PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 24/96

PG-01 Regras e Recomendações para a Escolha dos Nomes de Elementos em Programas C e 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 24/96

Editor: Carlos J. P. Lucena Setembro, 1996

PG-01 Regras e Recomendações para a Escolha dos Nomes de Elementos em Programas C e 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-01 Regras e recomendações para a escolha dos nomes de elementos em programas C e 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 24/96

Resumo

Neste documento estabelecemos regras e recomendações para a escolha de nomes de elementos de programas redigidos em C ou em C++. É definida a estrutura padrão {**<Produto>|<Domínio>**}_**<Prefixo tipo><Tema><Sufixo Tipo>**. Esta estrutura contempla todas as categorias de nomes e formas de uso de nomes em programas C e C++.

Como resultado da adoção destas regras e recomendações espera-se que:

- a criação de novos nomes seja o mais independente possível do redator do programa.
- seja fácil lembrar e entender os nomes dos elementos já declarados ao criar ou alterar um programa, sem precisar recorrer a documentação complementar.
- seja fácil assegurar que o correto uso dos elementos do programa.
- seja fácil localizar o documento e a correspondente seção contendo a especificação e a declaração completa do elemento denominado.
- programadores possam rápida e facilmente entender e corretamente modificar programas redigidos por outros.

Palavras chave: engenharia de software, nomes, nomes em APIs, nomes públicos, nomes encapsulados, nomes locais, garantia de qualidade.

Abstract

In this document we establish rules and recommendations for choosing names of program elements. The standard structure of a name is **{<Product>|<Domain>}_<Type prefix><Theme><Type suffix>**. This structure is used for all categories and usage modes of names in C or C++ programs.

When adopting this standard it is expected that:

- the creation of names is as developer independent as possible.
- names in use can be easily and correctly remembered when creating or modifying a program, without needing to search extensive documentation.
- names can easily be shown to be correctly used wherever they occur.
- documentation defining names can easily be located.
- programmers can easily and rapidly understand and make correct changes to existing programs.

Keywords: encapsulated names, local names, names in APIs, public names, quality assurance, software engineering.

---

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

Histórico de evolução

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

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

Arquivo: Nmscpp01
Editado: 16 setembro, 1996
Impresso: 16 setembro, 1996

Documentos correlatos

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
redação mais precisa de alguns exemplos
eliminação de erros de indicador (valor incorreto associado ao indicador)

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

Descrição de evolução

Descrição da retroação

Créditos

Revisores

André Derraik	(TeCGraf PUC-Rio)	Versão 1.0
Claudio de Oliveira	(PUC-PR)	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

Itautec Philco
CNPq
CENPES Petrobrás

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 objetivos:

- uniformizar a forma de escolher nomes ao desenvolver ou manter programas redigidos em C ou em C++.
- poder ser aplicada a qualquer categoria de elemento, tais como constantes, variáveis, tipos declarados, funções, métodos e classes.
- poder ser aplicada a qualquer gênero de programa, tais como: aplicativos, servidores, bibliotecas e módulos individuais.

Como resultado desta norma espera-se que:

- seja fácil lembrar os nomes dos elementos já declarados ao redigir ou alterar um programa.
- seja fácil entender corretamente os nomes dos elementos contidos em trechos de código sendo lido, sem precisar recorrer a documentação complementar.
- seja fácil assegurar que os elementos do programa possuam tipos consistentes com o contexto em cada ponto em que estes elementos forem utilizados.
- seja fácil acompanhar corretamente o comportamento do programa ao utilizar ferramentas de apoio à depuração.
- seja fácil localizar o documento e a correspondente seção contendo a especificação e a declaração completa do nome.
- programadores possam rápida e facilmente entender e corretamente modificar programas redigidos por outros.

2. Motivação

O código fonte de um programa é lido, revisado e alterado por diferentes pessoas, em diferentes épocas e realizando diferentes tarefas. Poucas vezes o código fonte desses programas será processado por compiladores. Ou seja, o código fonte é redigido primordialmente para pessoas lerem e não para compiladores processarem. A cada vez que se lê um programa, é necessário determinar o significado exato de cada elemento encontrado no código fonte. Quanto mais precisa e completamente os nomes dos elementos sugerirem o seu significado e sua correta forma de uso, menos esforço será gasto para se compreender completa e corretamente o significado do programa. A má escolha dos nomes é uma das principais causas da dificuldade de compreensão do código fonte de um programa. Nomes pouco elucidativos obrigam o leitor a recorrer a documentação complementar. Infelizmente, corre-se o risco do leitor não fazê-lo, por achar que reconheceu ou se lembrou do significado dos nomes. Mesmo que o leitor procure a documentação complementar, nomes mal escolhidos induzem o leitor a fazer falsas suposições sobre seus significados.

Nomes são símbolos que identificam os elementos do programa. Esses elementos podem ser, entre outros: *constantes*, *variáveis*, *tipos*, *funções*, *métodos* e *classes*. O número de diferentes nomes utilizados em um único módulo pode chegar a algumas centenas. Considerando todos os módulos que constituem um programa de tamanho médio, podem estar em uso alguns milhares de diferentes elementos, cada qual com o seu nome. Finalmente, em uma empresa podem estar em uso centenas de milhares de diferentes nomes, considerando todos os elementos encontrados em todos os módulos desenvolvidos e mantidos nesta empresa. Memorizar corretamente o significado de todos nomes, é certamente uma tarefa demorada se não impossível. Portanto, ao ler o texto fonte de um programa, é desejável que se consiga determinar precisamente o *significado*, o *local da declaração* e as *propriedades computacionais* de cada nome encontrado, sem precisar passar por um extenso período de memorização, e sem precisar consultar documentação complementar tais como listas de declarações de variáveis, de constantes ou de funções. Deseja-se, também, assegurar que não ocorram conflitos de nomes contidos em diferentes módulos, sem que os desenvolvedores necessitem trocar informações ao criarem um novo nome público.

