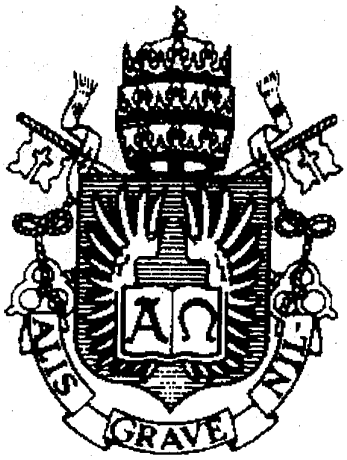

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 27/96

PG-04 Regras e Recomendações Específicas para a Programação em C++

- Versão 1.01 -

Arndt von Staa
(Editor)

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 27/96
Editor: Carlos J. P. Lucena Setembro, 1996

PG-04 Regras e Recomendações Específicas para a Programação em C++ *

- Versão 1.01 -

Arndt von Staa
(Editor)

* Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca Documentação e Informação
PUC-Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 - Rio de Janeiro, RJ
Brasil
Tel +55-21-529 9386 Fax +55-21-511 5645
E-mail: biblio@inf.puc-rio.br www: http://www.inf.puc-rio.br

PG-04 Regras e recomendações específicas para a programação em C++

Versão 1.01

editor: A.v. Staa[1]
arndt@inf.puc-rio.br

Laboratório de Engenharia de Software
Departamento de Informática
Pontifícia Universidade Católica
22453-900 Rio de Janeiro,
Brasil

Setembro 1996

PUC-RioInf.MCC 27/96

Resumo

Neste documento estabelecemos regras de estilo par programas redigidos em C++ no que esta linguagem difere de C.

Palavras chave: estilo de programação C++, manutenibilidade.

Abstract

In this document we establish style rules for programs written in C++, where this language differs from C.

Keywords: C++ coding style, maintainability.

---

[1] Trabalho apoiado por: CNPq, Bolsa de Pesquisador 300029/92-6, CENPES/Petrobrás, Itautec/ Philco

Histórico de evolução

PG-04 Regras e recomendações específicas para a programação em C++

Gestor: Laboratório de Engenharia de Software
Departamento de Informática, PUC-Rio

Arquivo: Nrcpp01
Editado: 16 setembro, 1996
Impresso: 16 setembro, 1996

Documentos correlatos
PG-01 Regras e recomendações para a escolha de nomes de elementos em programas C e C++
PG-02 Regras e recomendações para o uso de constantes simbólicas em C e C++
PG-03 Regras e recomendações para a programação em C e C++
PG-05 Regras e recomendações para a inclusão de especificações

Versão V1.01

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/jul/1996
Data entrada em vigor: 01/jul/1996

Data de início da próxima revisão 01/jan/1997

Descrição de evolução
correções ortográficas e sintáticas
acertos nos exemplos

Descrição da retroação

Versão V1.00

Editores:
Arndt von Staa (PUC-Rio)

Status: Em uso

Data homologação: 01/mar/1996
Data entrada em vigor: 01/mar/1996

Data de início da próxima revisão 01/jan/1997

Descrição de evolução

Descrição da retroação

Créditos

Revisores

André Derraik	(TeCGraf PUC-Rio)	Versão 1.0
Claudio de Oliveira	(PUC-PR)	Versão 1.01
Geraldo Machado Costa	(LES PUC-Rio)	Versão 1.0
Lincoln Nobumiti Kanamori	(Itautec Philco)	Versão 1.0
Pedro Alexandre O. Giovani	(Itautec Philco)	Versão 1.0
Pedro Jorge E. Hübscher	(LES PUC-Rio)	Versão 1.0
Renan Martins Baptista	(CENPES)	Versão 1.0
Rosa Maria Ramalho Correia	(Itautec Philco)	Versão 1.0

Apoio

CENPES Petrobrás
CNPq
Itautec Philco

Marcas registradas e nomes de produtos

MS-DOS e Windows são marcas registradas da Microsoft Corp

Sumário

1. Objetivo

O presente conjunto de regras e recomendações tem como objetivos:

- definir um estilo de programação uniforme para programas em C++ no que toca as características específicas de C++. O estilo comum para programas redigidos em C ou em C++ está contido no documento *PG-03 Regras e recomendações para a programação em C e C++*.
- conduzir a programas que facilitem verificar se estão corretos.
- conduzir a programas de fácil manutenção.

