
# PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 25/96

# PG-02 Regras e Recomendações para o Uso de Constantes Simbólicas em C e C++

## - Versão 1.00 -

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 25/96
Editor: Carlos J. P. Lucena Setembro, 1996

# PG-02 Regras e Recomendações para o Uso de Constantes Simbólicas em C e C++ *

- Versão 1.00 -

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-02 Regras e recomendações para o uso de constantes simbólicas em C e C++

Versão 1.00

**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 25/96*

## Resumo

Neste documento estabelecemos regras e recomendações para o uso de constantes em programas redigidos em C ou C++.

Como resultado desta norma espera-se:

- facilitar a manutenção de programas, sempre que esta manutenção afete valores constantes.
- facilitar o entendimento do significado de fragmentos de código fonte contendo constantes, sem necessitar recorrer a documentação adicional que explique o significado dos valores.

**Palavras chave**: constantes, constantes simbólicas, manutenibilidade, garantia de qualidade.

## Abstract

In this document we establish rules and recommendations for using constants in C or C++ programs.

When adopting this standard it is expected that:

- the maintenance of programs is simplified when involving constant values.
- constants can be easily and correctly remembered when creating or modifying a program, without needing to search extensive documentation.
- programmers can easily and rapidly understand and make correct changes to values and use of constants in existing programs.

**Keywords**: constants, symbolic constants, maintainability, quality assurance.

---

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

# Histórico de evolução

PG-02 Convenções para o uso de constantes simbólicas em C e C++
Versão 1.00

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

Arquivo: Const01
Editado: 11 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++

---

## 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 02/jan/1997

Descrição de evolução

Descrição da retroação

# Créditos

Revisores

| | | |
|---|---|---|
| André Derraik | (TeCGraf PUC-Rio) | Versão 1.0 |
| 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, Windows, Visual C++ são marcas registradas da Microsoft Corp.

# Sumário

# 1. Objetivo

A presente norma tem por objetivo:

- uniformizar o uso de constantes em programas redigidos em C ou C++.

Como resultado desta norma espera-se:

- facilitar a manutenção de programas, sempre que esta manutenção afete valores constantes.
- facilitar o entendimento do significado de fragmentos de código fonte contendo constantes, sem necessitar recorrer a documentação adicional que explique o significado dos valores.

**Observação:**

A escolha de nomes para constantes simbólicas é normalizada na norma:

PG-01 Regras e recomendações para a escolha de nomes de elementos em programas C e C++

# 2. Motivação

Programas contém diversas constantes. São exemplos: a dimensão declarada de um vetor, a dimensão declarada para uma variável do tipo *string*, o valor recebido ao selecionar um item de menu, e o valor de constantes matemáticas tais como $\pi$ ou $e$.

Freqüentemente, uma mesma constante é utilizada em diversos lugares num mesmo código fonte ou até em diferentes arquivos de código fonte. Finalmente, um mesmo valor pode ter significados diferentes em diferentes lugares. Por exemplo, o valor 256, em um determinado contexto pode representar o tamanho máximo de um *string*, em outro pode representar a largura de uma janela medida em pixels.

A inclusão de constantes como valores literais (por exemplo 3,1415, 100, "faltam parâmetros") diretamente no código fonte dos programas dificulta a manutenção destes programas. Para realizar qualquer modificação que afete uma destas constantes, é preciso localizar todos os lugares em que ela é utilizada. Além disso, constantes com diferentes significados (semânticas) podem ter o mesmo valor literal, dificultando, assim, a correta alteração do programa. Ou seja, ao alterar um programa não basta encontrar a seqüência de caracteres que designa o valor literal da constante, é necessário também identificar qual o significado do valor literal no contexto em que ela está sendo utilizada. Por exemplo: o tamanho do nome de um arquivo e o tamanho do nome de um "path" podem ter sido fixados ambos em 47. Mais tarde resolve-se mudar o tamanho destes nomes, por exemplo, para 80 para nomes completos de arquivo, 12 para nomes de arquivos sem diretório, e 64 para nomes de diretórios. Utilizar uma operação do tipo *"troca tudo"* não resolve pois, dependendo do lugar no código fonte, o valor 47 deverá ser trocado para um desses 3 diferentes valores.

Diferentes plataformas (processadores, compiladores, bibliotecas, etc.) podem requerer diferentes definições para constantes matemáticas e para constantes de controle. Por exemplo, pode-se querer utilizar precisão dupla em algumas plataformas e precisão estendida em outras. Embutir as constantes literais no código traz consigo a desvantagem de necessitar manter diversos programas, um para cada plataforma.