Estas propriedades tornam fácil:

- lembrar corretamente dos nomes dos elementos já declarados ao redigir ou alterar um módulo.
- escolher bons nomes novos ao criar ou manter programas.
- entender corretamente um trecho de código sendo lido, sem precisar recorrer a documentação complementar.
- assegurar que os elementos do programa possuam tipos consistentes com o contexto em que estão sendo utilizados.
- argumentar a correção de trechos de programa.
- acompanhar corretamente o comportamento do programa ao utilizar ferramentas de apoio à depuração.
- localizar a especificação e a declaração completa do nome.

Espera-se maior rapidez ao criar, avaliar e manter programas, em virtude da significativa redução do número de erros de programação e de manutenção, quando comparado com programação realizada sem utilizar convenções para a escolha de nomes. Cabe salientar que não se deseja eliminar os documentos (usualmente comentários) que descrevem os elementos contidos em um programa. O que se deseja é reduzir a freqüência com que se necessita recorrer a esta documentação, sem contudo por em risco o correto entendimento e a corretude[2] do módulo sendo criado ou mantido.

Nomes escolhidos segundo a presente norma tendem a ser longos. Experimentos realizados na década de 70 mostraram que o tempo requerido para redigir nomes longos e mnemônicos é virtualmente igual ao necessitado para redigir nomes curtos. Na realidade, o tempo requerido para programar, ler e manter, depende muito mais do tempo requerido para se compreender um nome, do que do tempo gasto ao digitá-lo. Os mesmos estudos mostraram que o número de erros cometidos, tais como seleção do nome errado e mistura de tipos incompatíveis, é muito maior quando se usa nomes pouco sugestivos.

---

[2] *Corretude: [correto + -(t)ude] S.f. propriedade (qualidade) de estar correto.* Preferimos utilizar um neologismo ao invés do termo *correção* para evitar a freqüente ambigüidade entre as acepções "corrigir" (tornar correto) e "estar correto".

3. Conceitos

Neste capítulo introduziremos alguns conceitos necessários para o entendimento da presente norma.

3.1 Tipos e controle de tipos

Uma preocupação básica da presente norma é assegurar que cada nome reflita corretamente o seu *tipo computacional* e o seu *tipo semântico*. A seguir explicaremos estes conceitos.

Do ponto de vista de linguagens de programação, tipos correspondem a propriedades computacionais, tais como *inteiro*, *flutuante*, *string*, *ponteiro*, *estrutura* (`struct`, `union`), e *classe*. Tais tipos chamamos de *tipos computacionais*. Os compiladores modernos controlam o correto uso de tipos computacionais, gerando mensagens de erro ou de advertência para todas as expressões que misturem tipos de forma inapropriada, como, por exemplo, atribuições de valores do tipo *string* a valores do tipo ponto flutuante. Infelizmente, algumas linguagens, entre elas C e C++, permitem misturas duvidosas de tipos. Por exemplo, em C e em C++ é permitido atribuir-se um inteiro a um ponteiro. Evidentemente tais misturas de tipos são fontes de inúmeros erros de processamento. Cabe salientar que ambas as linguagens controlam tipos, emitindo mensagens de advertência ou de erro quando forem encontradas misturas duvidosas de tipos. No entanto, através de *type casts* o programador pode definir explicitamente o tipo a ser utilizado para interpretar determinado espaço de dados.

Tipos computacionais aparecem em duas formas, *tipos primários* e *tipos declarados*. Um tipo primário é um tipo disponível na linguagem de programação. São exemplos: *caractere*, *inteiro*, *flutuante*, e *ponteiro*, bem como vetores formados por tais tipos, tais como *strings* (vetores de caracteres) e matrizes de flutuantes. Um tipo declarado, é um tipo criado pelo programador. Em C e em C++ programadores podem criar tipos utilizando declarações `class`, `struct`, `union`, `enum`[3] ou `typedef`, os compiladores controlarão o correto uso destes tipos. Por exemplo, é emitida uma mensagem de erro ao atribuir um ponteiro para um objeto de uma classe a uma variável declarada para conter um ponteiro para uma determinada estrutura. Ou, ainda, ao atribuir um ponteiro para um objeto de uma classe a um ponteiro para um objeto de outra classe que não faça parte da estrutura de herança.

No entanto, o mero controle de tipos computacionais é insuficiente, uma vez que permite que se faça coisas tais como: atribuir o número de elementos disponíveis em estoque — inteiro representando uma contagem — ao valor no estoque — inteiro representando uma quantia; ou atribuir um valor flutuante que representa uma velocidade a um outro valor flutuante que representa uma distância. O que se observa nestes exemplos é que cada elemento de programa possui um *tipo semântico* em adição ao tipo computacional. O tipo semântico denota o *significado* do elemento. Assim dois elementos com o mesmo tipo computacional podem corresponder a valores com significados distintos.

Para facilitar o correto entendimento de um nome, é necessário que ele reflita correta e completamente os três tipos: tipo computacional primário, tipo computacional declarado e tipo semântico. Isto será conseguido estabelecendo uma estrutura de composição dos nomes de elementos. Nesta estrutura, o tipo primário faz parte do **prefixo de tipo**, enquanto que o tipo semântico faz parte do **tema** (definido mais adiante). O tipo declarado ora fará parte do **prefixo de tipo**, sempre que se tratar de um nome de tipo padronizado, ou do **tema**, sempre que o nome não for padronizado. Esta forma híbrida reduz em muito o tamanho das tabelas de nomes padronizados, simplificando a adoção da presente norma.

Cabe aqui uma pequena ressalva de terminologia. Encontra-se freqüentemente o termo *tipo abstrato* associado a uma declaração de tipo `struct` ou `union`. O uso deste termo para estas construções vale somente se elas contiverem também todas as funções que manipulam os elementos de dados contidos. Isto decorre da definição de um tipo abstrato: um tipo definido pelas operações realizadas sobre uma estrutura de dados cuja organização está encapsulada. Quando estas construções não definem as operações, elas são meramente um tipo de dados, já que somente definem o conjunto de valores que uma variável deste tipo poderá ter. Note que uma classe satisfaz a definição de um tipo abstrato. Na realidade uma classe é uma generalização do conceito de tipo abstrato.

