**Desenvolvendo uma Biblioteca de tempo de execução C para MSVC – C Runtime (CRT)**

Como você sabe, não podemos usar o tempo de execução fornecido com o Windows. A razão é bastante simples. O tempo de execução do Windows C ++ depende muito de um sistema operacional Windows. Como estamos desenvolvendo um novo sistema operacional, esse tempo de execução não existe.

Devido a isso, devemos criar nosso próprio código de tempo de execução C ++. Isso pode ficar complicado. Muitos recursos C ++ requer o uso de um tempo de execução. No entanto, como desativamos o tempo de execução, o compilador gerará erros interessantes ao usar esses recursos. Outras vezes são simplesmente imprevisíveis e podem causar uma falha tripla.

Em aplicativos, o que chama main ()? A biblioteca de tempo de execução. O que chama e inicia todos os objetos globais? A biblioteca de tempo de execução. O que fornece certos apoios de palavras-chave que vinculam o sistema (como novo e excluir)? A biblioteca de tempo de execução. O que configura as informações da pilha initil? Novamente: **a biblioteca de tempo de execução.**

Não definir uma biblioteca de tempo de execução pode causar resultados imprevisíveis. Por um lado, objetos globais e estáticos nunca serão inicializados. Outro problema é que o uso de determinadas palavras-chave é imprevisível. Objetos globais e estáticos nunca serão desalinhados. Além disso, o compilador depende de determinadas rotinas - usualmente definido pelo tempo de execução padrão. Definir e chamar funções virtuais pode tornar-se imprevisível. A chamada de rotinas virtuais puras será imediatamente travada. E, diga bem com o **new, delete, type e exceptions** .

Criar um pequeno tempo de execução C ++ é essencial para que C ++ possa funcionar corretamente.

**Operadores Globais**

Você precisará definir os operadores globais New e Delete. O problema, no entanto, é que não temos nenhum gerenciador de memória para trabalhar nisso. Por isso, por enquanto, não faça nada:

void\* \_\_cdecl ::operator new (unsigned int size) { return 0; }

void\* \_\_cdecl operator new[] (unsigned int size) { return 0; }

void \_\_cdecl ::operator delete (void \* p) {}

void \_\_cdecl operator delete[] (void \* p) {}

Agora, podemos usar o **New** e **Delete** sem erros - embora eles realmente não façam nada ... ainda assim, de qualquer maneira os criamos.

**Manipulador de chamadas de função virtual**

As funções virtuais são funções declaradas na classe, mas não contêm nenhuma implementação. O objetivo principal é **forçar as** classes derivadas a sobrecarregar essa função.

Não é possível chamar uma função virtual diretamente através de meios normais. Chamar uma função virtual resultará em comportamento indefinido, porque essa função na verdade não existe - nunca foi definida.

Se uma função virtual receber alguma chamada, o compilador tenta usar **\_purecall ()** como o manipulador de chamadas. Se isso não existir, o resultado é imprevisível. Ou seja, uma falha tripla.

Por isso, nosso tempo de execução C ++ precisará defini-lo:

int \_\_cdecl ::\_purecall() { for (;;); return 0; }

**Suporte de ponto flutuante**

Tudo está funcionando de forma excelente com nosso novo kernel MSVC ++. Ou seja, até tentar compilar **float i = 2/2;**e BAM! Somos atingidos por erros. Mais especificamente, erros externos não resolvidos .

Não há nada de errado com isso ... Assim como usar os operadores New e Delete, sem que eles sejam definidos. Simularmente, o MSVC ++ precisa de algumas rotinas definidas para trabalhar com matemática de ponto flutuante.

**\_fltused**

Isso é usado pelo MSVC ++ para determinar se o ponto flutuante está atualmente em uso. Isso deve ser definido como 1 e deve ser dada ligação C se construir para C ++:

extern "C" int \_fltused = 1;

**\_ftol2\_sse ()**

Dependendo do seu nível de otimização, o MSVC ++ pode incorporar chamadas para \_ftol2\_sse () para converter um flutuador em um longo. Eu não usarei a SSE aqui, então vou escrever minha implantação usando a FPU:

extern "C" long \_\_declspec (naked) \_ftol2\_sse()

{

int a;

#ifdef ARCH\_X86

\_asm fistp [a]

\_asm mov ebx, a

\_asm ret

#endif

}

**Outras rotinas**

Eu defini outras rotinas, que são \_CIcos (), \_CIsqrt () e \_CIsin (). Até que possamos verificar que essas rotinas são necessárias, vou mantê-las dentro da nossa biblioteca de tempo de execução.