Finalmente, ao se encontrar uma constante literal no código fonte de um programa, nem sempre se é capaz de identificar o seu significado examinando exclusivamente o texto próximo ao local onde está redigida a constante. Por exemplo, a maioria dos sistemas gráficos tipo *Windows* sinaliza o item de menu escolhido por meio de um valor numérico. O valor retornado corresponde ao valor escolhido pelo programador que definiu o menu. Para se saber agora se está correta a constante utilizada no programa que processa as seleções, precisa-se consultar a definição do menu. Usualmente a definição do menu se encontra em um arquivo diferente do arquivo de código fonte sendo editado. Pior, se alguém alterar a definição deste menu, o programa que processa as seleções poderá deixar de funcionar.

Em resumo, o uso de constantes literais diretamente no código de um programa contribui para uma perda de manutenibilidade, de portabilidade e de legibilidade deste programa. Este problema é facilmente eliminado através do uso sistemático de constantes simbólicas. Nas linguagens C e C++ constantes simbólicas são definidas por intermédio de declarações `#define`, `enum`, e `const <variável inicializada>`.

# 3. Terminologia usada

*Módulo* é uma unidade de compilação.

*Módulo servidor* é o módulo sendo criado, examinado ou mantido. No contexto desta norma é o módulo no qual se está trabalhando no momento.

*Módulo cliente* é um módulo que utiliza os serviços prestados pelo módulo servidor.

*Módulo de definição* é o conjunto de declarações tornadas públicas pelo módulo servidor. Estas declarações são necessárias para que os possíveis módulos clientes possam ser corretamente compilados, bem como para assegurar a consistência das interfaces entre os módulos.

*Módulo de implementação* contém o código fonte do módulo servidor. Este código será submetido ao compilador para gerar o módulo objeto correspondente ao módulo servidor. O módulo de implementação importa os módulos de definição de todos os outros módulos servidores de que necessita, inclusive o módulo de definição próprio, bem como arquivos contendo as tabelas de constantes que porventura use.

# 4. Definição da norma

Regra 1: Todas as constantes utilizadas em um módulo devem ser simbólicas.

Regra 2: As declarações de constantes simbólicas devem ser agregadas nos seguintes domínios:

1. *constantes públicas*, definidas no módulo de definição.
2. *constantes encapsuladas*, definidas no módulo de implementação.
3. *constantes tabulares*, definidas em arquivos de inclusão.

Regra 3: O nome simbólico da constante deve ser redigido segundo a norma *PG-01 Regras e recomendações para a escolha de nomes em programas C e C++.*

Exceção 4: Constantes geradas por ferramentas, ou sujeitas a normas de organização de programas especializadas e publicadas, seguem as convenções de caixa alta/baixa destas ferramentas ou normas.

Regra 5: Todos os arquivos de inclusão devem ter o seguinte formato[2]:

```
// Comentário cabeçalho do arquivo

#ifndef _NomeArquivo_HHH
#define _NomeArquivo_HHH

   Inserir nesta posição o texto a incluir

#endif

// Comentário de término do arquivo
```

onde

`NomeArquivo` corresponde ao nome do arquivo de inclusão.

`HHH` é o nome do tipo do arquivo (`H` para módulos de definição C, `HPP` para módulos de definição C++ e `INC` para tabelas de definições de constantes)

Recom. 6: Os arquivos de inclusão devem agregar constantes tabulares de uma única natureza.

Recom. 7: Quando arquivos de inclusão agregam constantes relacionadas com dois ou mais módulos, todos os nomes devem seguir as regras de nomes públicos.

Recom. 8: Quando arquivos de inclusão agregarem constantes relacionadas com um único módulo, todos os nomes do arquivo de inclusão devem seguir as regras de nomes encapsulados.

Recom. 9: Constantes relacionadas a um mesmo conceito devem ser definidas em uma declaração `enum`. Constantes individuais devem ser declaradas ou através de uma declaração `#define`, ou através de uma declaração `const`.

Recom. 10: Prefira declarar constantes na forma `const <declaração de variável inicializada>`.

*Constantes públicas* são constantes definidas em um determinado módulo servidor e que precisam ser tornadas visíveis para pelo menos um dos módulos cliente deste módulo servidor. Todas as constantes públicas devem ser definidas no *módulo de definição* deste módulo servidor. Somente valores que serão efetivamente utilizados por algum módulo cliente deverão figurar como constantes públicas.