---

[3] Cabe salientar que de um ponto de vista rigoroso tipos declarados aparecem sob a forma *<construtor> <nome>*, onde *<construtor>* é um de `enum`, `struct` ou `union`. Utilizamos no texto meramente o *<construtor >*, uma vez que simplifica a redação sem incorrer em ambigüidade.

3.2 Composição de programas C e C++

Nesta seção apresentamos, de forma resumida, a organização padrão do código fonte de programas redigidos em C e em C++. Este padrão será detalhado em outra norma.

Programas C ou C++ são compostos por diversos *módulos*. Cada módulo corresponde a uma unidade de compilação independente. Através de um projeto criterioso assegura-se que cada módulo implementa exatamente uma abstração bem definida. São exemplos de abstrações: listas, tabelas de símbolos, reticulados de pontos, janelas em sistemas semelhantes a Windows. Ao gerar um programa, cada módulo fonte será compilado, produzindo um módulo objeto. Posteriormente, estes serão combinados — *link* — entre si e com módulos compilados contidos em bibliotecas, formando o programa executável.

O código de um módulo fonte é sempre formado por dois arquivos:

- o *módulo de definição*, contém todas as declarações e todo o código de interface necessários para que o módulo possa ser utilizado por outro. Ou seja, todas as declarações de elementos públicos (`extern`, `public`, `protected`) estarão necessariamente contidas no módulo de definição.
- o *módulo de implementação*, contém todas as declarações e o código encapsulado do módulo.

O módulo de definição deve ser redigido de tal forma que:

- múltiplas inclusões dele ao compilar um módulo não gerem duplicidades de declaração.
- ele possa ser utilizado tanto ao compilar módulos cliente, como ao compilar o correspondente módulo de implementação.

Esta organização do código fonte permite criar módulos bem delimitados, implementando uma única abstração. Isto facilita localizar o módulo no conjunto de todos os módulos desenvolvidos. Facilita, ainda, reutilizar o módulo em diferentes programas. O uso de módulos de definição tal como sugerido, assegura que as interfaces estejam sempre completas e corretamente definidas. Como tanto o próprio módulo, como todos os seus módulos cliente utilizam uma mesma definição de interface para serem compilados, não existe o risco da interface esperada por um módulo cliente ser diferente da efetivamente tornada disponível pelo módulo servidor.

O conceito de módulo de definição pode ser generalizado para *módulo de definição de produto*. Este é um módulo de definição, portanto contém somente declarações públicas, declarando todos os elementos públicos tornados disponíveis por um determinado produto. Um exemplo é a definição de interface de uma biblioteca DLL (*dynamic linking library*) ou de um componente tornando disponíveis diversas APIs (*aplication program interface*). Neste caso, todos os nomes públicos de um módulo e que também sejam públicos de um produto que utiliza este módulo, estarão definidos no módulo de definição deste produto ao invés de estarem definidos no módulo de definição próprio do módulo. Este último conterá somente os elementos públicos que não são tornados públicos pelo produto.

4. Definição da norma

Neste capítulo apresentamos a norma do ponto de vista conceitual. No Anexo A é reapresentada a norma, explicando como aplicá-la ao escolher um novo nome para cada uma das categorias de elementos disponíveis na linguagem C++. Finalmente, no Anexo B são listados e explicados todos os componentes padronizados já definidos.

4.1 Estrutura genérica de um nome

Regra 1: Todos os nomes, independentemente da categoria do elemento que denominam, possuem a mesma estrutura. A estrutura padrão dos nomes de elementos é a seguinte:

{<Produto>|<Domínio>}_<Prefixo tipo><Tema><Sufixo Tipo>

onde:

Produto quando presente, identifica o produto que torna público o elemento.

Domínio quando presente, determina o módulo onde o elemento está declarado, ou o escopo da declaração contida em uma classe. Produto e domínio são mutuamente exclusivos.

Prefixo tipo quando presente, identifica o *tipo computacional* do elemento.

Tema determina o significado do elemento designado pelo nome. O tema deve refletir precisamente o significado do objeto ou da ação designada pelo nome[4].

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

Recom. 2: Exceto para elementos rascunho, que serão definidos mais adiante, nomes devem ter entre 8 e 20 caracteres.

Recom. 3: Selecione um idioma (português, inglês) no qual deve ser redigido o módulo. Uma vez que tenha sido escolhido, o módulo deverá ser redigido integralmente neste idioma. Conseqüentemente os catálogos de componentes de nome padronizados, definidos mais adiante, devem considerar os vários idiomas em uso na empresa.

Os exemplos a seguir ilustram como decompor um nome:

DI_CriarJanelaSinc	**Produto**: DI – Desenhador de Interfaces –; **Tema**: CriarJanela; **Sufixo tipo**: Sinc. . Sabe-se que **DI_** é um **Produto** e não um **Domínio** consultando a tabela de nomes padronizados da empresa. Reconhece-se que se trata de uma função, pelo fato do componente **Tema** começar com um verbo no infinitivo.
BT_fInserirChave	**Domínio**: BT; **Prefixo tipo**: f – a função retorna uma condição de retorno (*flag*); **Tema**: InserirChave. Sabe-se que **BT_** é um **Domínio** e não um **Produto** consultando a tabela de nomes padronizados da empresa.
i	variável local de rascunho, índice.
idPagina	variável local, **Prefixo tipo**: id; **Tema**: Pagina.
BT_ID_ERRO_TAM_VALOR	**Domínio**: BT; **Prefixo tipo**: ID; **Tema**: ERRO_TAM_VALOR. Trata-se de uma constante global declarada através de um *#define*. Reconhece-se isto pelo fato do nome ser redigido somente com letras maiúsculas. Reconhece-se que a constante é global pública pelo fato de iniciar com um domínio.

[4] Escolhemos o termo *Tema* ao invés do termo *Nome* para evitar ambigüidades na redação. Adicionalmente, este termo corresponde exatamente à intenção com que aparece na norma, qual seja: *1. proposição que vai ser tratada ou demonstrada; assunto; 3. radical ou elemento primitivo de uma palavra, ao qual se acresce uma desinência ou sufixo* [Dicionário Aurélio Eletrônico, v1.3].

