Controlando o Vazamento de Memória em Jogos Implementados em C++

EDUARDO JOSÉ TORRES SAMPAIO ROCHA
GEBER LISBOA RAMALHO
ANDRÉ SANTOS

CIn/UFPE – Centro de Informática - Caixa Postal 7851, 50732-970, Recife, PE, Brasil
{ejtsr,glr,alms}@cin.ufpe.br

Abstract

Memory leakage is a common problem in games developed in C++. Some solutions have been available for a while in Software Engineering literature, but it seems that the computer games community does have not used them yet. This paper discusses the use and extension of two techniques for controlling memory leakage.

Keywords: *Memory Leak, C++, Smart Pointers*

1 Introdução

Jogos de computador requerem alto desempenho, o que tem levado a adoção de C++ como linguagem de programação. Infelizmente, a ausência de um gerenciador de memória em C++ faz com que os "vazamentos de memória" sejam muito comuns nos programas. De fato, fica a cargo do programador desalocar toda a memória dinâmica o que deixa muita margem para erros.

Há soluções na literatura de engenharia de software[1][9], que com as devidas extensões poderiam ser utilizadas no desenvolvimento de jogos. Entretanto, pelo que pudemos constatar examinando os principais motores *(game engines)* de código aberto (Gênesis 3D[2], Crystal Space[3] e o Golgotha[4]), nenhuma destas soluções são realmente utilizadas na prática.

Este documento apresenta duas técnicas visando minimizar problemas de vazamentos de memória. Estas técnicas foram validadas no desenvolvimento do ForgeV8i, um motor para jogos isométricos baseado no ForgeV8 [5].

A próxima seção fala um pouco da motivação que levou a elaborar tais técnicas. A seção 3 mostra as duas técnicas em maior detalhe. E a seção 4 traz as conclusões e trabalhos futuros.

2 Em busca da Solução

A implementação de um sistema para gerenciar a memória é algo muito complexo e qualquer que seja ela, não pode levar à perda de desempenho, que é um atributo chave em jogos.

Uma possível solução é desenvolver ou adotar algum gerenciador de memória já disponível[6]. Infelizmente, o código de um gerenciador de memória é algo muito complexo, que pode aumentar consideravelmente o número de *bugs* de um motor de jogos. Além disso, ainda teria que se levar em consideração a que momento o gerenciador iria realizar a coleta automática de "lixo" (desalocação automática de áreas de memória que garantidamente não serão mais acessadas), pois se trata de uma aplicação de tempo real.

Tentamos então uma solução mais adaptada e fácil de implementar no contexto de jogos. Verificamos que a grande maioria dos casos de "vazamento de memória" ocorre na manipulação de *strings* (ponteiros para *char*) e na desalocação de objetos criados dinamicamente. Identificado isto, adotamos duas técnicas que foram utilizadas em todo o projeto do ForgeV8i, diminuindo significativamente os problemas de vazamento de memória.

3 Técnicas para Minimizar o Vazamento de Memória

A seguir apresentamos duas técnicas. A primeira nada mais é que uma recomendação a ser seguida. A segunda é baseada em uma solução já difundida na comunidade de engenharia de software: *Smart Pointers*[1]. Esta solução teve que ser modificada para não degradar o desempenho do sistema.

3.1 Manipulação de Strings

A *string* de caracteres é imprescindível em qualquer aplicação. Em C e C++, a maior parte do tratamento é feito com um ponteiro para caracteres (`char*`). Veja a seguinte função:

```
void printName(){
   char *ret = new char[4];
   strcpy(ret,"String Grande");
   std::cout<< ret;
};
```

A função aloca uma área de memória em seguida copia uma string que passa do limite alocado. Este é um erro comum. E além desta, existem várias outras armadilhas que envolvem os ponteiros para `char`.

A solução para a manipulação de *strings* é bem simples: a utilização de uma classe *String*. Existem várias implementações, mas a que já se tornou padrão é a classe *string* da STDLib (*Standard Library*). No ForgeV8i nós adotamos a `std::string` como padrão. Ela se encarrega de alocar e desalocar o espaço em memória necessário para a manipulação de *strings*. O código acima pode ser reescrito da seguinte forma:

```
void printName(){
   string ret = new String("String Grande");
      std::cout<< ret;
};
```

A classe `string` possui vários métodos que facilitam o gerenciamento da alocação de *strings*.

Adotando-se esta técnica em todo o código, já se diminui bastante o vazamento de memória.

