "Multitasking Applications in Win32"
 * This is from Chapter 9, Listing 9-2
 * Shows the wrong way to try and start a thread
 * based on a class member function.
*
* THIS FILE IS NOT SUPPOSED TO COMPILE SUCCESSFULLY
 * You should get an error on line 51.
 */
```

```
#include <windows.h>
#include <stdio.h>
#include <process.h>

typedef unsigned (WINAPI *PBEGINTHREADEX_THREADFUNC)(
   LPVOID lpThreadParameter
  );
typedef unsigned *PBEGINTHREADEX_THREADID;

class ThreadObject
{
public:
   ThreadObject();                  // Construtor
   void StartThread();
   virtual DWORD WINAPI ThreadFunc(LPVOID param);
   void WaitForExit();
private:
   HANDLE  m_hThread;               // Handle para thread criada
   DWORD m_ThreadId;                // Identificador da thread
};

ThreadObject::ThreadObject()       // Inicializa membros privados da classe
{
   m_hThread = NULL;
   m_ThreadId = 0;
}

void ThreadObject::StartThread()  // Cria Thread
{
   m_hThread = (HANDLE)_beginthreadex(NULL,
                        0,
                        (PBEGINTHREADEX_THREADFUNC)ThreadFunc,
                        0,
                        0,
                        (PBEGINTHREADEX_THREADID)&m_ThreadId );
   if (m_hThread) {
      printf("Thread launched\n");
   }
}


void ThreadObject::WaitForExit() // Espera fim da thread
{
   WaitForSingleObject(m_hThread, INFINITE);
   CloseHandle(m_hThread);
}
```

```
DWORD WINAPI ThreadObject::ThreadFunc(LPVOID param)
{
   // Do something useful ...

   return 0;
}

void main()
{
   ThreadObject obj;