Como resultado destas regras espera-se que:

- todos os programas tenham um estilo consistente e uniforme, independentemente de quem os escreveu ou está mantendo.
- seja fácil entender corretamente um trecho de código sendo lido, sem precisar recorrer a documentação complementar.
- os programas contenham poucos erros decorrentes do uso de construções propensas a provocar problemas.
- os programadores possam rápida e facilmente entender e corretamente modificar programas redigidos por outros.

A presente norma é específica para a linguagem C++, tratando somente das porções de C++ que não fazem parte da linguagem C. Na norma *PG-03 Regras e recomendações para a programação em C e C++* encontra-se um outro conjunto de regras e recomendações que se aplicam a programas redigidos em C++. Esta divisão foi feita para reduzir a duplicação de texto, uma vez que C++ é uma linguagem que contém C.

Como conseqüência da adoção da presente norma, espera-se uma redução significativa do número de erros de programação, quando comparado com a programação realizada sem regras bem definidas. Além de procurarem atingir o ideal de programas corretos por construção, a presente norma estabelece um estilo uniforme para programação na linguagem C++. Desta forma elimina-se o personalismo tão pernicioso à qualidade de programas, facilitando-se o desenvolvimento e a manutenção de programas por várias pessoas.

2. Motivação

C++ é uma linguagem complexa desenvolvida com o objetivo de integrar o poder da programação orientada a objetos à linguagem C, preservando a compatibilidade com a grande quantidade de linhas de código C já existentes nas diversas empresas e bibliotecas de programas. É sabido também que tanto C como C++ permitem construções que são altamente sujeitas a erro ou são difíceis de compreender. Nesta norma procuramos restringir o uso da linguagem C++ nas partes em que ela especificamente é diferente de C, sem reduzirmos a capacidade de expressão e de criação de programas eficientes.

Ao contrário da linguagem C, a linguagem C++ ainda não possui um padrão ANSI, mas já existe um comitê responsável por esta padronização (comitê ANSI X3J16). A homologação de um padrão ANSI-C++ e a sua adoção pelos desenvolvedores de compiladores, pode impactar algumas das regras e recomendações contidas na presente norma. Em virtude de ainda não existir uma norma de linguagem C++ em processo final de homologação, podem ocorrer diferenças em função do compilador adotado. A presente norma procura manter-se dentro do que parece ser consenso no comitê de normalização.

3. Definição da norma

3.1 Acesso a membros de classes

Regra 1: Os qualificadores de visibilidade `public`, `protected` e `private` de uma classe devem ser declarados nesta ordem.

Colocando a sessão `public` logo a frente no código, toda a informação de interesse do usuário se encontrará no início da classe. A sessão `protected` é do interesse a quem considera herdar da classe. A seção `private` precisa ser declarada, mas contém detalhes que, em princípio, não deveriam ser de conhecimento dos usuários da classe. O agrupamento dos membros baseado na visibilidade também facilita encontrar informação de uso da classe sempre que necessário.

Regra 2: Funções membro, *métodos*, não devem ser definidas no código de definição da classe.

As definições de classes devem conter somente os dados e os protótipos dos métodos. Caso se deseje definir um método simples como `inline`, isto deverá ser feito de forma explícita, incorporando o código de um método `inline` no módulo de definição da classe. Esta forma de organizar o código facilita entender a composição da classe, pois separa bem o que é declarativo do que é executável. Nas regras definidas na seção *3.2 Métodos `inline`* discutiremos em mais profundidade o uso de métodos `inline`.

Exemplo:

```
// Solução que não deve ser usada
   class String
   {
      public:
         int Length( ) const
         {
            return Len ;
         }
      private:
         int Len;
   } ;

// Solução seguindo a Regra 2
   class String
   {
      public:
         int Length( ) const ;
      private:
         int Len ;
   } ;
   inline int String::Length( ) const
   {
      return Len;
   } ;
```

Regra 3: Nunca especifique como `public` ou `protected` um dado que é membro de uma classe.

