

Automação em Tempo Real

Módulo 8: Uso de Threads em ambientes OO

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

I invented the term Object-Oriented, and I can tell you I did not have C++ in mind
- Alan Kay

Professor: Constantino Seixas Filho

sábado, 7 de maio de 2005

1

Thread como função membro de um objeto

- Threads podem ser utilizadas como funções membro de um objeto
- Estes objetos são úteis para modelar os *sprites* de um game, os personagens/equipamentos de uma simulação, etc
- Entretanto o uso de uma thread em C++ requer certos cuidados e não é tão simples como parece

Objeto da
Classe Aluno



Objeto da
Classe Professor

2

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.

3

Primeira tentativa



```
#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;
}
```

4

Exemplo - Sistema supervisorio



```
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)
{ // Faça algo útil
    return 0;
}

void main()
{
    ThreadObject obj;
    obj.StartThread(); // Cria thread
    obj.WaitForExit(); // Espera fim da thread
}
```

5

Ao compilar



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

Onde está o erro ?

- 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

6

Observações:

- Em uma função membro não estática, a palavra chave *this* é um ponteiro para o objeto através do 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

7

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.

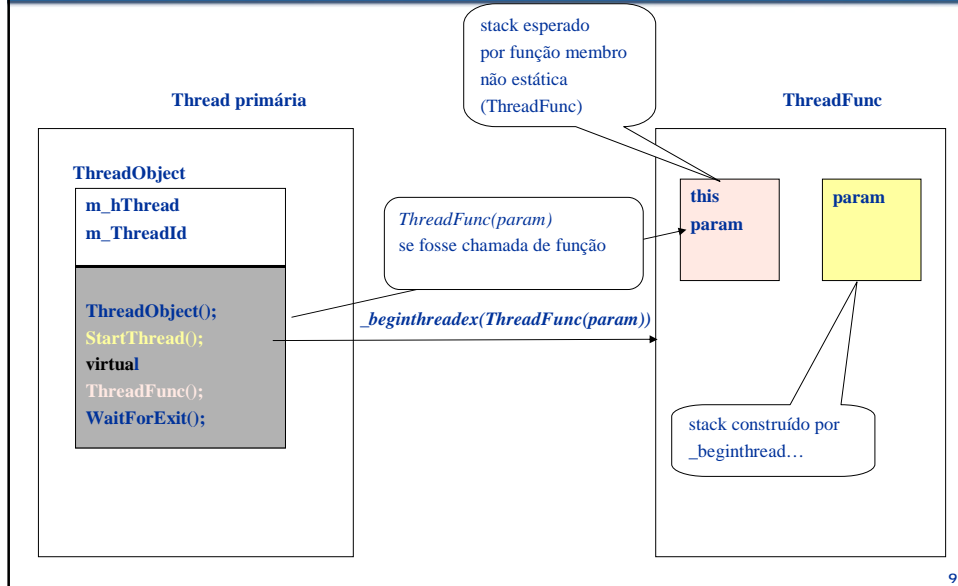
Exemplo:

```
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 convenção de chamada de função resulta em maior código para o programa principal ?

8



Solução para o problema:

Para iniciar um objeto thread de uma função membro:

- Use uma função membro estática, que chamará a função membro desejada.
- 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.

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 sua classe, sem afetar o restante do 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

11

Estratégia 1: Função membro estática



```
// 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;
};
```

12

Estratégia 1: Função membro estática



```
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);
}
```

13

Estratégia 1: Função membro estática



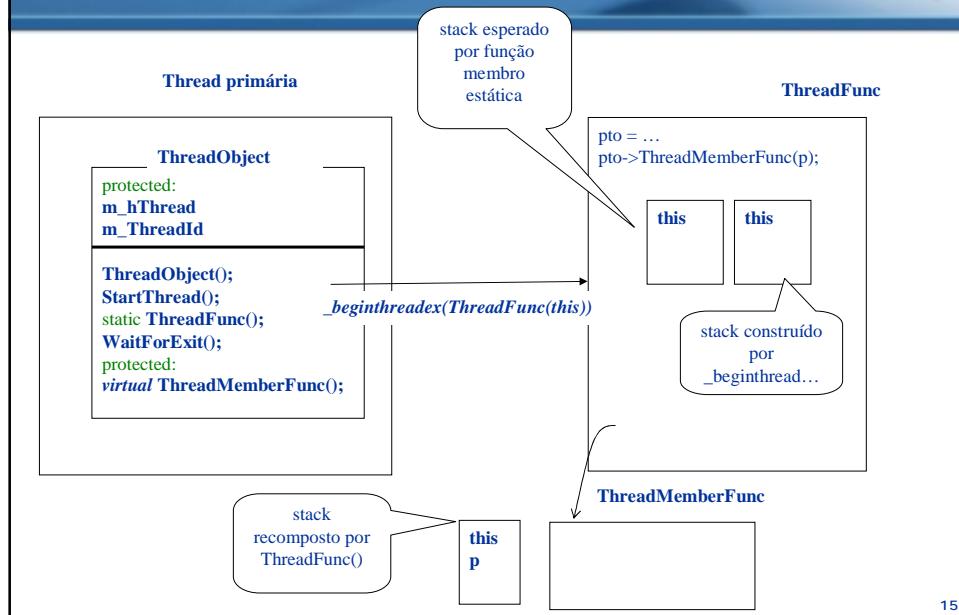
```
//
// This is a static member function. Unlike C static functions, you only
// place the static declaration on the function declaration in the class, not on // its implementation.
// Static member functions have no "this" pointer, but do have access rights.
//
DWORD WINAPI ThreadObject::ThreadFunc(LPVOID 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();
}
```

14

Stack Frame



15

Estratégia 2: Função membro no estilo C