   obj.StartThread();          // Cria thread
   obj.WaitForExit();          // Espera fim da thread
}
```

Ao tentar compilar o programa obtemos a seguinte mensagem de erro:

:\Ufmg2\BadClass\BadClass.cpp(51) : error C2440: 'type cast' : cannot convert from 'overloaded function type' to 'unsigned int (__stdcall *)(void *)'
None of the functions with this name in scope match the target type

**Motivo:**

- Toda função membro não estática tem um parâmetro oculto.
- Este parâmetro é usado toda vez que a função acessa um membro de dado ou quando o programador utiliza o ponteiro *this* dentro da função membro.
- A função ThreadObject::ThreadFunc(LPVOID param) tem efetivamente dois parâmetros: o ponteiro *this* e *param*.

Quando o WNT cria uma nova thread ele também cria um novo stack para a nova thread. e depois recria a chamada da função no novo stack, isto é, ele deve transferir os parâmetros da nova thread para o seu stack particular. *Param* é transferido corretamente, mas *this* não é copiado. O WNT não sabe que desta vez você está criando uma função thread, para uso num ambiente orientado a objeto, e que *this* deve ser empilhado como o primeiro parâmetro.

**Observações:**

- Em uma função membro não estática, a palavra chave *this* é um ponteiro para o objeto pelo qual a função é chamada.

- Dada uma classe A, uma chamada à função membro A::f, por exemplo,

  A *pa;
  pa-> f(2);

  poderia ser transformada em
  f(pa, 2);  // código gerado

- Assim, dentro de uma função membro, a palavra this aponta para o objeto sobre o qual a função membro foi invocada.

Bjarne Stroustrup, C++ Manual de referência comentado

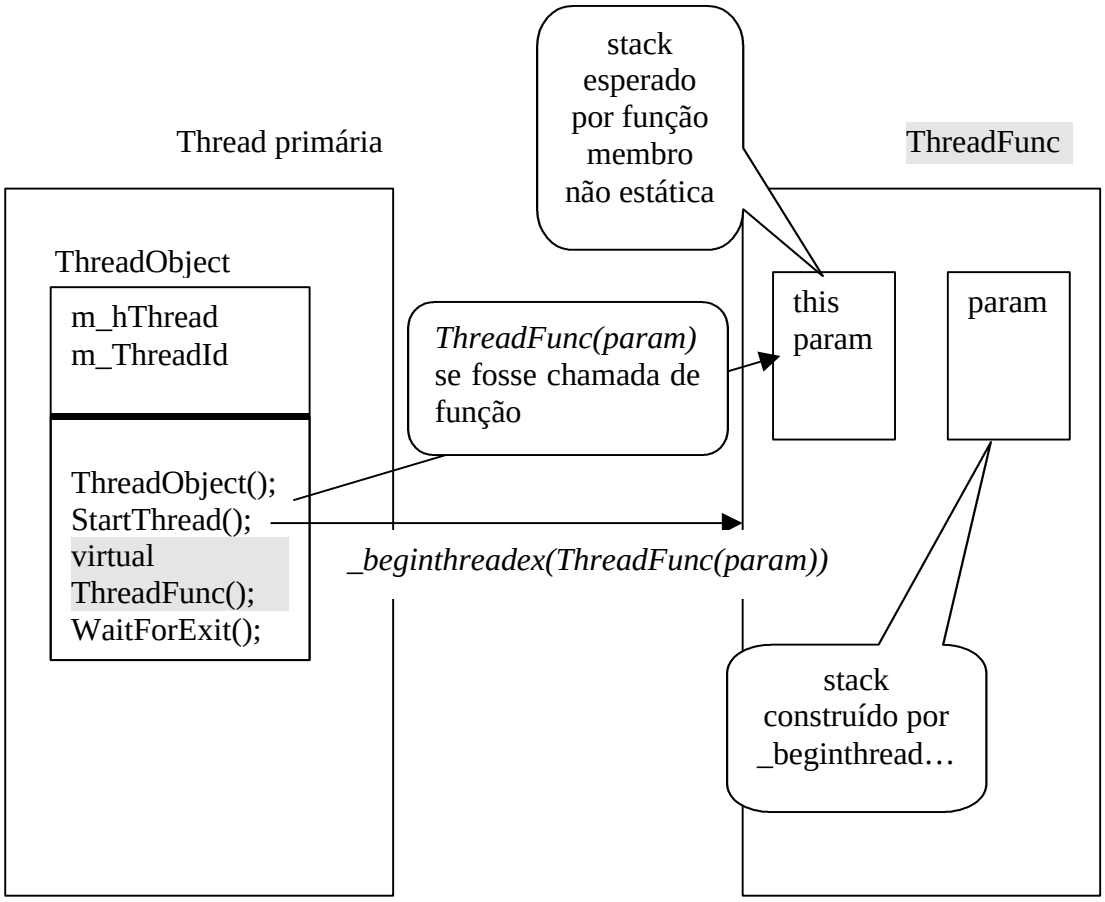