Exceção 4: No caso de interfaces com outras linguagens, por exemplo C, pode ser necessária a declaração de `struct` com visibilidade `public`.

Uma variável `public` representa uma violação do princípio de encapsulamento de dados, que é um dos princípios básicos de programação orientada a objetos. Pela mesma razão o uso de variáveis `protected` em uma classe não é recomendável, embora sejam visíveis a um universo menor de classes. Ao invés de declarar dados `public` ou `protected`, utilize funções de acesso a estes dados.

A seqüência da declaração assegura que o compilador produza mensagens de erro ao tentar utilizar dados mais restritos em domínios de visibilidade menos restritos, o que reforça a adoção da Regra 1.

3.2 Métodos `inline`

Recom. 5: Uma *função de acesso*, isto é, um método que retorna o valor de um membro de uma classe, deve ser `inline`.

Recom. 6: Uma *forwarding function*, isto é, um método cujo corpo seja formado por exatamente uma chamada de função, deve ser `inline`.

Recom. 7: Um método com corpo muito pequeno — por exemplo: expressões simples, ou uma a duas linhas de código — deve ser `inline`, exceto quando for um construtor ou um destrutor, ou quando contiver controles.

Funções de acesso possuem, em geral, um corpo muito pequeno. Isto vale também para *forwarding functions*. A definição de funções de corpo pequeno como sendo `inline` otimiza o programa, tornando-o mais rápido e, em geral, menor. Isto ocorre pois são eliminados os *overheads* de chamada de função.

Regra 8: Funções que chamam métodos `inline` não devem ser `inline`.

Regra 9: Construtores e destrutores não devem ser `inline`.

Regra 10: Defina métodos `inline` explicitamente.

Regra 11: O texto de código de métodos públicos ou protegidos `inline` faz parte do módulo de definição próprio do módulo contendo a classe.

Regra 12: O texto de código de métodos privados `inline` faz parte do módulo de implementação do módulo contendo a classe.

Funções que chamam métodos `inline` são geralmente muito complexas para os compiladores e acabam sendo compiladas como funções normais. Este problema é especialmente comum com construtores e destrutores que necessitam chamar os construtores de suas classes base antes de executar o próprio código.

Cabe salientar que diversos compiladores podem optar por não expandir o código de uma função `inline` no local da chamada, caso o tamanho do corpo ultrapasse determinado limite ou a sua estrutura se torne muito complexa. Caso o compilador emita uma mensagem de advertência informando que tratou um método `inline` como um método normal, retire a declaração `inline` deste método e mova o código para o correspondente módulo de implementação.

Em alguns compiladores otimizadores pode-se deixar para o compilador decidir automaticamente se um determinado método deve ser `inline` ou não. O uso desta propriedade pode levar à construção de programas não portáteis entre diferentes compiladores.

O código de um método `inline` é expandido no local da chamada. Para isto ser possível ele precisa ser conhecido ao encontrar-se a chamada. Conseqüentemente o código fonte de funções `inline` precisa estar disponível no módulo de definição caso o método seja público ou protegido. No entanto, não precisa estar no módulo de definição se for privado. Conforme estabelecido na norma *PG-03 Regras e recomendações para a programação em C e C++*, cada módulo possui um único módulo de definição. Este deve conter somente o código fonte necessário para definir a interface própria do módulo e deve ser o menor possível. Assegurando-se que todos os módulos cliente de um dado módulo, bem como o próprio módulo, utilizem exatamente o mesmo módulo de definição, assegura-se que não ocorram erros de inconsistência de interface.

3.3 Friends

Recom. 13: Utilize `friend`'s somente para tornar disponíveis funções adicionais efetivamente necessárias, e que por alguma razão devem ser definidas fora do escopo da classe.

Em algumas situações pode ser conveniente definir uma classe contendo operações definidas fora do escopo desta classe ou mesmo definidas em outra classe. Esta situação ocorre, por exemplo, se uma função necessita de dados normalmente não requeridos pela classe. Por exemplo, em uma classe lista pode ser interessante definir um iterador desta lista como uma classe independente, já que a variável de controle da iteração não precisa

fazer parte da classe lista. Para que uma função possa ter acesso aos membros encapsulados de uma classe, é necessário que ela seja declarada `friend`. Desta forma consegue-se criar o iterador sem violar as regras de encapsulamento.

