Brechas na memória

Introdução

Vazamentos de memória (ou *memory leaks*) ocorrem quando a referência a memórias dinamicamente alocada pela aplicação são perdidas, tornando essas regiões da memória inacessíveis a aplicação. Este artigo discute como evitar que isso ocorra em Symbian C++.

Quando se desenvolve para dispositivos móveis, onde os recursos são limitados, o programador deve evitar que vazamentos de memória ocorram em seu projeto. Como não existe uma gerência automática de memória em `c++` cabe ao programador ser cauteloso quanto a alocação dinâmica de memória. Este processo é bastante dispendioso e de difícil depuração.

Como ocorre um vazamento de memória?

Um vazamento de memória ocorre quando a referência à memória apontada por um ponteiro é perdida, conseqüentemente a memória alocada por ele não poderá mais ser acessada.

Imagine o seguinte código:

```
void bla()
{
    CMyApp* self = new (ELeave) CMyApp;
    self->ConstructL();
    delete self;
}
```

`self` é um ponteiro local que está alocando na memória heap um objeto de CMyApp. Após a instanciação do objeto, `self` chama a função ConstructL().

Como convenção de nomes adotada pelo Symbian funções que podem lançar exceções (abandonos na nomenclatura Symbian) devem terminar em **L**.

No exemplo acima se ConstructL() lançar uma exceção (abandonar) o código após a sua chamada nunca será executado e a memória alocada por `self` nunca será liberada.

Diferente do mecanismo de exceção padrão `c++` o Symbian utiliza um mecanismo próprio para tratamento de erros em tempo de execução. Por este motivo o programador deve estar atento a variáveis locais que alocam recursos uma vez que ao ocorrer um abandono, dentro de uma função, o Symbian irá destruir a pilha da função (o que inclui suas variáveis locais) porém não irá liberar os recursos previamente alocados por essas variáveis, ocasionando sua perda. Também vale ressaltar que o destrutor (caso essas variáveis sejam objetos) não será chamado, por esse motivo classes que implementem um destrutor nunca devem ser colocadas na stack.

Ocorrendo um abandono em `ConstructL()` o ponteiro `self` é destruído porém sem liberar a memória que alocou, teremos no código acima um vazamento de memória e isto deve ser evitado.

Regras para evitar vazamentos de memória

Seguindo algumas regras é possível tornar o código mais seguro e evitar o desperdício de recursos.

Regra nº1

Ponteiros locais que aloquem memória dinamicamente devem ser colocados na pilha de limpeza (*cleanup stack*) caso façam chamadas à métodos que possam gerar abandonos. Isto é necessário pois como explicado acima ao ocorrer um abandono as variáveis locais serão destruídas porém os recursos alocados por elas não serão liberados. Uma vez que o ponteiro esteja na pilha de limpeza, esta se encarregará de liberar os recursos alocados na ocorrência de uma falha.

ex:

```
CMyApp* self = new (ELeave) CMyApp;
CleanupStack::PushL(self); // Ponteiro adicionado a pilha de limpeza
self->ConstructL(); // Função que pode gerar um abandono
CleanupStack::Pop(self); // Removendo o ponteiro da pilha de limpeza
```

Após o código com possibilidade de gerar abandono ser executado o ponteiro deve ser removido da pilha de limpeza caso não faça nenhuma outra chamada a códigos que abandonem.

Regra nº2

Atributos de uma classe que alocam recursos não devem ser colocados na pilha de limpeza uma vez que quando um código abandonar a destruição da *stack* não o afetará. Estes atributos devem ser destruídos no *destrutor* da classe que o contem.

Por convenção de nomes, em

(C) Copyright Nokia 2013. All rights reserved.

Symbian, membros de dados de uma classe iniciam com a letra **i**.

```
class Carro;

class Pessoa
{
    public:
      void bla();
    private:
      Carro* iCar;

}

void Pessoa::bla()
{
     iCar = new (ELeave) Carro;
     iCar->funcAbandonaL(); // Função que abandona
}
```

No exemplo acima caso a função `funcAbandonaL()` lance um abandono, a pilha da função `bla()` será destruída mas como `iCar` não é uma variável local o ponteiro não será destruído e a memória que foi alocada com o operador *new* não será perdida.

Regra nº3

O construtor e destrutor de uma classe não podem abandonar. Ocorrendo um abandono no construtor não haverá como liberar um recurso que tenha sido alocado antes do abandono. O mesmo serve para o destrutor que interromperá sua execução caso possua algum código que abandone e por este motivo não poderá liberar outros recursos após o código que abandonou.

Para alocar recursos durante a instanciação de um objeto deve-se utilizar o `Construtor em duas fases`. E caso seja necessário liberar recursos ao fim de vida de um objeto é aconselhável criar uma função específica para esse fim que deverá ser chamada antes da destruição do objeto, dessa forma possibilitando ao programador gerenciar possíveis erros.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.