A convenção de chamada WINAPI (#define WINAPI __stdcall) implica:

- Os argumentos são passados da direita para a esquerda (convenção PASCAL).
- A função chamada retira os seus próprios argumentos do stack.
- O nome decorado é formado por nome da função + @ + número de bytes dos parâmetros. Exemplo:

  int func( int a, double b ) é decorado como: _func@12

Na convenção de chamada _cdecl, o stack deve ser limpo por quem chama a função.

Qual chamada de função resulta em maiores códigos para o programa principal ?




Solução para o problema:

Para iniciar um objeto thread de uma função membro:

1. Use uma função membro estática, que chamará a função membro desejada.
2. Use uma função no estilo C, que chamará a função membro desejada.

**Objetivo:**
Uma função auxiliar irá construir um *stack frame* correto para ser usado pela thread.

**Vantagens do método 1:**
A função membro estática tem acesso aos membros de dado privados e protegidos da classe.

**Função membro estática:**
Uma função membro estática não possui um ponteiro ***this***, assim ela só pode acessar membros não estáticos de sua classe pelo uso de . ou ->.
A finalidade do uso de membros *static* é reduzir a necessidade de variáveis globais, proporcionando alternativas que são locais a uma classe.
Uma função membro ou variável *static* atua como global para membros de usa classe, sem afetar o restante de um programa. Seu nome não entra em choque com os nomes de variáveis ou funções globais nem com os nomes de outras classes.

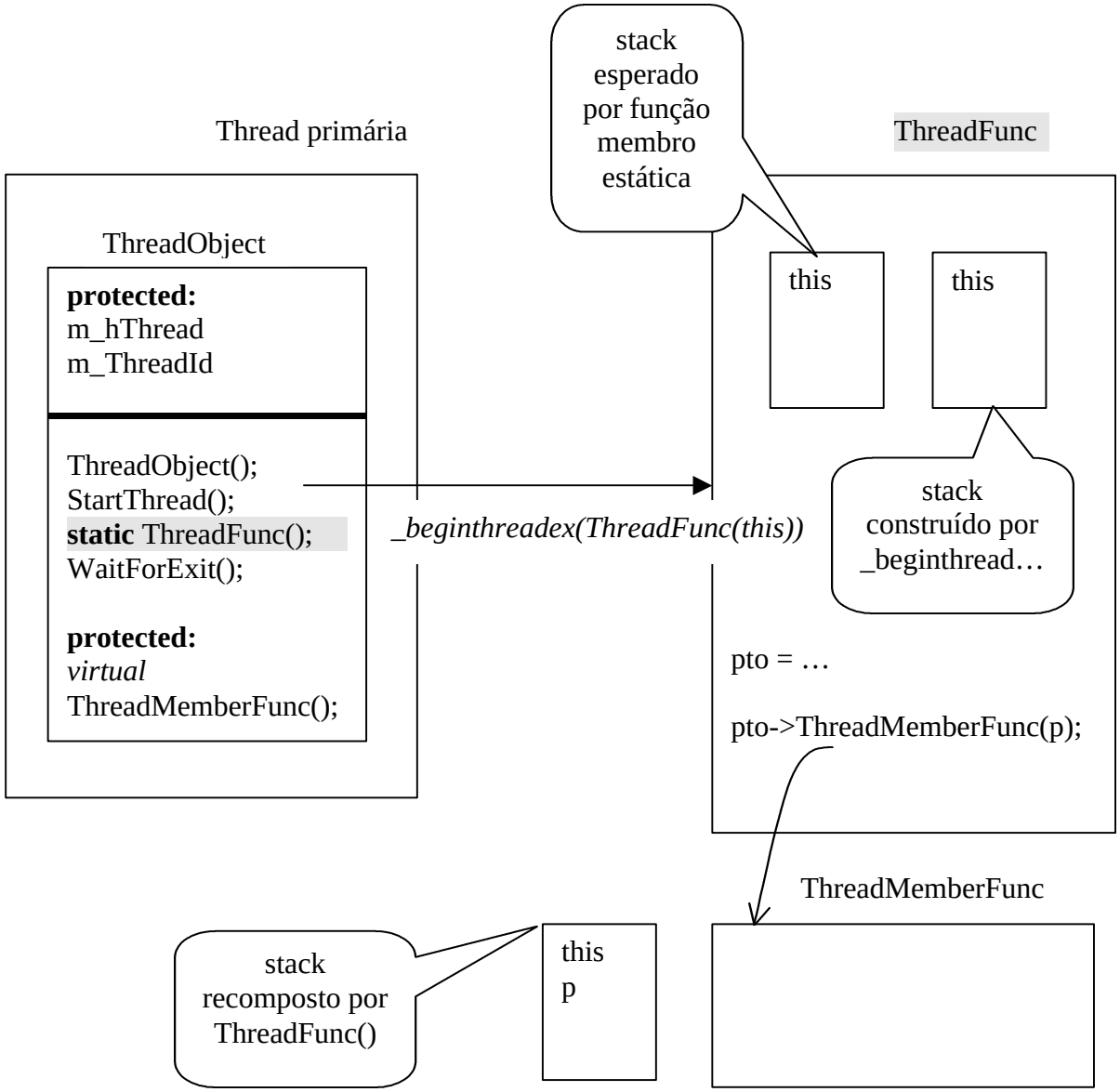
Bjarne Stroustrup, C++ Manual de referência comentado

## Exemplo: Programa corrigido
## Estratégia 1: Função membro estática

