Otimizando C++ para Sistemas Embarcados

Luiz de Barros Oliveira Neto
Lboneto@gmail.com
14/11/2009

Por que usar C++ em um Sistema Embarcado?

- Aplicações mais complexas
- Encapsulamento e controle de acesso
- Melhor testabilidade e mantenabilidade
- Melhor reuso de código
- Checagem mais forte de tipos
- Portabilidade

Cuidado com “O Lado **não Embedded** da Força”

Aproxime se da Plataforma de Hardware

- Antes de Iniciar o Desenvolvimento em C++, conheca os recursos e limitações da sua plataforma de hardware e de software:
  - Microcontrolador
  - Capacidades de Memória
  - Tamanho da “palavra” do microcontrolador
  - Disponibilidade de MMU?
  - Frequência de Clock
  - Sistema Operacional / Núcleo de Tempo Real
  - Capacidades para execução de Múltiplas Tarefas/Threads
  - Disponibilidade de “Garbage Collector”
  - Estabeleça um canal de comunicação com a área de Engenharia logo no início do projeto – Se possível, antes da especificação da plataforma de hardware.

Ambiente de Desenvolvimento, Debug e Teste pode causar distúrbios na força

- Ambientes Integrados de Desenvolvimento, Depuração e Teste (incluindo simuladores) em plataformas Host podem deixar o desenvolvimento de software embarcado muito semelhante do ambiente Host.
- Boa parte do Software desenvolvido em plataformas GNU/Linux embarcado pode, por exemplo ser desenvolvida e testada em condições “similares” em um PC.
- O desenvolvimento de Software Embarcado não pode abstrair o lado “embedded”.

Cuidado com a Arquitetura – ela pode facilmente conduzir para o lado não embedded da força

- O projeto de um software embarcado em C++ deve SEMPRE levar em conta as capacidades e recursos da plataforma embarcada.
- Deve ser tomado cuidado especialmente com Arquiteturas “universais” portáveis entre plataformas embarcadas ou não.
- Dificilmente uma arquitetura irá conseguir resolver todos os problemas atuais e os que poderão vir a acontecer no futuro.
- Seja econômico e “Pense Embarcado”

Reduzindo Riscos e mantendo-se no lado Embedded da Força

- Sempre que possível, realize o maior número possível de testes na plataforma de hardware real do produto.
- Testes em ambientes Host podem mascarar problemas e dificultar soluções.
- Testes de Integração ao longo do desenvolvimento podem reduzir riscos de problemas ligados a interdependências e concorrência por recursos.
- Tenha seu time de engenharia de hardware como parceiro. Compartilhe responsabilidades, realize provas de conceito.

Otimizações que não otimizam! [Dewhurst 2009].

- Register
- Multiplicações ou Deslocamentos
- Qual a melhor forma de “zerar” uma variável?

Register

```
int MinhaClasse::calculeRapido()
{
    register int i, j, k;
    ~~~
}
```

Os compiladores estão se tornando a cada dia mais “espertos” . Hoje em dia, eles são capazes de realizar este tipo de otimização automaticamente (não seguem a orientação do programador) e normalmente o fazem de forma mais eficiente do que nós.

Multiplicar ou Deslocar?

- Qual das operações é mais eficiente?

```
a *= 16;
```

```
a  = a << 4;
```

Os compiladores C/C++ atuais já são capazes de realizar este tipo de otimização automaticamente e, o farão, apenas se for realmente mais eficiente

Como atribuir valor zero?

- Qual a melhor forma em termos de performance para atribuir valor zero a uma variável inteira?

A=0;

A ^= A;

Os compiladores já são capazer de realizar este tipo de otimização, sem a necessidade de obscurecer o código.