ID_PAG_NIL — **Prefixo tipo**: ID; **Tema**: PAG, **Sufixo tipo**: NIL. Trata-se de uma constante global encapsulada declarada através de um *#define*, representando um valor identificando uma página não definida.

4.2 Prefixo de produto

Regra 4: O prefixo de produto identifica os nomes públicos tornados disponíveis por produtos de software. O prefixo de produto é opcional. Quando existe, o correspondente elemento é público e estará declarado no módulo de definição do produto. O prefixo de produto é mutuamente exclusivo com o prefixo de domínio.

Regra 5: O prefixo produto é composto por duas ou três letras, devendo ser diferente de **st_**, **pr_**, **pv_**, **m_**, **mpr_**, **mpv_**, descritos mais adiante nesta norma, bem como dos prefixos de outros produtos e dos prefixos de domínio conhecidos na empresa.

Recom. 6: O prefixo de produto deve ser estabelecido pela gerência de desenvolvimento.

O prefixo de produto e o prefixo de domínio servem a dois propósitos:

1. assegurar que elementos públicos distintos tenham necessariamente nomes diferenciados entre si, sem requerer que os desenvolvedores precisem se comunicar para assegurar esta diferenciação.
2. facilitar a localização do módulo de definição que torna disponível um nome público.

O prefixo de produto é utilizado para definir elementos públicos de um produto que poderá ser combinado com outros programas. São exemplos de tais produtos: bibliotecas de classes de uso generalizado, APIs e funções públicas contidas em bibliotecas DLL.

O prefixo de produto designa elementos públicos para toda a empresa e, possivelmente, para clientes dela. O prefixo de produto pode corresponder a produtos desenvolvidos pela empresa ou a produtos de terceiros utilizados pela empresa. Por exemplo, nas bibliotecas de desenvolvimento de aplicações de Windows da Microsoft, o prefixo **WM_** é utilizado para designar mensagens de janelas.

Os objetivos do prefixo de produto são:

- assegurar que todos os nomes públicos tornados disponíveis pelo produto sejam diferentes de nomes contidos em outros produtos e de nomes contidos em módulos de um determinado projeto. Note que os demais componentes do nome podem ser iguais, e, mesmo assim, é assegurada a distinção entre os nomes. Exemplo: *ABC_ExibirJanela* e *EFG_ExibirJanela*. Os dois nomes, apesar de possuírem a mesma semântica – exibir o conteúdo de uma janela –, são distintos um do outro.
- assegurar que o usuário cliente do produto disponha de uma forma homogênea de se referir a elementos do produto. Isto facilita ao cliente determinar o significado específico de cada elemento, bem como fornece meios para localizar a documentação contendo a descrição de uso detalhada do elemento.

A gerência de desenvolvimento deve escolher um nome de prefixo de produto sugestivo do nome do produto. Pode ser conveniente criar prefixos de produto diferentes para partições do produto. Neste caso, devem existir tantos módulos de definição de produto quantas forem as partições do produto.

4.3 Prefixo de domínio

Regra 7: O prefixo de domínio identifica os nomes globais de um módulo ou de uma classe. O prefixo de domínio é utilizado, ainda, para identificar o escopo de membros de classes. O prefixo de domínio é opcional. Quando existe, o correspondente elemento é global. O prefixo de domínio é mutuamente exclusivo com o prefixo de produto.

Regra 8: Os seguintes prefixos de domínio possuem significado especial:

st_ identifica elementos globais encapsulados no módulo. Estes nomes estarão definidos no módulo de implementação. Caso se trate de uma variável ou de uma função, ela deverá ter o atributo `static`.

m_ identifica os membros públicos de uma classe.

mpv_ identifica os membros privados de uma classe.
mpr_ identifica os membros protegidos de uma classe.
vazio nomes sem prefixo de produto nem de domínio, identificam elementos locais de uma função, ou métodos de uma classe. Distingue-se a categoria do elemento pelo componente **Tema** do nome. Se este inicia por um verbo no infinitivo, ou pela palavra **On** seguida de um verbo no infinitivo, trata-se de um método ou função, caso contrário o nome é o de uma variável local.

Regra 9: prefixo de domínio é composto por uma a três letras, devendo ser diferente de **st_**, **pr_**, **pv_**, **m_**, **mpr_**, **mpv_**, e dos demais prefixos de produto e de domínio conhecidos. Caso o prefixo de domínio não seja um de **st_**, **pr_**, **pv_**, **m_**, **mpr_**, **mpv_**, o nome é público e estará definido no módulo de definição.

Regra 10: Cada módulo define um prefixo de domínio próprio, independentemente de se definir elementos globais ou não.

Recom. 11: O prefixo domínio de cada módulo deve ser estabelecido pelo líder de projeto.

Para efeito desta norma distinguimos entre *funções* e *métodos*. Quando uma função não for membro de uma classe ela continuará a ser chamada de função. No entanto, as funções membro de classes, serão chamadas de métodos. Similarmente, distinguimos um *elemento global* de um *elemento membro*. Um elemento global não pertence a uma classe, já um elemento membro está definido no interior de uma classe.

Nesta norma não distinguimos métodos quanto ao fato de serem públicos, protegidos ou privados. Isto se deve ao fato da linguagem C++ permitir a redefinição do escopo através de *overloading*. Outras normas tratarão de regras e recomendações relativas ao bom uso da linguagem C++.

O prefixo de domínio é utilizado para identificar os elementos públicos de um projeto, bem como os nomes encapsulados em um módulo ou classe. Caso se trate do projeto de um produto, nomes públicos identificados por prefixos de domínio não estarão disponíveis no produto, sendo seu uso restrito ao conjunto de módulos que perfazem este produto. Deve-se procurar uma abreviação sugestiva do significado do módulo ao definir o prefixo de domínio.