```
/*
 * Member.cpp
 * Sample code for "Multithreading Applications in Win32"
 * This is from Chapter 9, Listing 9-3 shows how to start a thread based on
* a class member function using a static member function.
*/

// Só os .h mais básicos serão incluídos.
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <process.h>

typedef unsigned (WINAPI *PBEGINTHREADEX_THREADFUNC)(
   LPVOID lpThreadParameter
   );
typedef unsigned *PBEGINTHREADEX_THREADID;

//
// This ThreadObject is created by a thread that wants to start another
// thread. All public member functions except ThreadFunc() are called
// by that original thread. The virtual function ThreadMemberFunc() is
// the start of the new thread.
//

class ThreadObject
{
public:
   ThreadObject();
   void StartThread();
   void WaitForExit();
   static DWORD WINAPI ThreadFunc(LPVOID param);

protected:
   virtual DWORD ThreadMemberFunc();
   HANDLE  m_hThread;
   DWORD   m_ThreadId;
};

ThreadObject::ThreadObject()       // Aqui nada muda
{
   m_hThread = NULL;
```

```
    m_ThreadId = 0;
}

void ThreadObject::StartThread()
{
    m_hThread = (HANDLE)_beginthreadex(NULL,
        0,
        (PBEGINTHREADEX_THREADFUNC) ThreadObject::ThreadFunc,
        (LPVOID)this, // passa pointer para objeto como parâmetro
        0,
        (PBEGINTHREADEX_THREADID) &m_ThreadId );

    if (m_hThread) {
        printf("Thread launched\n");
    }
}

void ThreadObject::WaitForExit()          // Aqui nada muda
{
    WaitForSingleObject(m_hThread, INFINITE);
    CloseHandle(m_hThread);
}

//
// This is a static member function.  Unlike C static functions, you only
// place the static declaration on the function declaration in the class, not on
// its implementation.
// Static member functions have no "this" pointer, but do have access rights.
//
DWORD WINAPI ThreadObject::ThreadFunc(LPVOID param)
{
    // Use the param as the address of the object
    ThreadObject* pto = (ThreadObject*)param;

    // Call the member function. Since we have a proper object pointer, even
    // virtual functions will be called properly.

    return pto->ThreadMemberFunc();
}

// This above function ThreadObject::ThreadFunc()  calls this function after
// the thread starts up.
DWORD ThreadObject::ThreadMemberFunc()
// Função que desempenhará as funções da thread
{
    // Do something useful ...
```

```
    return 0;
}

void main()
{
    ThreadObject obj;