Recom. 14: Utilize `friend`'s com parcimônia.

A existência de muitos `friend`'s em uma classe é um sinal de que o projeto é pouco modular. Ao definir uma classe ou função `friend`, assegure-se que a integridade da classe não tenha sido afetada.

3.4 Métodos `const`

Regra 15: Todo método que não altere pelo menos uma das variáveis membro de um objeto deve ser declarada `const`.

A declaração `const` assegura que as variáveis de instância de objetos não serão modificadas. Para reduzir possíveis interferências entre objetos, é recomendável que se declare `const` todos os objetos que não se deseja modificar. Isto permite que o compilador assegure que valores não serão modificados acidentalmente. Métodos `const` são a única forma de se operar objetos `const`.

Regra 16: No caso do comportamento de um objeto ser dependente de dados externos, esses dados não devem ser modificados por métodos `const`.

Um método `const` deve alterar somente os dados locais desta função. Caso ele opere com ponteiros, os valores apontados devem ser constantes, ou ser valores alocados internamente à função. Para regras relativas a valores `const` consulte a norma *PG-03 Regras e recomendação para a programação em C e C++*. O exemplo a seguir ilustra o uso de métodos `const`, inclusive *overloading* com a propriedade `const`.

```
static unsigned const iDim = 1024 ;
class Buffer
{
   public:
      void Buffer( char * pszSeq ) ;
      char & operator[]( unsigned Indice ) ;  // retorna um lvalue
      char   operator[]( unsigned Indice ) const ;
...private:
      char mpv_sBuffer[ iDim ] ;
} ;

inline char & operator[]( unsigned Indice )
{
   return mpv_sBuffer[ Indice ] ;
}

inline char operator[]( unsigned Indice ) const
{
   return mpv_sBuffer[ Indice ] ;
}

Buffer::Buffer( char * pcSeq )
{
   strncpy( mpv_sBuffer , pszSeq , sizeof( mpv_sBuffer ));
}

void main( void )
{
   const Buffer CteNome = "abcd" ; // objeto constante
         Buffer VarNome = "xyzw" ; // objeto variável

   VarNome[ 1 ] = '!' ; // pode ser realizado
   CteNome[ 1 ] = '!' ; // erro de compilação
```

```
    cout << CteNome[ 1 ] << VarNome[ 2 ] ; // pode ser realizado
}
```

3.5 Construtores e destrutores

Regra 17: Uma classe que utilize new para alocar instâncias gerenciadas pela classe, deve definir um construtor de cópia.

Exceção 18: Em alguns casos é necessário que objetos compartilhem uma área de dados. Neste caso não é necessário definir um construtor de cópia. No entanto, **é necessário assegurar** que esta área de dados não seja desalocada enquanto existirem ponteiros apontando para ela.

Instâncias gerenciadas pela classe são instâncias vinculadas a variáveis membro de tipo ponteiro ou referência e que são desalocadas pelo objeto. Um construtor de cópia é recomendado quando um objeto é inicializado utilizando um objeto do mesmo tipo. Caso um objeto gerencie a alocação e desalocação de um objeto, somente o valor do ponteiro será copiado. Isto pode provocar duas chamadas do destrutor para um mesmo objeto, levando a erros de execução.

Regra 19: Todas classes utilizadas como base e que contenham funções virtuais, devem definir um destrutor virtual, mesmo que este faça nada.

Esta regra assegura o correto funcionamento quando se utiliza ponteiros para a classe. Se um ponteiro para uma classe é atribuído a uma instância de uma classe derivada dela, somente o destrutor da classe base será ativado. Caso um objeto de uma classe base sem destrutor virtual dependa da ativação do destrutor da classe derivada, o programa executará errado.

Recom. 20: Evite o uso de objetos globais em construtores e destrutores.

Exemplos da utilização correta de construtores e destrutores são apresentados no Anexo B.

Recom. 21: Evite o uso de construtores com um único parâmetro.

Construtores de um único parâmetro podem ser utilizados pelo compilador como funções de conversão implícitas. Estas conversões implícitas geram objetos temporários nem sempre conhecidos pelo programador, muitas vezes produzindo resultados inesperados.