**Inicializando globais e dados estáticos**

Tudo vai bem até agora, exceto o que diz respeito aos globais? Lembre-se de que o tempo de execução é responsável por executar todas as rotinas globais e inicializar todos os objetos globais e estáticos? Como desativamos o tempo de execução, temos que fazê-lo.

Para fazer isso, temos que entender exatamente como o MSVC ++ lida com os construtores (ctors).

MSVC ++ usa uma seção especial (semelhante para .data, .bss, .text, etc.) dentro da imagem binária final para os construtores. Quando o compilador MSVC ++ encontra um objeto que deve ser executado pelo código de inicialização, ele coloca um **inicializador dinâmico** dentro desta seção. Isso significa que - para cada inicializador dinâmico que precisa ser executado na inicialização, todos podem ser encontrados localizado dentro desta seção especial.

Esta seção é a seção **.CRT** . **Esses inicializadores dinâmicos são uma matriz de ponteiros de função de 4 bytes, que são armazenados em .CRT.** Assim, se conseguimos encontrar uma maneira de analisar esta seção, podemos chamar cada ponteiro de função que o MSVC ++ estabeleceu para nós, e assim chamar cada rotina que precisa ser chamada na inicialização.

Não podemos fazer isso com C ++ sozinho, no entanto, como estes nomes de seção são muito específicos para MSVC ++. Além disso, como estamos construindo o noso tempo de execução, a seção .CRT é atualmente inexistente. Nós temos que adicionar esta seção nós mesmos. A **única** maneira de fazer isso é usar o pré-processador.

**Convenções de nomeação**

Tudo bem ... Esta seção pode ser um pouco confusa. Os nomes das seções usadas no MSVC ++ são muito estranhos. Seriamente-- *.CRT $ XCU* ? No que eles estavam pensando?

Na verdade, esses nomes de seção têm um propósito. os nomes das seções são compostos por duas partes, separadas pelo sinal de dólar "$". **A primeira parte é usada como o nome da seção base** . A segunda parte indica onde está localizado na imagem final.

Ou seja, podemos pensar em um nome de seção com este formato:

.section\_name$location\_name

O **nome da seção** pode ser .code, .data, .bss, .CRT ou qualquer outro nome da seção. O nome da **localização** é um nome que representa onde estamos na seção. Para um exemplo, em **.CRT $ XCA** , .CRT é o nome da seção, XCA é a sua localização nessa seção. Não importa o nome desse local; O importante é que ele está em **ordem alfabética** .

Aqui está um exemplo:

.CRT$XCA

.CRT$XCU

.CRT$XCZ

Observe que as seções estão dispostas nas letras do alfabeto. O arranjo da segunda parte determina sua localização, na qual eles são armazenados dentro da imagem final. No exemplo acima, **.CRT $ XCA será o primeiro, .CRT $ XCU é o segundo, .CRT $ XCZ é o último.**

Vamos ver de outra maneira, misturando eles ...

.CRT$XCZ

.CRT$XCA

.CRT$XCU

A mesma coisa se aplica aqui. .CRT $ XCA é a primeira seção, novamente. Este exemplo ilustra isso: **a ordem dessas seções depende do seu alfavalue – então: a vem antes de z, então .CRT $ XCA vem antes .CRT $ XCZ.**Observe o último caractere.

Poderemos ver essas seções dentro do mapa do vinculador, quando as configurarmos.

**Criando novos nomes de segmentos**

Para criar uma nova seção, precisamos usar a diretiva **#pragma data\_seg ()** . Esta diretiva garante que todos os dados alocados após sua colocação dentro desta nova seção.

Esta diretiva assume a forma:

#pragma data\_seg( ["section-name"[, "section-class"] ] )

"section\_class" é retido apenas para fins de compatibilidade, e agora é ignorado pelo MSVC ++.

A parte importante, portanto, é o primeiro parâmetro - "nome da seção", que dá o novo nome da seção

#pragma data\_seg (".CRT$XCA")

// Todas as varices alocadas aqui são agora colocadas dentro da seção .CRT$XCA, em vez da seção .data

Para voltar à seção padrão (.data), use este pragma sem parâmetros:

//! Selecione novamente o segmento de dados padrão (.data) para o resto da unidade

#pragma data\_seg ()

**Fusão de Seções**

Por padrão, não podemos ler ou escrever na seção .CRT. No entanto, podemos ler e usar a seção .data. O que queremos é as mesmas **permissões** para os **dois** nomes de seção.