Recursos “Gratúitos”

- Os seguintes recursos são disponíveis sem “custo” na linguagem C++ [Dewhurst 2009], [Quiroz 1998] :
  - Namespaces
  - Sobrecarga de Funções
  - Funções-membro estáticas
  - Funções-membro não virtuais
  - Inicializadores padrão para argumentos
  - Proteção de Acesso
  - New and delete
  - Construtores e Destrutores (*)
  - Passagem de parâmetros por referência

Recursos “Baratos”

- Funções-membro Virtuais
- Funções “inline”

Funções-Membro Virtuais

- Permitem implementações de diferentes comportamentos, baseado no tipo da classe utilizando uma “interface” comum;
- Na maioria dos casos, a “ligação” da função membro correspondente é feita apenas em “run-time” com a utilização de uma tabela associada à classe (VTBL);
- Possuem um custo em tempo de execução praticamente desprezível (2 a 4 instruções por chamada) [Saks 2003], [ISO 2005] na maioria dos casos, podendo ser utilizadas de forma eficiente em sistemas embarcados;

Funções-Membro Virtuais

```
Class Point
{
    public:
        int type;
        int x;
        int y;
        virtual void draw();
}

Class Circle: public Point;
Class Square: public Point;
```

Implementação via Switch

- O custo de dereferenciamento e comparação normalmente será maior do que o da chamada do método Virtual.

```
Switch (objeto->type)
{
   case Circle:
       desenhaCirculo(objeto);
      break;
 case Square:
      desenhaQuadrado(objeto);
      break;
 case Point:
      desenhaPonto(objeto);
      break;
}
```

Funções Inline

- A declaração de Funções como “inline” é eficiente para casos onde elas não sejam grandes o suficiente a ponto de justificar serem definidas realmente como funções [Dewhurst 2009], [Quiroz 1998];
- Funções complexas não devem ser “inlined” sob pena de haver um aumento do tamanho do código;
- O custo de uma chamada em microcontroladores modernos é bem mais simples hoje do que na era CISC [Quiroz 1998];
- Indicado para Funções pequenas e chamadas com grande frequência;
- Não é indicado para construtores e destrutores, pois eles podem conter quantidade significativa de código gerado implicitamente.

Usar com moderação

- Templates
- Herança Múltipla
- Alocação Dinâmica

Templates

- São considerados a chave para modernos códigos estáticos e “type safe” [Stroustrup 2005];
- Não há overhead adicional em se utilizar chamadas de função ou funções membro de classes template. Templates podem, se usados com cuidado, reduzir o overhead de uma chamada de função devido ao “inlining” mais eficiente [ISO 2005] ;
- É recomendado que não sejam executados em templates códigos que exigam processamentos complexos (templates podem “inchar”o código se não forem bem utilizadas;
- Cuidado com a STL em compiladores embarcados/não embarcados.

Usando Templates (1)

```
// stack.hpp
template<typename T, int size> class Stack
{
private:
    T stack_[size];
    int top_;

public:
    Stack();
    void push(T);
    T pop();
};
```

Usando Templates (2)

```
template<typename T, int size>
Stack<T, size>::Stack()
{
   top_ = 0;
}

template<typename T, int size>
void Stack<T, size>::push(T item)
{
   if (top_ == size)
      { /* overflow */ }
   stack_[top_++] = item;
}

template<typename T, int size>
T Stack<T, size>::pop()
{
   if (top_ == 0)
      { /* underflow */ }
   return stack_[--top_];
}
```

Usando Templates (3)

```
// main.cpp
#include "stack.hpp"

Stack<int, 100> main_int_stack;
int int_val;


int main()
{
    main_int_stack.push(100);
    int_val = main_int_stack.pop();
}
```

Exemplos de Utilização

- Pilha
- Filas Circulares
- HashTables
- Listeners e Receivers
- Message Queues

Herança Múltipla

Class Handle: public SerialPort, PPP

- Permite combinar a funcionadade de 2 ou mais classes.
- Deve ser usado com cuidado em sistemas embarcados pois sempre necessitarão de tabelas de funções virtuais maiores do que nos casos de herança simples.