3.6 Atribuição

Regra 22: Uma classe que utilize o operador new para alocar instâncias da classe, deve definir um operador de atribuição.

Exceção 23: Em alguns casos é necessário que objetos compartilhem áreas de dados. Neste caso, não é necessário definir um construtor de atribuição. No entanto **é necessário assegurar** que esta área de dados não seja desalocada enquanto existirem ponteiros apontando para ela.

O operador de atribuição não é herdado como os demais operadores. Caso um operador de atribuição não tenha sido explicitamente definido, ele será definido automaticamente. Tais operadores não executam cópia de conteúdo, ao invés disto copiam membros utilizando o operador atribuição para cada tipo primitivo. Como, por *default*, ponteiros são copiados por valor, uma mesma área de dados será apontada por dois ou mais ponteiros. Esta multiplicidade de ponteiros poderá levar a problemas ao desalocar o objeto, ver Regra 17.

Regra 24: Um operador de atribuição que execute uma operação destrutiva não deve ser capaz de destruir o objeto sobre o qual está operando.

Um erro comum é a atribuição de um objeto a si mesmo (a = a). Normalmente, antes da atribuição ser feita, o destrutor é executado para todas as variáveis do objeto destino alocadas no *heap*. Se um objeto for atribuído a si

próprio, os valores das variáveis serão perdidos antes de serem copiados. Ao encontrar uma atribuição a si próprio, o operador de atribuição deve deixar inalterado o objeto destino.

Recom. 25: Operadores de atribuição devem retornar a referência do tipo `const` para o objeto destino.

No caso do operador de atribuição retornar `void`, não será possível utilizar a seqüência de atribuição a = b = c. Exemplo:

```
void CMClasse::operator=( const CMClasse mc ) ;          // Não
CMClasse & CMClasse::operator=( const CMClasse mc ) ;    // também não
const CMClasse & CMClasse::operator=( const CMClasse mc ) ;   // OK
```

3.7 Sobrecarga de operadores (*overloading*)

Recom. 26: Utilize parcimoniosamente a sobrecarga de operadores.

Sobrecarga de operadores tem vantagens e desvantagens. Uma vantagem é a redução do volume de texto a ser redigido. Uma desvantagem é a dificuldade de entender operadores cuja semântica não seja óbvia com relação aos objetos envolvidos.

Regra 27: Caso um operador possua inverso, como em == e !=, ambos os operadores devem ser definidos.

Evite casos em que o operador "+" seja definido e o operador "-" não o seja.

3.8 Sobrecarga de funções

Regra 28: Todas as variações de sobrecarga de uma mesma função devem ter a mesma semântica.

No caso de um conjunto de funções que executam o mesmo tipo de operação e que diferem apenas no tipo de parâmetro, conserve sempre o nome da operação (*overloading*), porém assegure que o tipo retornado seja o mesmo em todas elas. Exemplo.

```
class String
{
   public:
      int Contem( const char    cParm  ) ;
      int Contem( const char*   pcParm ) ;
      int Contem( const String& szParm ) ;
} ;
```

3.9 Tipos de valores retornados por métodos

Regra 29: Métodos públicos não devem retornar referências não `const` para dados membro do objeto, tampouco devem retornar ponteiros para dados membro do objeto.

Regra 30: Métodos públicos não devem retornar referências ou ponteiros não `const` para dados externos ao objeto, a menos que o objeto compartilhe dados com outros objetos.

A permissão de acesso direto a membros privados de um objeto via um ponteiro, além de permitir atribuições imprevistas, é uma franca violação das regras de encapsulamento. Em adição, retornando ponteiros, pode-se criar situações em que áreas de dados são desalocadas múltiplas vezes, ou situações em que ponteiros apontam para áreas já desalocadas.

3.10 Herança

Recom. 31: Evite herança para relações do tipo *parts-of*.

Uma relação do tipo *parts-of* ocorre quando um objeto é composto de vários objetos. Nestes casos eles devem ser implementados como variáveis de instância e não como super classes daquele objeto. Esta última forma gera uma hierarquia de classes complexa e com pouca flexibilidade.