    obj.StartThread();
    obj.WaitForExit();
}
```

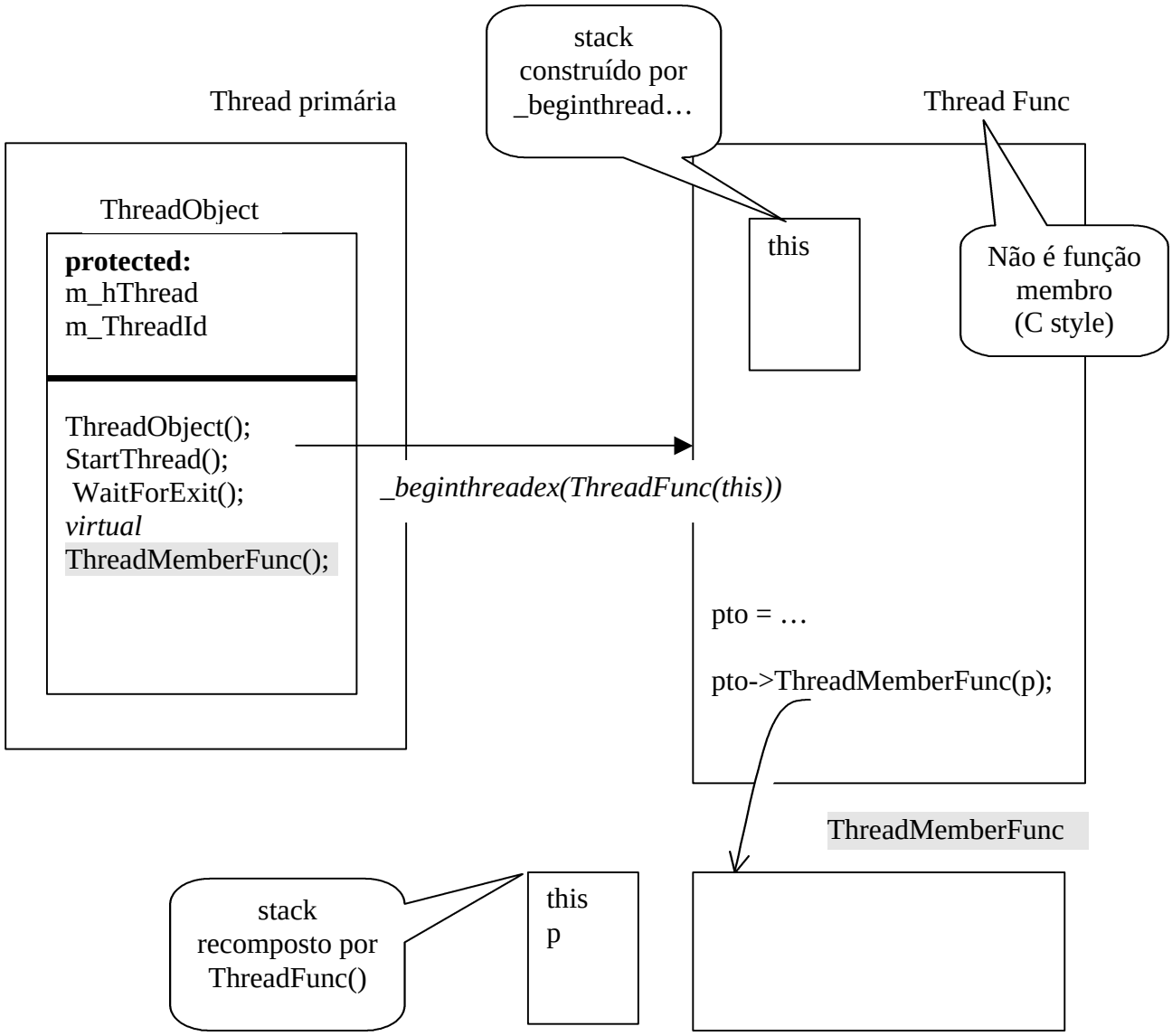


## Exemplo: Programa corrigido
## Estratégia 2: Função membro no estilo C

```
/*
 * Member2.cpp
 *
 * Sample code for "Multithreading Applications in Win32"
 * This is from Chapter 9, just after Listing 9-3.
 * Shows how to start a thread based on a class member function using
 * a C-style function.
*/

// Só os .h mais básicos serão incluídos.
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <process.h>

// Work around the buggy _beginthreadex() prototype
typedef unsigned (WINAPI *PBEGINTHREADEX_THREADFUNC)(
   LPVOID lpThreadParameter
   );
typedef unsigned *PBEGINTHREADEX_THREADID;

// Define the prototype for the function used to start the thread.
// Função no estilo C que chamrá a função membro
DWORD WINAPI ThreadFunc(LPVOID param);

class ThreadObject
{
public:
   ThreadObject();
   void StartThread();
   void WaitForExit();