Podemos corrigir isso combinando as duas seções em conjunto, o que assegurará que ambas as seções possam ser lidas e esctritas:

//! Agora, mova os dados CRT para a seção .data para que possamos ler / escrever neles

#Pragma Comment (linker, "/merge:.CRT=.data")

Ok ... Então vamos ver ... Todas as rotinas de inicialização globais são armazenadas como ponteiros de função dentro da seção .CRT$XCU da imagem binária. Podemos declarar uma seção antes e depois desta seção, e assegurar-se de que estão logo após o outro, graças à convenção de nomeação e ao vinculador. Como eles estão bem um ao lado do outro, podemos declarar um variável para apontar para essas seções - efetivamente apontando para o primeiro e último ponteiro de função na matriz de inicialização. Vamos ver isso em seguida ...

**Inicializando Configuração Global**

Vamos ver o código real, e dividi-lo:

//! Ponteiro de função typedef

typedef void (\_\_cdecl \* \_PVFV) (void);

Primeiro, digitamos um ponteiro de função para melhorar a legibilidade. Este ponteiro de função é usado para apontar para cada inicializador global.

/ \*\*

\* O MSVC ++ cria inicializadores dinâmicos e desalinhadores, o que nos ajuda a chamar as rotinas.

\* O compilador e o vinculador vinculam todos os inicializadores dinâmicos em uma tabela de ponteiro de função dentro de uma

\* seção chamada .CRT$XCU.

\* /

// Tempo de execução C ++ padrão (STD CRT) \_\_xc\_a aponta para o início da tabela de inicialização

#pragma data\_seg (".CRT$XCA")

\_PVFV \_\_xc\_a [] = {NULL};

O código acima cria a seção .CRT$XCA. Ao declarar isso com um "A", nós asseguramos que isso será correto antes da próxima seção .CRT ser definida, então garantimos que o inicio da nossa seção acontece antes de .CRT$XCU.

\_\_xc\_a é um nome de CRT MSVC ++ padrão usado como ponteiro para o início da lista de inicialização, armazenado em .CRT$XCU

//! .CRT$XCU está localizado aqui.

Nosso .CRT$XCU está localizado antes de .CRT$XCZ e depois de .CRT$XCA por causa da convenção de nomenclatura usada.

//! Tempo de execução C ++ padrão (STD CRT) \_\_xc\_z aponta para o final da tabela de inicialização

#pragma data\_seg (".CRT$XCZ")

\_PVFV \_\_xc\_z [] = {NULL};

Esta é a seção .CRT$XCZ. Novamente, devido às convenções de nomenclatura, é garantido estar logo após a lista do iniciador dentro de .CRT$XCU. Ao definir um ponteiro de função aqui, **é garantido apontar para a última rotina de inicialização - 1, dentro da matriz de inicialização dentro de .CRT$XCU.**

\_\_xc\_z é o nome padrão utilizado pelo MSVC ++ CRT.

//! Selecione novamente o segmento de dados padrão (.data) para o resto da unidade

#pragma data\_seg ()

Para todos os outros dados, queremos usar a seção .data, então volte a essa seção ...

//! Agora, mova os dados CRT para a seção .data para que possamos ler / escrever

#Pragma Comment (linker, "/merge:.CRT=.data")

Mesclar a seção .CRT com nossa seção .data. Isso garante que podemos acessar a seção .CRT da seção .data. Para inicializar cada rotina, basta percorrer cada ponteiro da função e chamá-la. **Aviso: Cuidado com os ponteiros de função nula. Chamar um ponteiro de função nula resultará em um salto inválido para alguma localização aleatória na memória, o que resultará em uma falha tripla.**

void \_\_cdecl \_initterm (\_PVFV \* pfbegin, \_PVFV \* pfend)

{

// Percorrer cada inicializador

while (pfbegin <pfend)

{

// Execute o inicializador global

if (\* pfbegin! = NULL)

(\*\* pfbegin) ();

// Vá para o próximo inicializador dentro da tabela do inicializador

++ pfbegin;

}

}

// Isso inicializa todas as rotinas do inicializador global:

\_initterm (\_\_ xc\_a, \_\_xc\_z);

**Limpando o ambiente**

Agora, todas as rotinas de inicialização globais podem ser executadas. Qual é o próximo passo? Limpar tudo após a posição atual.

A boa notícia é que é muito mais fácil trabalhar com as rotinas do inicializador. Tudo o que precisamos para definir uma localização para armazenar uma série de ponteiros da função de desinstalador global em algum local na memória:

//! tabela ponteiro de função para tabela de desinstalador global

static \_PVFV \* pf\_atexitlist = 0;

//! Máximo de entradas permitidas na tabela

static unsigned max\_atexitlist\_entries = 32;

//! Quantidade atual de entradas na tabela

static unsigned cur\_atexitlist\_entries = 0

Estes são os nossos ponteiros de função que usamos para acompanhar o local onde estamos no conjunto de desinstaladores globais.