Recom. 32: Se uma classe derivada necessitar acesso a dados da classe base, use funções membro protegidas para tornar visíveis os dados.

Esta recomendação é consistente com a regra 3 que estabelece o encapsulamento total. Uma vantagem é que as propriedades dos dados não são visíveis nas classes derivadas e, portanto, podem ser modificadas. O pressuposto é que programadores de classes derivadas conhecem suficientemente bem as propriedades dos dados privados da classe base, podendo utilizá-los corretamente sem necessitar referenciá-los por nome. Reduz-se assim o acoplamento entre classes.

3.11 Conversão de tipos (*type cast*)

Regra 33: Nunca utilize conversão de tipos explícita entre objetos.

Exceção 34: Uma conversão de tipos explícita pode ser utilizada:

- quando a conversão implícita é incerta.
- quando se converte um espaço de dados anônimo (e.g. `void`) para um objeto ou estrutura (`struct`, ou `union`).

Conversão de tipos pode ser realizada de forma explícita ou de forma implícita, dependendo se é realizada por instrução do programador – *type casts* – ou por iniciativa do compilador. Conversão entre objetos de diferentes tipos podem gerar resultados inesperados, levar a código não portátil, e tornar difícil a localização das falhas causadoras de erros de execução. Veja exemplos no Anexo A.

Existem duas formas de estabelecer conversões implícitas. Ou o programador torna disponível uma função que realiza uma conversão, ou o compilador utiliza a definição padrão da linguagem para realizar a conversão. Ambas podem gerar problemas. Caso não encontre uma função que satisfaça os tipos dos parâmetros, C++ tenta converter os tipos. Caso mais de uma função de conversão for encontrada, ou mais de uma forma de converter for encontrada, ocorrerá um erro de compilação. Outro problema é a criação de objetos temporários durante o processo de conversão, este objeto torna-se então o parâmetro efetivamente passado e não o parâmetro atual original.

Regra 35: Nunca converta ponteiros para objetos de uma classe derivada, para ponteiros a objetos de uma classe base virtual.

É possível converter um ponteiro para um objeto de uma classe derivada em um ponteiro para um objeto de uma classe base. No entanto, a conversão inversa não é possível. Ou seja, a conversão de tipos entre classes não é reversível. Conseqüentemente, em uma classe virtual, a função virtual utilizada com relação a um ponteiro para um objeto da classe virtual base será sempre a função desta classe virtual base, independentemente do fato deste ponteiro ter sido gerado para um objeto de uma das classes derivadas. Esta é a razão para a proibição da conversão de ponteiros para classes base virtuais.

A classe virtual base poderia possuir uma função capaz de converter ponteiros para objetos desta classe, em ponteiros para objetos de classes derivadas. Por exemplo, podería-se guardar um identificador da classe base no objeto da classe virtual. Evidentemente esta função precisa ter conhecimento de todas as classes derivadas da classe virtual base. O uso deste tipo de função não é aconselhável por negar os princípios básicos da utilização de herança – desconhecimento dos herdeiros de uma classe base.

Regra 36: Nunca converta uma variável `const` para uma não `const`.

Regra 37: Nunca escreva código que dependa de funções que realizem alguma conversão implícita de tipos de dados.

Exceção 38: Em algumas ocasiões pode ser necessário redigir construtores com um único parâmetro, por exemplo ao interagir com código redigido em C.

3.12 Alocação de memória

Regra 39: Não use `malloc`, `realloc` ou `free`. Utilize somente `new` e `delete` para alocar e desalocar objetos.

`malloc`, `realloc` e `free` são utilizados para alocar dinamicamente memória no *heap*, porém não ativam construtores ou destrutores de objetos.

Recom. 40: Evite o uso de dados estáticos.

Dados estáticos criam diversos problemas, principalmente devido ao fato de poder levar a funções não reentrantes.

Regra 41: Utilize sempre "`[ ]`" ao desalocar vetores com o comando `delete`.

No caso de um vetor X de certo tipo T estar alocado, o comando `delete X` ocasionará a execução de um destrutor somente para o primeiro objeto do tipo T. A forma correta é `delete [] X`. Neste caso o destrutor será chamado para os todos os objetos efetivamente alocados.

