ISSN

COMUNICADO TÉCNICO

Nº. 2, maio/97, p.1-5

MODULARIZAÇÃO DE PROGRAMAS C/C++

Evandro Bacarin[1]

1. Introdução

Modularização é reconhecida como sendo a única técnica disponível para reduzir a complexidade do projeto e da implementação de grandes sistemas (Ghezzi & Jazayeri, 1987). Consiste em dividir o sistema em módulos menores de forma que cada módulo tenha uma única finalidade dentro do sistema e que sejam implementados independentes uns dos outros.

De acordo com Shooman (1983), esta técnica apresenta vantagens como: manutenção mais fácil do sistema, maior facilidade para escrever e depurar programas, permite dividir programas em módulos menores e atribuir a implementação de módulos mais complexos a programadores mais experientes. Ainda segundo Shooman (1983), em entrevistas feitas com programadores que experimentaram esta técnica, os seguintes benefícios foram reportados:

Benefício	Programadores que reportaram o benefício (%
Maior facilidade de manutenção	89
Melhor projeto do programa	85
Maior facilidade de teste do programa	78
Controle mais apurado do projeto	67

Também foram apresentadas algumas desvantagens:

Desvantagem	Programadores que reportaram a desvantagem (%)
Maior utilização do computador durante o desenvolvimento	28
Programas menos eficientes	27
Maior dificuldade de treinar programadores	19
Maior dificuldade de documentação	13

ador da Embrapa-CNPTIA, Caixa Postal 6041 - 13083-970 - Barão Geraldo - Campinas, SP.

Em vista disso, algumas linguagens de programação (Modula-2 e Oberon, por exemplo) implementam mecanismos que permitem a organização de programas em módulos, proporcionando um controle mais efetivo do escopo de aplicação dos nomes (de constantes, variáveis, procedimentos, etc.) do programa.

As linguagens C e C++, por sua vez, não possuem estruturas sintáticas que implementam este conceito. A modularização é obtida com maior dificuldade através da definição de macros expandidas por seus pré-processadores.

Este comunicado, portanto, apresenta algumas técnicas que permitem a modularização eficaz de código C/C++.

2. Modularização

Um módulo deve ser dividido em duas partes[2]: *módulo de interface* e *módulo de implementação.* O módulo de interface (arquivo .h, .hpp) deve conter as declarações dos nomes que poderão ser usados em outros módulos. São elas: protótipos de funções, declaração de variáveis globais, declaração de constantes, declaração de tipos (estruturas, uniões, classes, typedefs, enumerações) e macros. O módulo de implementação (arquivo .c, .cc, .cxx, entre outros) contém a efetiva implementação (definição). A seguir é detalhado o formato de cada um destes arquivos.

2.1. Módulo de Interface

O módulo de interface possui o seguinte formato:

```
#ifndef <MODULO>__H
#define <MODULO>__H

/*
 *    Comentários, identificação do autor, data etc
 */

< includes >
< macros >
< declaração de constantes,tipos,estruturas,uniões,enumerações e
  classes >
< declaração de variáveis globais exportadas >
< protótipos das funções >

#endif /* <MODULO>__H */
```

As diretivas de compilação `#ifndef <MODULO>__H`, `#define <MODULO>__H` e `#endif /* <MODULO>__H */` são de vital importância e, portanto, não devem ser omitidas, pois evitam que ocorram múltiplas definições de um mesmo nome (de variável, estrutura, função, etc.) quando o arquivo for incluído por outros módulos que compõem o programa. `<MODULO>` deve ser substituído pelo nome do arquivo.

É recomendável que no início do arquivo sejam acrescentados comentários que identifiquem sua finalidade, autores, data, etc.

Na seção `<includes>` são colocadas as diretivas para inclusão de arquivos de cabeçalhos (módulos de interface) necessários para compilação deste arquivo. Nenhuma outra inclusão deve ser efetuada.

[2] Talvez excetuando o módulo principal.

A seção de macros (`#define`) deve se restringir ao absolutamente necessário. Não esquecer que não é necessário utilizar macros para definir constantes em C++.

A seguir são declaradas as constantes, tipos, estruturas, uniões, classes e enumerações que se quer exportar para outros módulos.

As variáveis globais exportadas para outros módulos devem sempre ser declaradas como externas (`extern`).

Por fim, os protótipos das funções exportadas são declarados. Recomenda-se que estes protótipos estejam agrupados segundo suas funcionalidades ("rotinas de inicialização da estrutura X", "rotinas para iteração sobre a estrutura Y", etc.).

É aconselhável que todas as declarações do módulo de interface sejam adequadamente comentadas. Sugere-se que o comentário *não* seja repetido na respectiva definição dentro do módulo de implementação, evitando inconsistências entre os comentários dos dois módulos.