Os objetivos do prefixo de domínio são:

- assegurar que todos os nomes públicos de módulos de um determinado projeto sejam diferentes dos nomes públicos em outros módulos deste mesmo projeto, bem como que sejam diferentes de nomes globais encapsulados no módulo ou em uma classe.
- assegurar que o usuário cliente do módulo disponha de uma forma homogênea de se referir a elementos deste módulo. Isto facilita ao cliente determinar o significado e o módulo de definição que contém a declaração dos nomes de todos os elementos globais que porventura utilize.

Exemplos

BT_pObterElementoLista	identifica uma função que retorna um ponteiro – p – para um elemento de lista e está definida no módulo BT.
ST_NUM_PORTADORES	é uma constante simbólica declarada por `#define`, encapsulada no módulo a que pertence o texto corrente. Para satisfazer o fato de ser encapsulada, a constante deve estar declarada no respectivo módulo de implementação.
ExibirJanela	é um método de uma classe. Descobre-se isto pelo fato do nome não possuir prefixo de produto, nem de domínio, nem de escopo e principiar por verbo.
mpv_idSimb	é uma variável privada membro de uma classe, e que contém a identificação de um símbolo.

4.4 Regras comuns a prefixos de produto e de domínio

Regra 12: Os prefixos produto têm precedência sobre os prefixos de domínio.

Regra 13: O prefixos de produto e domínio são redigidos em letras maiúsculas e são separados dos demais componentes do nome por um caractere "_" (sublinhado).

Exceção 14: Os seguintes prefixos de domínio, na realidade *prefixos de escopo*: **st_**, **pr_**, **pv_**, **m_**, **mpr_**, **mpv_** são redigidos em letras minúsculas, exceto quando fizerem parte de constantes simbólicas declaradas com `#define`, quando serão redigidos em letras maiúsculas.

Recom. 15: A gerência de desenvolvimento deve manter um documento – *Catálogo de Prefixos de Produtos e Domínios* – identificando todos os prefixos produto e de domínio conhecidos na empresa. O documento deve registrar, para cada produto, subproduto e módulo, o prefixo e o nome completo do produto ou do módulo.

Os prefixos de produtos e os de domínio são opcionais e mutuamente exclusivos. A identificação da categoria do prefixo é feita procurando-se pelo prefixo em uma lista de prefixos conhecidos mantido pela gerência de desenvolvimento. A ausência de um prefixo inicial terminando com o caractere sublinhado, denota a inexistência no nome dos prefixos de produto e de domínio. Distinguem-se os prefixos de escopo dos prefixos de produto e de domínio pelo fato de serem redigidos com letras minúsculas.

Evidentemente todos os elementos públicos de um produto estarão definidos em algum módulo. A Regra 12 estabelece que prefixos de domínio são ignorados sempre que um elemento é tornado público por um produto. Note que diferentes módulos podem tornar públicos elementos de um mesmo produto. Mais ainda, nem todos os nomes públicos de um dado módulo estarão disponíveis nos produtos que utilizam este módulo. Ou seja, um módulo pode utilizar nomes públicos com prefixos de produto, declarados no correspondente módulo de definição de produto, e outros nomes públicos com prefixos de domínio, declarados no módulo de definição do próprio módulo.

Os prefixos de produto e de domínio devem ser diferentes uns dos outros. Desta forma jamais surgirão ambigüidades ao encontrar nomes públicos. Para assegurar esta distinção de nomes, deve-se criar e manter um *Catálogo de Prefixos de Produtos e Domínios* centralizado. O catálogo registra em um único documento ou base de dados todos os prefixos de produto e de domínio conhecidos na empresa. Idealmente este catálogo será organizado sob a forma de um hipertexto capaz de ser consultado durante a criação, leitura ou alteração de um módulo.

No conjunto de prefixos de escopo (ver Exceção 14) estão previstos os prefixos **pv_** e **pr_**. Na presente norma estes prefixos não estão sendo utilizados. Se fossem utilizados, eles corresponderiam a identificadores, respectivamente, de métodos *privados* e de métodos *protegidos*. A restrição de uso destes prefixos é preventiva, visando facilitar a introdução de novas regras caso venha-se a querer diferenciar explicitamente o escopo de métodos.

Ao encontrar um nome de um elemento, determina-se o seu escopo e local onde se encontra a sua declaração examinando o prefixo de produto ou de domínio:

- *prefixo igual a um dos prefixos de produto conhecidos*: neste caso o nome identifica um elemento público de um produto. A declaração do elemento estará contida no módulo de definição do produto correspondente ao prefixo de produto.
- *prefixo igual a um dos prefixo de domínio*: neste caso o nome identifica um elemento público tornado disponível por um módulo. A declaração do elemento estará contida no módulo de definição do módulo correspondente ao prefixo de domínio.
- *prefixo igual a **st_***: neste caso o nome identifica um elemento global encapsulado no módulo e não pertence a qualquer uma das classes definidas neste módulo. A especificação do elemento estará contida no módulo de implementação em que aparece o nome. Caso seja uma variável ou uma função, ela está declarada com o atributo `static`.
- *prefixo igual a **m_***: neste caso o nome identifica um elemento público de uma classe onde este elemento não é um método. A declaração do elemento estará contida na declaração da classe.
- *prefixo igual a **mpv_***: neste caso o nome identifica um elemento privado de uma classe onde este elemento não é um método. A declaração do elemento estará contida na declaração da classe.
- *prefixo igual a **mpr_***: neste caso o nome identifica um elemento protegido de uma classe onde este elemento não é um método. A declaração do elemento estará contida na declaração da classe.

- *ausência de prefixo*: neste caso o nome identifica um método público, protegido ou privado de uma classe se o tema iniciar por um verbo, ou identifica um elemento local à função a que pertence o código corrente caso o tema não principie com um verbo.

4.5 Prefixo de tipo

Regra 16: Todos os elementos de um módulo devem identificar o seu tipo computacional através do prefixo de tipo.

Exceção 17: Não terão prefixo de tipo:

1. métodos,
2. funções retornando `void`,
3. constantes simbólicas inteiras, inclusive constantes enumeradas,
4. nomes cujo tema implica, por convenção, o tipo computacional,
5. elementos locais de rascunho, definidos mais adiante nesta norma.