*Constantes encapsuladas* são constantes definidas em um determinado módulo servidor e que não precisam ser visíveis para qualquer um dos módulos cliente deste módulo servidor. Todas as constantes encapsuladas devem ser definidas no *módulo de implementação* deste módulo servidor.

*Constantes tabulares* são constantes comuns a um ou mais módulos, processadores e/ou plataformas. Ou seja, são constantes que não podem ser encapsuladas, mas que também não são de *"propriedade"* do módulo servidor. Constantes tabulares são definidas em um arquivo específico que contém somente definições de constantes

---

[2] Linhas de comentários serão redigidos nesta norma segundo as convenções adotadas em C++. No caso de programas redigidos em C, comentários devem ser redigidos na forma `/* ... */`.

simbólicas. Os arquivos de constantes tabulares devem conter constantes de uma mesma natureza. Por exemplo: constantes que identificam as seleções de um menu devem ser armazenadas em um arquivo diferente das constantes que identificam os campos de um diálogo.

Algumas ferramentas de apoio ao desenvolvimento não permitem a adoção da recomendação relativa a constantes tabulares. Neste caso adote a convenção de nomes públicos para as constantes. e utilize a identificação do domínio para discriminar o módulo relativo ao qual a constante está sendo definida.

Arquivos de inclusão podem ser referenciados em outros arquivos de inclusão. Conseqüentemente é possível que um mesmo arquivo de inclusão seja incluído mais de uma vez ao compilar determinado módulo. Esta multiplicidade de inclusão leva a erros de compilação. Para evitar estes erros, o arquivo de inclusão deve estar contido em um segmento de compilação condicional tal como definido na Regra 5.

Constantes são definidas através de declarações `#define`, declarações `enum` e através de declarações de variáveis inicializadas precedidas do declarador `const`. Em C valores individuais devem ser definidos por meio de `#define`. Em C++ procure utilizar a forma `const`, uma vez que esta forma assegura controle de tipos. São exemplos de valores individuais: dimensão de vetor, valor específico com significado próprio tais como o ponteiro **NULL**. Exemplos de declarações:

em C:
```
#define SZ_MEU_STRING "Esta é a constante" ;
```

em C++
```
const char szMeuString[ ] = "Esta é a constante" ;
```

A desvantagem de se utilizar constantes `const` é um custo de execução ligeiramente maior. Além disso, restrições sintáticas podem impedir a declaração de constantes simbólicas na forma `const`. Por exemplo, em C constantes que definem dimensões ou outras propriedade de variáveis, não podem ser declaradas na forma `const`.

**Obs.:** Ao compilar gerando código de 16 bits, diversos compiladores armazenam os valores de constantes estáticas no segmento de dados, consumindo um espaço que já é escasso. Nestes casos pode, a primeira vista, ser vantagem utilizar extensões não portáteis da linguagem para forçar o armazenamento do valor em outro segmento. Por exemplo: ao utilizar o compilador Visual C++ gerando código de 16 bits, constantes *string* serão armazenadas no segmento de código ao declarar de forma semelhante a:

```
static char BASED_CODE szMeuString[ ] = "Esta é a constante" ;
```

No entanto, este tipo de declaração deve ser evitado, uma vez que compromete a portatilidade do programa. No exemplo na seção a seguir é ilustrado como este problema pode ser resolvido mantendo portatilidade.

Valores que definem um conjunto completo de possíveis valores devem ser definidos por meio de uma declaração `enum`. São exemplos: conjunto de condições de retorno de uma função, conjunto de condições de controle.

# 5. Exemplos

## 5.1 Exemplo: uso de constantes públicas e encapsuladas

Considere um módulo que implementa as funções de manipulação de um conjunto de tabelas de símbolos. Um dos requisitos deste módulo é a proibição de funções cancelarem a execução caso recebam parâmetros errados, ou caso encontrem condições de funcionamento ou uso anormais. Como conseqüência deste requisito, todas as funções deverão retornar alguma condição de retorno, indicando se operaram corretamente ou não. Consideremos em especial a função `TS_InserirSimbolo`. Esta função tem a seguinte assinatura[3]:

`TS_InserirSimbolo` : *IdTabela*, *Símbolo*, *Tabela[ IdTabela ]* ->
*IdSimbolo*, *CondRet*, *Tabela[ IdTabela ]*

onde:

*IdTabela* identifica a tabela na qual será inserido o símbolo. Esta tabela deve existir para se poder inserir um símbolo.

*Símbolo* uma cadeia de zero ou mais caracteres que representa o símbolo.