**Nós** definimos onde essas matrizes estão localizados. MSVC ++ adiciona o código de destrutor para cada objeto global, que adiciona um ponteiro de função à matriz de desinstalador global. Chamando uma rotina especialmente definida, **atexit ()** .

**Nota: MSVC ++ requer essa rotina. Não definir esta rotina resultará em erros ao definir um destrutor (dtor) de qualquer tipo.**

A rotina real é simples. Lembre-se de que, para cada objeto global, MSVC ++ incorpora código que irá chamar essa rotina. O dtor como um objeto é passado para esta rotina como um parâmetro. Por isso, tudo o que precisamos fazer é adicioná-lo ao final da nossa matriz dtor:

//! Para cada objeto global criado, MSVC ++ chama essa rotina com uma função ptr para cada dtor

int \_\_cdecl atexit (\_PVFV fn)

{

//! Certifique-se de que temos espaço livre suficiente

if (cur\_atexitlist\_entries >= max\_atexitlist\_entries)

return 1;

else {

// Adicionar a rotina de saída

\* (pf\_atexitlist ++) = fn;

cur\_atexitlist\_entries ++;

}

return 0;

}

Então ... Agora que temos uma maneira de adicionar dtors à lista (Lembre-se de que MSVC ++ faz isso automaticamente através de nossa função), Tudo o que precisamos fazer é inicializar a matriz de ponteiro de função original:

void \_\_cdecl \_atexit\_init (void)

{

max\_atexitlist\_entries = 32;

// Aviso: Normalmente, o STDC irá alocar dinamicamente isso. Porque não temos nenhum gerenciador de memória, basta escolher

// um endereço de base que você nunca usará agora

pf\_atexitlist = (\_PVFV \*) 0x500000;

}

Não muito difícil, espero :)

Há muitos conceitos possivelmente novos para nossos leitores, no entanto, então tudo isso pode ficar mais claro com um exemplo prático.

**O Ponto de Entrada**

Ok, então vamos ver ... Vimos que o endereço de entrada da imagem PE ou a rotina do ponto de entrada é executada imediatamente pelo carregador do 2º estágio. Nós configuramos nosso ponto de entrada para ser **kernel\_entry,** então vamos defini-lo:

Precisamos garantir que os registradores e a pilha sejam configurados antes de qualquer código ser executado. Isso é muito importante para garantir que reflitemos os descritores corretos no GDT dos bootloaders. Também precisamos configurar a pilha, pois o C ++ usa a pilha de forma adequada.

void \_cdecl kernel\_entry ()

{

#ifdef ARCH\_X86

\_asm {

cli // limpa as interrupções - não as habilite ainda

ax, 10h // deslocamento 0x10 no gdt para o seletor de dados, lembre-se?

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

mov ss, ax // Configuração da pilha de bases

mov esp, 0x90000 // Em seguida, armazenamos o ponteiro atual na pilha. Isso assegurará que

mov ebp, esp // qualquer rotina que chamarmos, tenha um lugar para retornar

push ebp

}

#endif

//! Execute construtores globais

InitializeConstructors ();

//! Ponto de entrada do kernel de chamada

main();

//! Limpe todos os dtors dinâmicos

Exit();

#ifdef ARCH\_X86

\_sm cli

#fim se

for (;;);

}

Isso é tudo o que é necessário! Enquanto o ponto de entrada estiver definido como **kernel\_entry** , esta rotina será colocada no endereço base inicial - O que deve ser definido como 1 MB

**Demo**

[Demo Download](http://www.brokenthorn.com/Resources/Demos/Demo5_1.zip) (MSVC ++)

Esta demo carrega e executa um kernel de 32 bits escrito em MSVC ++ 2005. Também inclui todo o código-fonte neste tutorial também.

**Conclusão**

Muitos conceitos neste tutorial são muito simples, não é? Mostramos a configuração do MSVC ++ 2005 para que possamos usar o compilador para uso nos kernels do sistema operacional. Nós também vimos como criar um ambiente básico de tempo de execução C ++, chamadas ctor e dtor, gerenciamento de funções virtuais e operadores globais.

Nos próximos tutoriais, iremos a analisar a criação de envios para diferentes compiladores. Este tutorial abordou a configuração do MSVC ++ 2005.

Este tutorial foi difícil de escrever - e ainda estou para terminar. Há tantas opções que o MSVC ++ tem, e descrever essas opções em detalhes levará muito tempo. Eu queria encontrar uma maneira de combinar contexto, não apenas uma lista de configurações de opções "faça isso". Ainda estou decidindo um estilo de formato para isso.