Regra 18: Todos os caracteres do prefixo de tipo devem ser redigidos em minúsculas sem separação do restante do nome.

Exceção 19: Quando o nome designar uma classe, o caractere inicial do prefixo de tipo será "C" maiúsculo.

Exceção 20: Quando o nome for o de uma constante simbólica declarada em `#define`, todos os caracteres do prefixo de tipo deverão ser maiúsculas e o prefixo deverá ser separado do restante do nome por um caractere sublinhado.

Regra 21: O prefixo de tipo de declarações `struct`, `union`, `enum` e `typedef` deve iniciar com as letras *tp* (tipo ou *type*).

Recom. 22: A gerência de desenvolvimento deve manter um *Catálogo de Prefixos de Tipo*.

Para facilitar o entendimento de um programa, e para aumentar a capacidade do programador ou revisor determinar se o programa está correto, é recomendado que o **tipo computacional** do elemento faça parte integrante do nome. Desta forma reduz-se a freqüência com que o programador ou revisor tem que recorrer a documentação relativa ao elemento. Adotamos nesta norma a *convenção húngara* desenvolvida por Charles Simony.

O tipo computacional denotado pelo prefixo tipo tem o formato:

<id tipo primário><id tipo declarado> onde:

id tipo primário é um dos tipos conhecidos pela linguagem sendo usada, por exemplo: inteiro, ponteiro, caractere, string, classe, declaração de tipo e vetor.

id tipo declarado é um dos tipos de dados criados pelo programador por intermédio de declarações: `class`, `struct`, `union`, `enum` ou `typedef`, por exemplo: a classe *CAluno*, ou a estrutura *tpElemLista*.

Os prefixos de tipos primários são sempre padronizados, enquanto que somente alguns dos prefixos de tipos declarados mais comuns serão padronizados. A tabela de prefixos de tipos padronizados está contida no Anexo B. O prefixo de tipo de um nome é composto pela concatenação de um ou mais prefixos de tipo primário e, caso o elemento possua um tipo declarado padrão, o prefixo de tipo deste tipo declarado.

Tendo em vista que ao programar, manter ou revisar um módulo, o programador ou revisor terá necessariamente que conhecer o tipo de cada elemento utilizado, sejam estes públicos ou internos ao módulo, recomenda-se fortemente que os nomes identifiquem o tipo semântico do elemento. Isto pode ser conseguido em uma das formas a seguir:

1. o tipo declarado é padronizado e incluído nas tabelas de nomes de tipos padrão. Por exemplo: *wnd* pode ser padronizado para denotar o tipo *descritor de janela*.
2. o tipo declarado é não padronizado sendo incorporado ao tema. Neste caso recomenda-se incluir na documentação das declarações de `class`, `struct`, `union`, e `typedef` um comentário estabelecendo um identificador de tipo declarado curto. Este identificador curto será

sempre incorporado ao tema. Por exemplo a classe *CAluno* poderia ser utilizada para declarar os objetos *AlunoNovo*, e *AlunoFormando*.

3. o tipo declarado é definido por convenção para algumas das palavras padronizadas do tema. As palavras padronizadas fazem parte das tabelas de componentes padronizados de nomes. Por exemplo, a palavra padronizada *Num* denota um inteiro maior ou igual a zero (`unsigned`).

Prefixos de tipo podem introduzir alguma ambigüidade, no sentido que um mesmo prefixo de tipo poderá ser interpretado de mais de uma maneira. No entanto, como o prefixo de tipo é meramente um reforço para a memória do leitor, esta ambigüidade é facilmente resolvida pelo contexto do programa no qual apareça o elemento.

Os identificadores de tipos primários e de tipos declarados estão definidos no *Catálogo de prefixos de tipo*. Este catálogo deve ser organizado como um hipertexto e deve estar disponível ao criar, examinar ou alterar um módulo. A título de ilustração, apresentamos a seguir uma lista parcial dos prefixos de tipo padronizados. Uma lista mais elaborada encontra-se no Anexo B.

Exemplos de tipos primários

c	char
C	nome de classe
h	"handle", o tipo computacional depende do recurso identificado
i	inteiro declarado com `int`
id	identificador de um elemento, usualmente um valor `unsigned`
l	long
p	ponteiro
s	string
sz	string terminado com o caractere zero
tp	declaração de tipo, utilizado em nomes de `struct`, `union`, `enum` e `typedef`

Exemplos de tipos declarados

mnu	menu
msg	mensagem
wnd	descritor de janela

Exemplos

hwndCorrente	handle da janela corrente. A documentação de hwnd faz parte dos manuais do compilador. Note que o identificador de prefixo *wnd* é suficiente para que se saiba que se trata de uma janela.
pszNomeAluno	ponteiro para um string terminado em zero contendo o nome de um aluno.
UN_CAluno	classe Aluno.
tpDiaSemana	tipo enumeração de dias da semana.
DiaSemanaSegunda	valor constante enumerada do tipo dia da semana, denotando segunda-feira.

4.6 Tema

Regra 23: Cada tema é composto de uma ou mais palavras. A leitura das palavras da esquerda para a direita deve refletir precisamente o significado do elemento denotado pelo tema.

Regra 24: Cada palavra componente de um tema deve iniciar com uma letra maiúscula, sendo as demais letras minúsculas. Não deve existir separador entre as palavras, nem entre o tema e o restante do nome.

Exceção 25: No caso de constantes declaradas em `#define`, todas as palavras do tema devem ser redigidas com letras maiúsculas e devem ser separadas entre si e dos demais componentes do nome por um caractere sublinhado.

Regra 26: A palavra inicial de um tema que denomina uma função ou método deve ser um verbo no infinitivo.

Exceção 27: Métodos que processam eventos devem iniciar com a palavra **On** seguida de um verbo no infinitivo.

Regra 28: Todas as palavras que formam um tema devem ser palavras existentes no idioma utilizado, devem estar ortograficamente corretas e devem ser utilizadas com o seu significado mais comum.