Alocação Dinâmica de Memória

- Introduz um overhead não determinístico de tempo de execução, em implementações padrão [ISO 2005], o que dificulta a sua utilização em sistemas de tempo real;
- Sistemas embarcados normalmente estão sujeitos a problemas devidos à fragmentação de memória, o que exige cuidado na utilização de alocação dinâmica [Dailey, 1999];

Melhorando a Alocação de Memória

- Os operadores new e delete providos pelo ambiente podem ser substituidos por versões mais eficientes e determinísticas implementadas pelo usuário;
- Estratégias de alocação baseadas em pools ou partições de tamanhos variados podem ser usadas para melhorar o tempo de resposta de alocações;
- Sempre que possível, deve se tentar alocar memória de forma estática ou por meio da pilha.

Evitar

- Classes Base Virtuais
- Dynamic Cast
- Run-time Type Identification
- Exceções
- Pouca utilização de ROM

Classes Base Virtuais

- Introduzem grande complexidade nos construtores e Destrutores [Dewhurst 2009] ;
- Causam ineficiencia por adicionar um overhead ao código das classes [Dewhurst 2009];
- Normalmente, os sistemas embarcados utilizam hierarquia de classes mais “lineares”, não necessitando deste recurso.

Dynamic Casts

- Na maioria das implementações, dynamic_casts<> produzem um overhead não determístico e que depende de detalhes da implementação [ISO 2005];
- É recomendado se utilizar o operator typeid, que é a maneira mais simples de se obter o tipo de uma classe.

Run-Time Type Identification

- RTTI é um recurso que necessita de espaço, pois requér uma estrutura para realizar a correspondência de tipo e tempo, pois a estrutura precisa ser mantida [Quiroz 1998].
- Deve ser evitado pois introduz um overhead considerável no código.

Exceções

- O tratamento de exceções pode introduzir um overhead em cada chamada de função [ISO 2005];
- É um dos recursos mais “caros” e normalmente demanda a utilização de RTTI [QUIROZ 1998].

Pouca utilização de ROM

- Sistemas Embarcados normalmente possuem restrições quanto à disponibilidade de memória RAM e portanto, o máximo de código deve poder ser executado de memoria ROM;
- Sistemas operacionais modernos podem suportar o compartilhamento de blocos de código e dados constantes entre múltiplas instâncias de programa ou objetos;
- Objetos que possuam inicializadores constantes e que não possam ser modificáveis são ROMable;
- Construtores em geral são inicializados dinamicamente, mas podem ROMable caso sejam identificados valores constantes na inicialização, durante a compilação.

Examinando o Assembly

- A maioria dos Compiladores e Ambientes de Desenvolvimento permite de forma muito simples a visualização do código assembler gerado. É importante verificar os códigos em assembler e checar por pontos de “inchaço”. O compilador algumas vezes pode não ser tão esperto como você pensa.

Referências

- [Saks 2003]. Dan Saks, "Reducing Run-Time Overhead in C++ Programs", 2003 Embedded Systems Conference San Francisco;
- [Dewhurst 2009]. Stephen C. Dewhurst, "Undercover C++:What's Efficient and What Isn't?", 2009 Embedded Systems Conference Silicon Valley;
- [ISO 2005] "Technical Report on C++ Performance". ISO/IEC PDTR 18015
- [Quiroz 1998] César A. Quiroz "Using C++ Efficiently in Embedded Applications" 1998 Embedded Systems Conference San Jose;
- [Stroustrup 2005] Bjarne Stroustrup "Abstraction and the C++ machine model" ICESS
- [Dailey 1999] Aaron Dailley "Effective C++ Memory Allocation" Embedded Systems Programming http://www.embedded.com/1999/9901/9901feat2.htm

Obrigado

Luiz de Barros Oliveira Neto

E-mail: Lboneto@gmail.com