   // A função membro deve ser declarada como pública ou a função no
   // estilo C não terá direitos de acessá-la.
      virtual DWORD ThreadMemberFunc();

protected:
   HANDLE  m_hThread;
   DWORD   m_ThreadId;
};

ThreadObject::ThreadObject()
```

```
{
   m_hThread = NULL;
   m_ThreadId = 0;
}

void ThreadObject::StartThread()
{
   m_hThread = (HANDLE)_beginthreadex(NULL,
      0,
      (PBEGINTHREADEX_THREADFUNC) ThreadFunc,
      (LPVOID)this,
      0,
      (PBEGINTHREADEX_THREADID) &m_ThreadId );

   if (m_hThread) {
      printf("Thread launched\n");
   }
}

void ThreadObject::WaitForExit()
{
   WaitForSingleObject(m_hThread, INFINITE);
   CloseHandle(m_hThread);
}

// Esta função é chamada quando a thread inicia
DWORD WINAPI ThreadFunc(LPVOID param)
{
   // Use the param as the address of the object
   ThreadObject* pto = (ThreadObject*)param;
   // Call the member function. Since we have a proper object pointer, even
   // virtual functions will be called properly.
   return pto->ThreadMemberFunc();
}

// A função ThreadFunc() chama esta função assim que a thread inicia
//
DWORD ThreadObject::ThreadMemberFunc()
{
   // Do something useful ...
   return 0;
}

void main()
{
   ThreadObject obj;
```

```
    obj.StartThread();
    obj.WaitForExit();
}
```



## Construindo Seções críticas mais seguras

Em C++ podemos construir classes contendo objetos de sincronização. As principais vantagens são:

- Os procedimentos para inicialização e encerramento ficam encapsulados e são automaticamente chamados pelo construtor e destrutor da classe, quando o objeto é criado e deletado. Isto diminui as chances de erro do usuário.

- Para proteger um certo tipo de dados, basta criar uma nova classe contendo o tipo básico anterior e um objeto de sincronização.

- A sintaxe para ativação de uma determinada diretiva é mais simples e direta.

### Exemplo: Encapsulando a diretiva critical section.

```
Class CriticalSection
{
public:
        CriticalSection();      // Construtor
        ~CriticalSection();     // Destrutor
        void Enter();           // Entra na seção crítica
        void Leave();           // Sai da seção crítica
private:
        CRITICAL_SECTION m_CritSect;
};

CriticalSection::CriticalSection() {
        InitializeCriticalSection(&m_CritSect);
}

CriticalSection::~CriticalSection() {
        DeleteCriticalSection(&m_CritSect);
}

CriticalSection::Enter() {
        EnterCriticalSection(&m_CritSect);
}

CriticalSection::Leave() {
        LeaveCriticalSection(&m_CritSect);
}
```

Vamos criar agora a classe string que protegerá todos os acessos a um objeto de sua classe, através de uma seção crítica.

```
Class String {
Public:
    String();
    virtual ~String();
    virtual void Set(char * str);
    int GetLength();
Private:
    CriticalSection   m_Sync;
    char*             m_pData;
};

String::String() {
    // O construtor de m_Sync será chamado automaticamente já que se trata
    // de uma variável membro..
    n_pData = NULL;
}

String::~String() {
    m_Sync.Enter();
    delete [] m_pData;
    m_Sync.Leave();
    // O destrutor do membro de dado m_Sync será chamado
    // automaticamente
}

void String::Set(char *str) {
    m_Sync.Enter();
    delete [] m_pdata;        // destrói string
    m_pdata = new char[::strlen(str)+1];  // aloca novo string
    ::strcpy(m_pData, str);               // copia string
    m_Sync.Leave();
}

int String::GetLength() {
    if (m_pData == NULL)
        return 0;
    m_Sync.Enter();
    int len = ::strlen(m_pData);
    m_Sync.Leave();
    return len;
}
```

A partir desta classe, todos os usuários estarão utilizando a classe string com exclusão mútua, garantindo um acesso seguro aos objetos da classe.

Vamos escrever uma nova função para esta classe. A função truncate retorna um substring limitado a um certo número de caracteres.