3.2 Desalocando Objetos

C++ possui o conceito de destrutor que é usado para desalocar um objeto da memória. Entretanto, a linguagem não deixa claro quem deve fazê-lo. Este é um dos principais problemas encontrados nos motores de jogos estudados que leva a um vazamento de memória. Considere a seguinte função:

```
string *getName(){
   string *ret = new string("Teste");
   return ret;
};
```

A função alocou na *heap* um objeto *string* e retornou a referência. Quem deve desalocar tal objeto? Tendo-se acesso ao código, pode-se verificar que a função não guarda nenhuma referência para a string, seja ela via variável estática ou, no caso das classes, via variável membro. Entretanto, se o acesso ao código não existe, a definição de quem deve desalocar o objeto não fica clara e isto termina levando a vazamento de memória.

Foi por isto que se resolveu usar uma solução baseada nos *Smart Pointers*. *Smart Pointers* são objetos que gerenciam o número de referências a um outro objeto. Quando este número chega a zero, uma ação é tomada, como por exemplo, desalocar o objeto da memória. Esta solução existe na *web* há pelo menos 5 anos. Entretanto, tínhamos que implementar algo que não tivesse um grande impacto no desempenho do motor.

Para entender melhor o que foi feito para não impactar o desempenho, vamos primeiro ver um exemplo de estruturação de um *smart pointer*:

```
<template class P>
class SmartPtr {
public:
  SmartPtr (P* realPtr =0);
  SmartPtr (const SmartPtr &cpy);
  ~SmartPtr ();
  SmartPtr& operator=(const SmartPtr& rhs);
  P* operator->() const;
   P& operator*() const;
private:
P *pointer;
};
```

Esta implementação garante que quando o objeto SmartPtr for destruído, o objeto

apontado por `pointer` também o será. Entretanto, ela possui alguns problemas. O que deve ocorrer quando o objeto é atribuído a outro? Quem deve desalocar quem?

Existe uma implementação na STDLib chamada `auto_ptr`[7]. Ela implementa o conceito de *ownership*. Toda vez que um `auto_ptr` é atribuído a outro, ele passa o comando para o outro. Só quem detém o comando deve desalocar o ponteiro.

Entretanto, deve-se também pensar em uma solução que também resolva um grande problema de degradação de desempenho introduzido com implementações mais elaboradas. Como os *smart pointers* vão ser muito utilizados nas chamadas e retornos das funções, o ideal é que o tamanho da classe seja de 4 *bytes,* ou seja, 32 *bits* - assumindo que o jogo seja para arquitetura Intel de 32 bits. Com este tamanho, o impacto dos *smart pointers* seria quase igual a de um inteiro. A arquitetura Intel de 32 *bits* trabalha muito bem quando trabalhamos com objetos deste tamanho. Então a idéia seria criar um repositório que controlaria o número de referências e os *smart pointers* conteriam um ponteiro para este repositório.

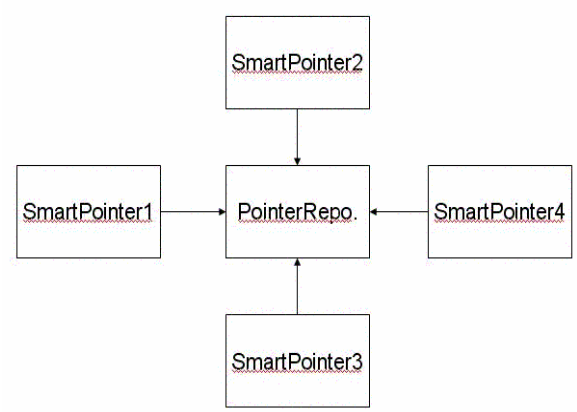

Figura 1- Repositório sendo apontado por vários *Smart Pointers*

Um ponteiro para alguma coisa tem tamanho 32 bits, ou seja, a classe que seria usada na chamada de funções teria exatamente este tamanho. Vejamos um exemplo da classe em uso:

```
SmartPtr<string> getName(){
   SmartPtr<string>    ret    =    new
string("Teste");
   return ret;
};
. . .
SmartPtr<string> name = getName();
```

A função `getName` retorna por valor um objeto *smart pointer*. Assim sendo, quando a função cria o `SmartPtr` pela primeira vez, a classe se encarrega de criar o repositório e de colocar o ponteiro para o objeto `string` dentro dela. Quando a função é retornada, a variável `ret` é copiada e atribuída a `name`, o que faz com que a referência seja incrementada, e com que `name` aponte para o repositório criado anteriormente. Quando `ret` é desalocada da pilha, a referência no repositório é decrementada. Todavia, ainda existe uma referência, portanto a *string* não é desalocada. Quando o bloco que contém `name` termina, e se não foi criada mais nenhuma referência para o repositório, a variável `name` é desalocada, o que faz com que a referência no repositório chegue a zero e a *string* seja desalocada.