```
// 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;

// Define the prototype for the function used to start the thread.
// Função no estilo C que chamrará 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 direito de acessá-la.
    virtual DWORD ThreadMemberFunc();
protected:
    HANDLE m_hThread;
    DWORD m_ThreadId;
};
```

16

Estratégia 2: Função membro no estilo C



```
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);
}
```

17

Estratégia 2: Função membro no estilo C



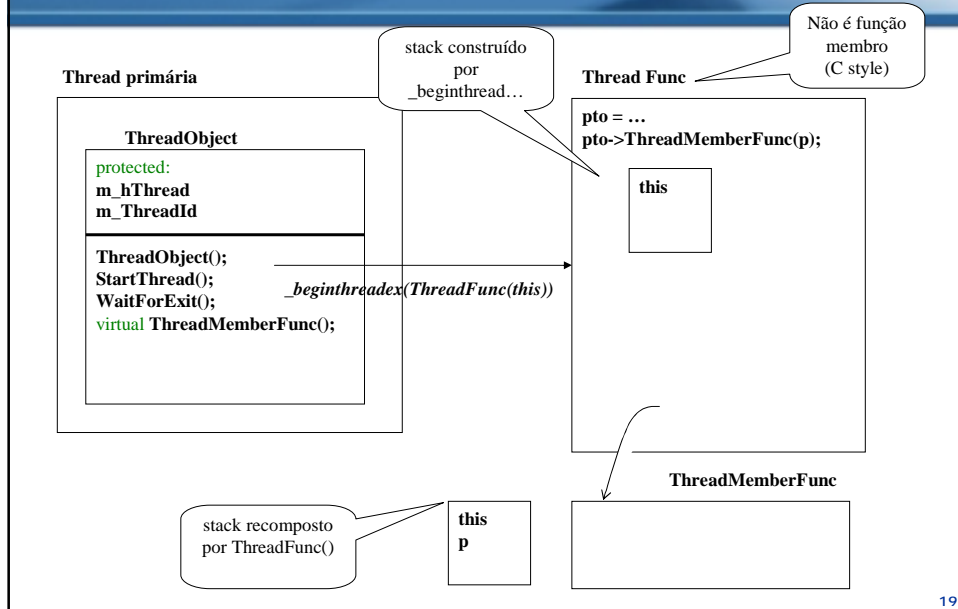
```
// Esta função é chamada quando a thread inicia
DWORD WINAPI ThreadFunc(LPVOID param)
{
    // Use the param as the address of the object
    ThreadObject* pto = (ThreadObject*)param;
    // Call the member function. Since we have a proper object pointer, even
    // virtual functions will be called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

18

Estratégia 2: Função membro no estilo C



19

Enacapsulando 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);
}
```

20

Classe String com Exclusão Mútua



```
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..
    m_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[sizeof(str)+1]; // aloca novo string
    strcpy(m_pData, str); // copia string
    m_Sync.Leave();
}
```

21

Classe String com Exclusão Mútua



```
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_Sync.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.
```

22

Classe Lock



```

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 reescrita:

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
}
    
```

23

Comparação das diretivas de controle de Exclusão Mútua



	Critical Section	Mutex	Semáforo Binário
V a n t a g e n s	<ul style="list-style-type: none"> •Em relação aos algoritmos de Decker: Não tem espera ocupada •Permite número de threads maior que dois •São diretivas de baixo overhead. 	<ul style="list-style-type: none"> •Permite sincronizar threads no mesmo processo ou em processos diferentes. •São objetos do kernel e portanto permitem uso das instruções Wait... •Possibilita temporizar a espera para conquistar a seção crítica. •Funções Wait... avisam quando thread sair da seção crítica sem sinalizar o Mutex. 	<ul style="list-style-type: none"> •Possui todas as vantagens do Mutex. •A thread que sinaliza o Mutex pode ser diferente da thread que realizou o Wait com sucesso.
D e s v a n t g e n s	<ul style="list-style-type: none"> •Só funcionam para sincronizar threads dentro de um mesmo processo. •Como não constituem objetos do kernel não podem ser usadas com instruções Wait..., logo não permitem temporizar timeout na espera. •Se a thread morrer dentro da seção crítica ou sair sem evocar <i>LeaveCriticalSection()</i>, a seção crítica ficará fechada para sempre. 	<ul style="list-style-type: none"> •Possuem overhead maior que Critical Sections. •A thread que sinaliza o Mutex deve ser a mesma que realizou o Wait (proprietária do Mutex). 	<ul style="list-style-type: none"> •Funções Wait... não retornam WAIT_ABANDONED quando uma thread apresenta problema dentro da seção crítica. •Maior overhead.

24

```
class LockableObject {
public:
    LockableObject() {}
    virtual ~LockableObject() {}
    virtual void Lock() = 0;
    virtual void Unlock() = 0;
}
```

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

25

```
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);
}
```

26

Class LockV2



```
class LockV2 {
public:
    Lock(LockableObject* pLockable);
    ~LockV2();
private:
    LockableObject* m_pLockable; // Objeto do tipo locker qualquer
};

LockV2::LockV2(LockableObject* pLockable) {
    m_pLockable = 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;
};
```

27

Class LockV2



```
StringV2::StringV2() {
    // O construtor de m_Lockable será chamado automaticamente já que se trata de uma variável membro..
    m_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[sizeof(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 sizeof(m_pData); // sai da seção crítica
}
```

28

Muito Obrigado

UFMG

Perguntas?

Constantino Seixas Filho

constantino.seixas@task.com.br