```
Void String::Truncate(int length) {
    if (m_pData == NULL)
        return 0;
    m_Sync.Enter();
    if length >= GetLength()) { // pediu para truncar além do fim do string
        m_Sysnc.Leave();
        return;
    }
    m_pData[length] = '\0';
    m_Sync.Leave();
}
```

A desvantagem deste procedimento é que temos que realizar o procedimento correto de saída em vários pontos do programa.

Vamos criar uma nova classe onde passamos como parâmetro na criação de um objeto, um outro objeto de sincronização. O construtor e destrutor desta nova classe que chamaremos de Lock efetuarão todo o trabalho de *housekeeping*.

```
Class Lock {
public:
    Lock(CriticalSection* pCritSect);
    ~Lock();
private:
    CriticalSection* m_pCritical;
};

Lock::Lock(CriticalSection* pCritSect) {
    m_pCritical = pCritSect;
    EnterCriticalSection(m_pCritical);
}

Lock::~Lock() {
    LeaveCriticalSection(m_pCritical);
}
```

A função truncate pode ser então rescrita:

```
Void String::Truncate(int length) {
    If (m_pData == NULL)
        return 0;
    // Ao declarar uma variável do tipo Lock o construtor será chamado
    // automaticamente.
    Lock lock(&m_Sync);
    if length >= GetLength()) { // pediu para truncar além do fim do string
        // lock efetuará limpeza automaticamente
        return; }
    m_pData[length] = '\0';
    // lock efetuará limpeza automaticamente
}
```

A vantagem desta nova classe é que agora fica impossível esquecer de efetuar a limpeza, após utilizar um objeto de sincronização.

A outra vantagem como veremos a seguir será que fica mais fácil escolher que objeto de sincronização iremos utilizar numa determinada aplicação.
Como já estudamos, cada diretiva de sincronização no WNT (*CriticalSection*, Mutex e Semáforo, possui uma capacidade diferente de resolver problemas.

As vantagens e desvantagens de cada diretiva podem ser visualizadas no quadro a seguir:

**Comparativo do poder de expressão das diretivas de sincronização do WNT:**

| | Critical Section | Mutex | Semáforo Binário |
|---|---|---|---|
| **Vantagens** | • Resolve todas as desvantagens do Algoritmo anterior.<br>• São diretivas de baixo overhead. | • Permite sincronizar threads no mesmo processo ou em processos diferentes.<br>• São objetos do kernel e portanto permitem uso das instruções Wait…<br>• Possibilita temporizar a espera para conquistar a seção crítica.<br>• Funções Wait… avisam quando thread sair da seção crítica sem sinalizar o Mutex. | • Possui todas as vantagens do Mutex.<br>• A thread que sinaliza o Mutex pode ser diferente da thread que realizou o Wait com sucesso. |
| **Desvantagens** | • Só funcionam para sincronizar threads dentro de um mesmo processo.<br>• Como não constituem objetos do kernel não podem ser usadas com instruções Wait…, logo não permitem temporizar timeout na espera.<br>• Se a thread morrer dentro da seção crítica ou sair sem evocar *LeaveCriticalSection()*, a seção crítica ficará fechada para sempre. | • Possuem overhead maior que Critical Sections.<br>• A thread que sinaliza o Mutex deve ser a mesma que realizou o Wait (proprietária do Mutex). | • Funções Wait… não retornam WAIT_ABANDONED quando uma thread apresenta problema dentro da seção crítica.<br>• Maior overhead. |

## Construindo Locks intercambiáveis:

Nós iremos construir uma Classe abstrata de dados denominada *LockableObject* que servirá de base para a construção de classes de dados concretas.