O módulo de interface deve ser escrito (e comentado) de forma a permitir que um programador, apenas consultando-o, seja capaz de utilizar adequadamente os recursos que provê.

2.2. Módulo de Implementação

A organização do arquivo de implementação é semelhante ao do módulo de interface. A ordem das "seções" é semelhante. Após as diretivas de inclusão, definições de macros e definição de constantes, tipos, classes, enumerações, estruturas, uniões, são definidas as variáveis globais exportadas (declaradas como externas no módulo de interface).

A seguir, são definidas as variáveis globais utilizadas apenas no módulo de implementação. Estas variáveis devem ser declaradas como estáticas (`static`). Recomenda-se que no módulo de implementação não ocorram declarações de variáveis externas (`extern`). Em casos inevitáveis deve haver um comentário explicitando o arquivo onde a variável foi definida.

As variáveis globais devem ser comentadas indicando suas finalidades, e sempre que possível, os efeitos colaterais que as alterações de seus valores podem acarretar. Sugere-se que os comentários sobre variáveis globais exportadas (declaradas no módulo de interface) *não* sejam repetidos no módulo de implementação, evitando inconsistências entre os comentários.

A seguir, devem ser colocados os protótipos das funções implementadas no módulo e não declarados no módulo de interface. Esta é uma exigência do ANSI-C++.

Finalmente, as funções são implementadas. Recomenda-se que sejam agrupadas de acordo com algum critério como, por exemplo, agrupar métodos de uma classe, agrupar funções que manipulam um certa estrutura. Na seqüência é mostrado um esquema da organização do módulo de implementação:

```
/*
 *    Comentários ...
 */

< includes >
< macros >
< definição de constantes, tipos, estruturas, uniões, enumerações e
  classes >
< definição de variáveis globais exportadas >
< definição de variáveis globais estáticas >
< protótipos das funções >
< implementação das funções >
```

3. Exemplo

São apresentados dois arquivos que exemplificam a construção de módulos de interface e de implementação:

3.1. Módulo de Interface (pilha.h)

```
#ifndef PILHA__H
#define PILHA__H

/*
* nome: pilha.h
* descrição: Implementação de pilha de valores inteiros
* comentários:
*     Implementa uma pilha de tamanho limitado (determinado na sua
*     criação). São implementadas operações para inserção e remoção
*     de números inteiros da pilha.
*     Os tempos de inserção e remoção são da ordem O(1).
*/

class Pilha
{
  public:
    Pilha(int tam);

    void empilha(int v);
    // Insere no topo da pilha o valor v. Se a pilha estiver cheia,
    // aborta a execução do programa.

    int desempilha();
    // Retira um valor do topo da pilha. Caso a pilha esteja vazia,
    // aborta a execução do programa.

  private:
    int     *vi;  // aponta a pilha alocada dinamicamente
    int     topo; // indice da primeira posicao livre da pilha
};

extern int NumeroDePilhasInvertidas;  // conta o numero de vezes
                                      // que a funcao inverte_pilha
                                      // foi invocada.

void inverte_pilha (Pilha p);
// Inverte a pilha "p", de forma que o topo esteja colocado na
// base da pilha e a base, no lugar do topo.

#endif /* PILHA__H */
```

3.2. Módulo de Implementação (pilha.cxx)

```
/*
 * nome: pilha.cxx
 * descrição: Implementa a estrutura de dados pilha.
 *
 * comentários:
 *
 *    Implementa a estrutura de dados pilha descrita em
 *    "Data Structures and its applications", Altamirando Cebola,
 *    editora CAMPOS ARADOS, 1878.
 *    O tamanho da pilha é limitado e definido durante sua criação.
 */

#include "pilha.h"

// variáveis globais exportadas
int NumeroDePilhasInvertidas;

// variáveis globais não exportadas
static int variavel_1;

// implementação de métodos e construtores de Pilha

Pilha::Pilha (int tam)
{ ... }

void Pilha::empilha(int i)
{ ... }

int Pilha::desempilha()
{...}

// implementação de funções exportadas

void inverte_pilha (Pilha p)
{ ... }
```

4. Referências Bibliográficas

SHOOMAN, M.L. *Software engineering: design, reliability and management.* New York: McGraw-Hill Book, 1983. 683p.

GHEZZI, C.; JAZAYERI, M. *Programming language concepts.* New York: John Wiley, 1987. 428p.

IMPRESSO

Empresa Brasileira de Pesquisa Agropecuária
Centro Nacional de Pesquisa Tecnológica em Informática para a Agricultura
Ministério da Agricultura e do Abastecimento
Cidade Universitária "Zeferino Vaz" Barão Geraldo - Caixa Postal 6041
13083-970 - Campinas, SP
Telefone (019) 239-9800 Fax: (019) 239-9594