Regra 42: Não aloque dados esperando que outra classe os desaloque.

Pode-se querer retornar um dado alocado como resultado de uma função, na esperança que o usuário desta função o desaloque ao terminar de utilizá-lo. No entanto, ninguém garante que o usuário não esqueça de desalocá-lo.

Exemplo:

```
String Func( const char * Parm )
{
   String * Temp = new String( Parm ) ;
   return * Temp ;
   // Temp jamais será desalocado e o usuário não poderá desalocar
   // Temp, pois uma cópia temporária é retornada
}
```

Recom. 43: Ao desalocar um espaço de dados, atribua um novo valor a todos os ponteiros que o referenciem.

Recom. 44: Evite ter mais de um ponteiro para um mesmo espaço de dados.

Ponteiros que apontam para áreas desalocadas devem receber valor `P_NIL` ou um novo valor, para evitar o acesso à esta área. Evidentemente, isto pode ser difícil de ser realizado, uma vez que nem sempre é possível saber quais são todos os ponteiros que apontam para um mesmo espaço de dados.

Anexo A. Exemplos de conversão de tipos

Exemplo 1. Utilização de construtores que contém somente um elemento como argumento.

É gerada um conversão implícita de tipo que pode, em certos casos, gerar erro no programa.

```
class String
{
   public:
      String( int length ); //   um construtor
} ;

// Função que tem um argumento do tipo String
   void aFunc( const String & aString) ;
...
//
   int x = 100;
// chamando aFunc com um inteiro
   aFunc( x );    //   CONVERSÃO implícita => aFunc( String( x ) )
```

Exemplo 2. Uma função sobrecarregada que ocasionará um erro de compilação.

No caso de não existir uma função que contenha exatamente o mesmo tipo de argumento, o compilador tenta a conversão para as funções que, talvez, tenham o mesmo tipo. Se mais de uma função for encontrada, ocorrerá um erro de compilação.

```
// Arquivo file1.hh
class String
{
  .public:
      String( char* cp ) ;    //  um construtor
      operator const char * () const;
         // operador de conversão para const char *
} ;

   void aFunc( const String& aString ) ;

// Arquivo file2.hh
class Word
{
   public:
      Word( char* cp );      //   um construtor
};
//   SOBRECARREGANDO a função
   void aFunc( const Word& aWord ) ;

//   Arquivo main.cc
main()
{
   aFunc( "hello");          // ERRO NA CONVERSÃO DO TIPO
   // a chamada de função aFunc( "hello") é ambígua pois ,
   // void aFunc( const String& ) e void aFunc( const Word& )
   // são conversões aceitas
};
```

Exemplo 3. Conversão implícita que gera um resultado inesperado

```
void Troque( int& x, int& y )
{
   int temp = x;
   x = y;
   y = temp;
```

```
}

int          Num   = 10 ;
unsigned int Outro = 20 ;
Troque( Num , Outro ) ;

   //   A execução será a seguinte
   // int tmp = int( Outro )     conversão implícita
   // Troque( aNumber, tmp )     Outro não será modificado
```

Exemplo 4. Conversão de um ponteiro para um classe derivada para um ponteiro de uma superclasse virtual.

E possível converter um ponteiro de uma instância de uma classe derivada de uma classe virtual para um ponteiro de um objeto da classe virtual. Esta conversão não é recomendável, porque ela é irreversível.

```
class VirtualClass
{
   public:
      virtual class Derived * asDerived() = 0;
};

class Derived: virtual public VirtualClass
{
   public:
      virtual Derived * asDerived();
};

Derived * Derived::asDerived()
{
   return ;
}

void main()
{
   Derived   d;
   Derived * dp = 0;
   VirtualClass * vp = ( VirtualClass * ) &d;

   dp = (Derived*) vp;    //    ERRO !

//conversão de um ponteiro da superclasse virtual

   dp = vp-> asDerived();    //    OK !
}
```