*Tabela[ IdTabela ]* é o valor da tabela *IdTabela* considerando todos os símbolos nela inseridos. Para utilizar `TS_InserirSimbolo` não é necessário saber como esta estrutura de dados é organizada, porém é necessário saber que existe e quais as regras que satisfaz. Para cada símbolo conhecido, a tabela contém um par *<Símbolo, IdSimbolo>*, associando o *Símbolo* a um *IdSimbolo* e assegurando uma relação um para um.

*IdSimbolo* é um identificador único em correspondência um para um com um dos símbolos contido em *Tabela[ IdTabela ]*. Caso o símbolo a inserir ainda não exista na tabela, será criado um novo *IdSimbolo* e o par *<Símbolo, IdSimbolo>* será incorporado à tabela. Caso o símbolo já exista na tabela, esta não será alterada. O valor retornado é sempre o *IdSimbolo* associado ao símbolo existente na tabela.

*CondRet* é a condição de retorno da função.

Esta função pode falhar na execução devido a erros de dados, por exemplo se o valor de *IdTabela* fornecido é o de uma tabela inexistente, ou devido ao esgotamento de recursos, por exemplo a memória livre disponível é insuficiente para acomodar o par *<Símbolo, IdSimbolo>*. Em adição, por uma decisão de projeto, o tamanho do símbolo pode ser limitado. Conseqüentemente, a função pode falhar em virtude de se tentar armazenar um símbolo grande demais para a implementação.

Temos aqui várias constantes:

`TS_ID_SIMB_NULL` significa: "símbolo não definido".

`TS_ID_SIMB_ERRO` significa: "função não gerou IdSimbolo devido a uma anomalia de uso".

Ambas as constantes são valores especiais de `IdSimbolo`, porém não esgotam a totalidade de valores possíveis. Além disso são valores públicos, uma vez que os módulos clientes do módulo tabela de símbolos necessitam conhecer estes valores especiais. Portanto o módulo de definição do módulo tabela de símbolos deve conter o seguinte fragmento de código (ver norma *PG-01 Regras e recomendações para a escolha de nomes em programas C e C++.* para a escolha de nomes):

```
typedef unsigned short TS_tpidSimb ;
#define TS_ID_SIMB_NULL   0xFFFF
#define TS_ID_SIMB_ERRO   0xFFFE
```

---

[3] Utilizamos o termo assinatura no sentido *funcional* e não no sentido de *protótipo* de função como é costumeiro em programação orientada a objetos. Ou seja, são *dados de entrada* todos os parâmetros e valores globais públicos ou encapsulados necessários para o correto funcionamento da função. São *dados de saída* o valor retornado, valores retornados por intermédio de parâmetros e valores retornados em variáveis globais públicas ou encapsuladas. Em outras palavras, a assinatura identifica todos os dados necessários e todos os resultados produzidos, independentemente da forma de se estabelecer a comunicação destes dados e resultados.

A declaração typedef do tipo TS_tpidSimb assegura que o tipo físico utilizado pelo módulo tabela de símbolos para representar um *IdSimbolo*, no caso um unsigned short, possa ser mudado sem que os módulos cliente precisem ser modificados, bastando que sejam recompilados.

Em C++ pode-se utilizar uma sucessão de declarações na forma:

```
typedef unsigned short TS_tpidSimb ;
const TS_tpidSimb TS_idSimbNull = 0xFFFF ;
const TS_tpidSimb TS_idSimbErro = 0xFFFE ;
```

A função retorna a condição de retorno *CondRet*. Note que a assinatura não determina se o retorno se dá como um valor retornado ou como uma variável global contendo o valor retornado. Considerando o conjunto de funções do módulo tabela de símbolos, existe um conjunto finito e pequeno de possíveis condições de retorno. Conseqüentemente, o módulo de definição do módulo tabela de símbolos deve conter o fragmento de código:

```
enum TS_fCondRet
{
   TS_CondRetOK ,
   TS_CondRetErroParm ,
   TS_CondRetMemInsuficiente ,
   TS_CondRetSimbGrandeDemais ,
   TS_CondRetSimbDemais
} ;
```

O tamanho máximo do símbolo precisa ser de conhecimento do módulo cliente, pois este deverá ser capaz de truncar símbolos grandes para o tamanho permitido. Portanto, o módulo de definição do módulo tabela de símbolos em C deve conter as declarações:

```
#define   TS_LIM_SIMB    32
#define   TS_DIM_SIMB    ( TS_LIM_SIMB + 1 )
```

EM C++ a declaração contida no módulo de definição seria:

```
const unsigned short TS_LimSimb = 32 ;
const unsigned short TS_DimSimb = TS_LimSimb + 1 ;
```