```
Class LockableObject {
Public:
    LockableObject() {}
    virtual ~LokableObject() {}
```

```
    virtual void Lock() = 0;
    virtual void Unlock() = 0;
}
```

Nós agora criaremos uma nova classe CriticalSection como derivada da classe LockableObject:

```
Class CriticalSectionV2: public LockableObject {
public:
        CriticalSectionV2();                // Construtor
        virtual ~CriticalSectionV2();       // Destrutor
        virtual void Lock();                // Entra na seção crítica
        virtual void Unlock();              // Sai da seção crítica
private:
        CRITICAL_SECTION m_CritSect;
};

CriticalSectionV2::CriticalSectionV2()
{
        InitializeCriticalSection(&m_CritSect);
}

CriticalSectionV2::~CriticalSectionV2()
{
        DeleteCriticalSection(&m_CritSect);
}

CriticalSectionV2::Lock()
{
        EnterCriticalSection(&m_CritSect);
}

CriticalSectionV2::Unlock()
{
        LeaveCriticalSection(&m_CritSect);
}
```

A classe Lock pode ser então rescrita, recebendo qualquer tipo de objeto de sincronização, ao invés de especificar que irá receber um objeto do tipo seção crítica. A função ficou muito mais genérica.

```
Class LockV2 {
public:
    Lock(LockableObject* pLokable);
```

```
    ~LockV2();
private:
    LokableObject* m_pLockable;         // Objeto do tipo locker qualquer
};

LockV2::LockV2(LockableObjetc* pLockable) {
    m_pLockablel = pLockable;
    m_pLockable->Lock();
}

Lock::~LockV2() {
    m_pLockable->Unlock();
}
```

Vamos rescrever a classe string baseada em nosso novo tipo de dados:

```
Class String V2{
Public:
    StringV2();
    virtual ~StringV2();
    virtual void Set(char * str);
    int GetLength();
Private:
    // Escolhemos objeto do tipo seção crítica como variável de locker
    CriticalSectionV2       m_Lockable; // assegura limpeza automática
    char*                   m_pData;
};

StringV2::StringV2() {
    // O construtor de m_Lockable será chamado automaticamente já que se
    // trata de uma variável membro..
    n_pData = NULL;
}

StringV2::~StringV2() {
    // O programa deve se assegurar que é seguro destruir o objeto
    delete [] m_pData;
    // O destrutor de m_Lockable será chamado automaticamente
}

void StringV2::Set(char *str) {
    LockV2 localLock(&m_Lockable);   // entra na seção crítica
    delete [] m_pdata;               // destrói string
    m_pdata = NULL;
    m_pdata = new char[::strlen(str)+1];  // aloca novo string
    ::strcpy(m_pData, str);          // copia string
```

```
    // Quando o objeto sai de escopo, seu destrutor é chamado e ele
    // abandona a seção crítica
}

int StringV2::GetLength() {
    LockV2 localLock(&m_Lockable);    // entra na seção crítica
    if (m_pData == NULL)
        return 0;
    return ::strlen(m_pData);                // sai da seção crítica
}
```

- Encapsulando os objetos de sincronização em classes, garantimos que ninguém irá acessar um dado inadvertidamente, sem exclusão mútua.

- Se ocorrer uma exceção o Win32 combinado com o C++ irá limpar o stack assegurando que todos os destrutores das variáveis alocadas sejam chamados. Com isso nenhuma seção crítica ficará trancada e nenhum objeto de sincronização deixará de ser fechado

## Exercícios:

1. Criar a classe Mutex derivada da classe *LockableObject*.

2. Criar a classe Semaphore derivada da classe *LockableObject*.

3. Codificar a classe ListaEncadeada (*class list*, dada em sala de aula), com exclusão mútua em todo acesso à estrutura de dados. Todas as operações de inserção e remoção de nodos na lista devem estar protegidas.

4. Escrever uma versão orientada a objeto do exemplo 2.3 do livro texto. A thread deve ser definida como uma função membro de uma classe.