Anexo B. Utilização de construtores e destrutores

Exemplo 1. Classe com construtor de cópia e destrutor bem definidos

```
class String
{
   public:
      String( const char * cp = "" ) ;  // construtor
      String( const String & sp ) ;     // construtor de cópia
      ~String();                        // destrutor
   private:
      char* sp;
};

// construtor
String::String( const char* cp ) : sp( new char[ strlen( cp ) + 1])
{
   strcpy( sp, cp );
}

String::String( const String & stringA ) :
                  sp( new char[ strlen( stringA.sp )])
{
   strcpy( sp, stringA.sp ) ;
}

String::~String()       // destrutor
{
   delete sp;
}

void main()
{
   String w1;
   String w2 = w1;
   // a chamada da execução da cópia será a seguinte
   // String::String( const String& )
}
```

Exemplo 2. Uma classe que não contém a definição de um construtor de cópia

A inexistência de um construtor de cópia ocasionará duas chamadas do destrutor para o mesmo objeto que está no topo da pilha de parâmetros o que resultará num erro de execução.

```
class String
{
   public:
      String( const char* cp = "" );    // construtor
     ~String();                         // destrutor
  private:
    char* sp;
} ;

String::String( const char* cp ) : sp( new char[ strlen( cp ) + 1 ])
//   construtor
{
   strcpy( sp, cp );
}

String::~String()
//   destrutor
{
   delete sp ;
```

```
}

void main()
{
   String w1 ;
   String w2 = w1 ;
   // CUIDADO !!  na cópia bit a bit de w1::sp o destrutor será
   // chamado duas vezes
   //    Primeiro quando  w1 for destruído e
   //    novamente quando w2 for destruído
```

Exemplo 3. Uma classe com inexistência de um destrutor virtual

Este exemplo mostra a conseqüência da não obediência à Regra 19.

```
class Fruit
{
   public:
      ~Fruit();
};

class Apple : public Fruit
{
  public:
    ~Apple();    //.destrutor
};

class FruitBasket

  public:
    FruitBasket( )  ;      // Cria uma FruitBasket
    ~FruitBasket( ) ;      //.apaga todas as Fruits
    void add( Fruit* ) ;   // Adiciona uma fruit
  private:
    Fruit* storage[ 42 ]; // máximo número de Fruits
    int numberOfStoredFruits
};

void FruitBasket::add( Fruit* fp )
{
// aloca um ponteiro para um objeto Fruit
   storage[ numberOfStoredFruits ] = fp ;
   numberOfStoredFruits++ ;
}

FruitBasket::FruitBasket( ) : numberOfStoredFruits( 0 )
{
}
FruitBasket::~FruitBasket()
{
   while ( numberOfStoredFruits > 0 )
   {
      numberOfStoredFruits-- ;
      delete storage[ numberOfStoredFruits ] ;
         //SOMENTE  Fruit::~Fruit será chamado
   }
```

Bibliografia

Diversas das regras aqui apresentadas, bem como a estrutura das presentes regras, seguem de perto o texto:

Henricson, M; Nyquist, E; *Programming in C++: Rules and Recomendations*; Ellemtel Telecommunications Systems Laboratories; Älvsjö, Suécia; 1992

Uma boa apresentação dos conceitos de orientação a objetos, tais como *encapsulamento*, *herança* e *overloading*, é encontrada em:

Khoshafian, Setrag; *Object Orientation, Concepts, Languages, Databases, User Interfaces*; Wiley Professional Computing; 1990

Uma boa referência para orientação a objetos utilizando C++ é

Montenegro, F.; Pacheco, R.; *Orientação a Objetos em C++*; Editora Ciência Moderna; 1994

A referência padrão para a linguagem C++ é:

Stroustrup, B.; *The C++ Programming Language, Second Edition*; Addison Wesley; 1991

As seguintes referências descrevem técnicas de análise e projeto orientadas a objetos utilizando técnicas estruturadas:

Booch,:G.; *Object-Oriented Design with Applications*; Benjamim/Cummings;1991

Rumbaugh, J.; Blaha, B.; Premerlani, W.; Eddy, F.; Lorensen, W.; *Object-Oriented Modeling and Design*; Prentice Hall; 1991

A seguinte referência descreve técnicas de análise e projeto orientadas a objetos utilizando técnicas funcionais:

Wirfs-Brock, R.; Wilkerson, B.; Wiener, L.; *Designing Object-Oriented Software*; Prentice Hall; 1990