Recom. 29: Assegure que os nomes dos elementos de um programa se diferenciam em mais de um caractere.

Recom. 30: A gerência de desenvolvimento deve manter um *Catálogo de Palavras e Abreviações Padronizadas*. Este catálogo, além de registrar todos as palavras e abreviações, deve fornecer, ainda, o significado padrão.

Recom. 31: Ao criar nomes de elementos, deve-se utilizar as abreviações contidas no catálogo.

Recom. 32: Procure utilizar somente palavras comuns. Evite palavras "difíceis", ambíguas ou de pouco uso.

Recom. 33: Assegure que cada palavra corresponda a um conceito concreto.

Recom. 34: Utilize sempre a mesma palavra para denotar um dado conceito. Evite o uso de sinônimos e de pronomes.

Recom 35: Associe sempre o mesmo conceito a uma dada palavra. Evite o uso de palavras com múltiplos significados

Recom. 36: Os verbos escolhidos como palavras denotadoras de função ou método devem corresponder sempre a ações cujo efeito seja claramente observável.

Recom. 37: Para manter o tamanho de nomes dentro do limite de 8 a 20 caracteres, as palavras que formam o correspondente tema podem ser abreviadas. No entanto deve-se sempre dar preferência à clareza e não ao tamanho do nome.

Recom 38: Nunca abrevie o verbo que denota uma função ou método.

Recom. 39: Para reduzir o tamanho de um tema, primeiro elimine artigos, advérbios, pronomes e preposições do conjunto de palavras que formam o tema. Elimine também as palavras que não contribuam para definir o significado exato do tema.

Recom. 40: Ao abreviar, evite trocadilhos e homofonias.

Recom. 41: Utilize uma das formas de abreviar a seguir, procurando sempre a que melhor reflita o significado do elemento sendo batizado:

1. Se a primeira sílaba da palavra for suficiente para denotar o significado, utilize a primeira sílaba. Caso não seja suficiente, observe os itens a seguir:
2. Se a palavra inicia com uma consoante, elimine todas as vogais da palavra.
3. Se a palavra inicia com uma vogal, deixe a vogal inicial e elimine todas as demais.

Recom. 42: Não abrevie palavras caso não economize 2 ou mais caracteres.

Recom. 43: Evite distinguir temas através de um numeral.

Recom. 44: Procure dar nomes curtos a temas de classes e de tipos declarados.

Recom. 45: Inclua o tema da classe ou do tipo declarado no tema do elemento que contenha, referencie ou retorne um valor deste tipo ou classe. Evidentemente, quando o prefixo tipo do nome já referenciar este tipo declarado, esta recomendação deixará de ser relevante.

O componente **Tema** do nome de um elemento designa o significado do elemento. Deve-se procurar utilizar temas sugestivos deste significado, ou seja deve-se procurar temas mnemônicos. Deve-se também procurar utilizar os temas de forma consistente, ou seja, para cada conceito use sempre o mesmo tema e, para cada tema, utilize o mesmo conceito. Considere como exemplo a palavra *número*. Usualmente ela denota uma quantidade. No entanto, como muitas identificações são numéricas, a palavra número freqüentemente é utilizada como identificação, por exemplo *número da conta corrente*. Esta ambigüidade é ruim. Para eliminá-la, basta que se convencione utilizar *número* sempre para denotar quantidade e *identificador* para o outro caso. Segundo esta convenção utilizaria-se *identificação da conta corrente*, mesmo que esta seja representada por um valor numérico.

Uma fonte perene para falhas de entendimento é o uso de palavras "inventadas", com significado incomum ou mesmo inexistentes. Ao criar temas, e com isto nomes de elementos, deve-se procurar em primeiro lugar a cla-

reza. Isto é muitas vezes difícil, já que o que é claro para a pessoa que está criando o nome, pode ser nada claro para as pessoas que eventualmente lerão este nome. O uso de palavras simples, de uso comum tende a reduzir a dificuldade de comunicação.

A gerência de desenvolvimento mantém um *Catálogo de Termos Padronizados*. Este catálogo registra todas as palavras, abreviações, prefixos e sufixos em uso na empresa. A versão inicial deste catálogo corresponde ao Anexo B. A organização típica de um catálogo como este deve ser a de um hipertexto.

Os nomes utilizados em um programa podem, em geral, ser agregados em famílias. Por exemplo é comum um programa conter funções, contadores, dimensões, valores mínimos ou máximos, todos relativos a uma mesma abstração. Esta abstração é identificada pelo tema. Através de um criterioso uso de palavras, pode-se criar famílias de nomes, onde cada membro da família designa um elemento relacionado a uma mesma abstração. Considere, por exemplo o tema *ItemEstoque*. Este pode ser utilizado para denotar a família de tudo o que tem a ver com a entidade do mundo real item de estoque. Exemplos de nomes relacionados são:

tpItemEstoque	designa um tipo declarado, tipicamente um `struct` que contém os dados de um item de estoque. Note que se o item de estoque correspondesse a uma classe ele teria o nome *CItemEstoque*. Note ainda que o nome do tema do tipo (ou da classe) é *ItemEstoque*.
pObterItemEstoque	designa uma função que, dada uma identificação de um item de estoque, torna disponíveis os dados do correspondente item de estoque. Esta função retorna um ponteiro para um item de estoque.
szNomeItemEstoque	string do nome do item de estoque, onde o *string* é terminado com zero. O nome é um dos campos do tipo (ou membro da classe) *ItemEstoque*.
idItemEstoque	identificador do item de estoque.
sidItemEstoque	identificador do item de estoque, neste caso um *string*.
DispItemEstoque	disponibilidade de determinado item no estoque.
NumItemEstoque	contagem de diferentes itens contidos no estoque. Note a potencial ambigüidade do prefixo *Num*. Se utilizado de forma não uniforme, em um determinado contexto poderia designar a disponibilidade de um certo item e, em outro contexto, a contagem de diferentes itens. Adotamos aqui o significado padrão "contagem". Os dois exemplos, número e disponibilidade, ilustram o uso do tema para denotar o tipo semântico.
ID_ITEM_ESTOQUE_NIL	constante designando uma identificação de item de estoque inexistente.