Esta versão modificada dos *smart pointers* minimiza bastante o vazamento de memória, pois a dúvida de quem deve desalocar o objeto acaba. Contudo, uma breve consideração sobre espaço merece ser feita. Fica claro que esta solução privilegia o desempenho sobre um pequeno detrimento do espaço em memória utilizado. Supondo que a implementação do repositório possua *8 bytes*, existiria uma penalidade de 8 *bytes* sobre os apontadores normais da linguagem para cada objeto que é referenciado usando-se este esquema. Explicando melhor, se no esquema normal da linguagem tivermos 4 ponteiros apontando para uma área de memória, isto implicaria em 16 bytes (4 bytes * 4). No esquema proposto nós teríamos 16 bytes dos *smart pointers*, mais 8 bytes do repositório.

Além do espaço perdido, existem alguns problemas na utilização que merecem ser conhecido. O primeiro deles é uma perda na transparência. Como os *smart pointers* não são construções *built-ins* da linguagem, a

utilização deles implica em situações como a seguinte:

```
SmartPtr<string> mes = new string
("Teste");
Std::cout<< mes.getRealPointer() <<
"\n";
```

A classe std::io não sabe como imprimir um objeto do tipo SmartPtr. É claro que isto é uma situação fácil de resolver, mas o propósito deste exemplo foi mostrar que as coisas deixaram de ser tão simples.

Um outro problema é o seguinte. Suponha que temos uma Classe chamada `Veiculo`. Temos ainda `Carro` e `Caminhao` que herdam de `Veiculo`. Dada a seguinte função:

```
void stop(SmartPtr<Veiculo> vei);
```

O que ocorreria se a seguinte chamada fosse feita:

```
SmartPtr<Carro> car;
stop(car);
```

Neste caso, nós teríamos um erro de compilação. Isto acontece porque não existe uma maneira do compilador converter o *smart pointer* de Carro para Veiculo.

Uma solução para este problema é a utilização de uma construção em C++ que se chama *member template*. A idéia é fazer um template de um determinado método:

```
template <class t>
operator SmartPtr<t> {
  return SmartPtr<t> (pointer);
};
```

Esta construção faz com que o compilador tenha como fazer a conversão de um tipo para o outro. Existem alguns problemas com esta abordagem. O principal é que o compilador da Microsoft Visual C++ 6.0 não suporta tal construção. Isto obriga o programador a ter que criar um método para cada classe da hierarquia. Se a hierarquia está bem definida, isto não é um problema muito sério. No caso do método `stop` acima, mais 2 métodos teriam que ser criados: um para a classe Carro e outro para a classe `Caminhao`.

Em suma, a penalidade no desempenho causada pelo uso dos *smart pointers* pode ser minimizada com a construção do repositório para os ponteiros. As desvantagens advindas do uso deste repositório (como a falta de transparência, e o problema da herança), podem ser remediadas como discutido.

4 Conclusões

O uso das técnicas aqui mostradas diminui o vazamento de memória, o que é muito útil no desenvolvimento de jogos. Estas técnicas foram validadas na construção do ForgeV8i.

O nosso próximo passo será aprofundar o estudo do problema de vazamento de memória no novo Visual C++ .NET da Microsoft.

5 Referências

[1] Alexandrescu, A. Modern C++ Design – Generic Programming and Design Patterns Applied, Addison Wesley, 2001.

[2] Genesis 3D, http://www.genesis3d.com (14/07/02)

[3] Crystal Space, http://crystal.sourceforge.net (09/08/02)

[4] Golghota, http://www.planetquake.com/golgotha (20/07/02)

[5] MADEIRA, Charles. Forge V8: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia. Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciências da Computação – Recife, 2001.

[6] A garbage collector for C and C++, http://www.hpl.hp.com/personal/Hans_Boehm/gc/ (15/07/02)

[7] Standard C++ Library General User's Guide http://www.science.uva.nl/ict/documentation/Sun_Compilers_5.0/stdlib/stdug/general/19.htm (08/08/02)

[8] Managed C++, http://www.gotdotnet.com/team/cplusplus/ (16/08/02)

[9] Murray, R. C++ Strategies and Tactics, Addison-Wesley, 1993.