Finalmente, o número máximo de símbolos permitidos pode ser limitado, por exemplo se a tabela for implementada sob a forma de um vetor. Porém, este valor não é de interesse de nenhum módulo cliente. Portanto o módulo de implementação C deve conter uma declaração semelhante a:

```
#define   ST_DIM_TAB_SIMB    1000
```

Já o módulo de implementação C++ deve conter a declaração:

```
const unsigned short st_DimTabSimb = 1000 ;
```

Conforme a norma *PG-01 Regras e recomendações para a escolha de nomes em programas C e C++*, o nome da constante global encapsulada deve ter o prefixo ST_. Idealmente esta declaração é inserida no código, próximo ao ponto onde é declarado o vetor. Por exemplo, em C declare::

```
#define   ST_DIM_TAB_SIMB   1000
st_tpSimb st_vtSimbTabSimb[ ST_DIM_TAB_SIMB ] ;
```

Em C++ utilize a declaração:

```
const unsigned short         ST_DimTabSimb = 1000 ;
st_tpSimb ST_vtSimbTabSimb[ ST_DimTabSimb ] ;
```

## 5.2 Exemplo: uso de constantes tabulares

Este exemplo é explicitamente dirigido a aplicações Windows utilizando o ferramental tornado disponível para esta plataforma. Embora o exemplo seja dirigido a uma plataforma específica, a estrutura do código ilustrada é idêntica à estrutura encontrada ao desenvolver programas para outras plataformas ou utilizando outros conjuntos de ferramentas de apoio ao desenvolvimento.

Considere o uso de menus em Windows. O exemplo a seguir ilustra uma declaração típica de um menu:

```
IDM_MENU MENU
{
   POPUP "&File"
   {
      MENUITEM "&New Segment...",      IDM_SEG_CRIAR
      MENUITEM "&Open Segment...",     IDM_SEG_ABRIR
      MENUITEM "&Close Segment...",    IDM_SEG_FECHAR
      MENUITEM SEPARATOR
      MENUITEM "&Clear log",           IDM_LOG_LIMPAR
   }
   POPUP "Sho&w"
   {
      MENUITEM "&List page..." ,       IDM_LISTA_SELEC
      MENUITEM "&Virtual page...",     IDM_PAG_SELEC
      MENUITEM "&Known segments",      IDM_SEG_EXIBIR
      MENUITEM "&Frames" ,             IDM_PORT_EXIBIR
   }
   POPUP "\a&Help"
   {
      MENUITEM "About &Debug...",      IDM_AUX_ABOUT
   }
}
```

Uma declaração tipo `MENUITEM` define um *string* que será exibido no vídeo e um valor que será transmitido pelo Windows caso este item seja selecionado pelo usuário. O valor transmitido é tipicamente um `unsigned short`. Portanto, ao invés de utilizar constantes simbólicas, poderíamos ter utilizado números, por exemplo os números 100, 101, 102, etc. Na aplicação que processa as mensagens recebidas do Windows, teremos que redigir uma rotina para cada uma destas possíveis seleções. Se utilizássemos um número, primeiro não saberíamos verificar, sem utilizar documentação complementar, se cada rotina é ativada pela opção de menu correta. Segundo, o programa deixará de funcionar se um programador inadvertidamente modificar as declarações acima. Utilizando um arquivo de constantes tabulares resolvemos este problema. No caso do Microsoft *Windows*, este arquivo é utilizado pelo *resource compiler* para gerar a representação interna do menu, e é utilizado pelo compilador ao compilar os módulos que processam as seleções. Asseguram-se, assim, a unicidade e a consistência das interfaces entre o processador de menus contido em *Windows* e o programa que executa as seleções do usuário.

Supondo que a identificação do módulo que processará as seleções deste menu seja ABC, o nome do arquivo de definições será: ABCIDM.INC. E o seu conteúdo será:

```
// Comentário cabeçalho do arquivo

#ifndef _ABCIDM_INC
#define _ABCIDM_INC
   #define   IDM_MENU                100
   #define   IDM_SEG_CRIAR           101
   #define   IDM_SEG_ABRIR           102
   #define   IDM_SEG_FECHAR          103
   #define   IDM_LOG_LIMPAR          104
   #define   IDM_LISTA_SELEC         105
   #define   IDM_PAG_SELEC           106
   #define   IDM_SEG_EXIBIR          107
   #define   IDM_PORT_EXIBIR         108
   #define   IDM_AUX_ABOUT           109
#endif

// Comentário de término do arquivo
```