Uma linha de código utilizando estes elementos poderia ter o seguinte aspecto:

```
pItemEstoque = pObterItemEstoque( idItemEstoque ) ;
```

Note que, mesmo sem conhecer as declarações completas de cada um dos elementos acima, podemos entender perfeitamente o significado desta atribuição, podendo, inclusive, verificar se a atribuição faz uso correto dos tipos e da semântica dos valores. Finalmente, em muitos casos podemos verificar se o uso de uma função está correto. Por exemplo, o prefixo *Obter* denota uma função que busca um valor correspondente a uma dada identificação. Portanto, a lista de parâmetros desta função deve conter um identificador. O componente **tema** deste identificador deve ser igual ao componente **tema** da função.

Nomes de elementos devem ter um tamanho entre 8 e 20 caracteres. Isto torna necessário abreviar os temas de modo que se consiga manter este tamanho. Ao abreviar observe as recomendações acima e, em especial, as a seguir:

- Utilize abreviações padronizadas contidos no *Catálogo de Palavras e Abreviações Padronizadas*.
- Dê sempre preferência à clareza e não ao tamanho do nome.
- Ao abreviar, evite trocadilhos e homofonias. Por exemplo, evite BIN2DEC para denotar uma conversão " binary to decimal").

Exemplos de abreviações:

Nome Original	Nome abreviado	Explicação
JanelaCorrente	JanCorr	*Jan* e *Corr* são abreviações padronizadas
ItemDeEstoque	ItemEstoque	eliminou-se a preposição *de*
NumeroDeLinhasNoTexto	NumLinTexto	utilizaram-se abreviações padronizadas
CalcularTamanhoDoSalto\ DeRolagemDaPagina	CalcularTamSaltoPag	Eliminaram-se palavras pouco significativas e abreviaram-se outras

4.7 Sufixo de tipo

Regra 46: Somente poderão ser utilizados sufixos de tipo padronizados. O Anexo B contém a versão inicial do *Catálogo de Sufixos de Tipo Padronizados*.

O sufixo de tipo complementa a semântica do tipo computacional do elemento. São poucos os casos em que ele se aplica, sendo todos eles identificáveis. Por esta razão, é proposto restringir o uso de sufixos àqueles previamente definidos.

Exemplos

ID_ITEM_ESTOQUE_NIL	constante designando uma identificação de item de estoque inexistente. O termo NIL é um sufixo tipo que denota chave ou identificador nulo.

4.8 Variáveis de rascunho

Regra 47: Variáveis de rascunho podem ser redigidas de forma simplificada.

Uma variável é de *rascunho* se satisfaz a todas as condições a seguir:

- é local a uma função.
- não é parâmetro desta função.
- se for passada como parâmetro, é um parâmetro `const`.
- não é inicializada na declaração.
- possui pequenos domínios de definição.

Entende-se por *domínio de definição* o número de linhas de código iniciando em uma atribuição à variável até o seu último uso antes da próxima atribuição ou do final do bloco em que é declarada. Assume-se nesta contagem que o programa é estruturado.

Exemplos

i, j, k	contadores de ciclos
Buffer	armazenamento temporário de dados de entrada
Temp	valor temporário

4.9 Constantes enumeradas

Regra 48: O início do tema de cada uma das constantes enumeradas deve ser igual ao nome da enumeração, excluindo-se o prefixo de tipo.

Ao encontrar uma constante enumerada é necessário saber qual é a enumeração que declara a constante. Utilizando o nome do tipo enumeração como prefixo do tema de cada uma das constantes enumeradas torna imediata a identificação da constante.

Exemplo em C++:

```
enum JAN_tpCores
{
   JAN_CoresAzul ,
   JAN_CoresVermelho ,
   JAN_CoresVerde ,
   JAN_CoresAmarelo
}
```

Exemplo em C:

```
typedef enum
{
   JAN_CoresAzul ,
   JAN_CoresVermelho ,
   JAN_CoresVerde ,
   JAN_CoresAmarelo
} JAN_tpCores
```

Anexo A Como escolher nomes

Neste anexo ilustramos como utilizar a norma em diversas situações encontradas ao redigir um programa. O texto do anexo é simplificado com relação à norma exposta nas seções anteriores. Portanto, para um entendimento mais completo e aprofundado, é fortemente recomendado ler-se o corpo da norma.

A estrutura geral utilizada para todos os nomes, independentemente da categoria do elemento que denominam, possui a mesma estrutura. A estrutura padrão dos nomes é a seguinte:

{<Produto>|<Domínio>}_<Prefixo tipo><Tema><Sufixo Tipo>

onde:

Produto	quando presente, identifica o produto que torna público o elemento.
Domínio	quando presente, determina o módulo onde o elemento está declarado, ou o escopo da declaração em uma classe. Produto e domínio são mutuamente exclusivos.
Prefixo tipo	quando presente, identifica o *tipo computacional* do elemento.
Tema	determina o significado do elemento designado pelo nome. O tema deve refletir precisamente o significado do objeto ou da ação designada pelo nome.
Sufixo tipo	quando presente, complementa o *tipo computacional* do elemento.

Os exemplos a seguir ilustram como decompor um nome redigido de acordo com esta regra:

DI_CriarJanelaSinc	**Produto**: DI – Desenhador de Interfaces –; **Tema**: CriarJanela; **Sufixo tipo**: Sinc. Reconhece-se que se trata de uma função pelo fato do componente **Tema** começar com um verbo no infinitivo.
BT_fInserirChave	**Domínio**: BT; **Prefixo tipo**: f – a função retorna uma condição de retorno; **Tema**: InserirChave. Sabe-se que **BT_** é um **Domínio** e não um **Produto** consultando a tabela de nomes padronizados no Anexo B
i	variável local de rascunho, índice.
idPagina	variável local, **Prefixo tipo**: id; **Tema**: Pagina.
BT_ID_ERR_TAM_VALOR	**Domínio**: BT; **Prefixo tipo**: ID; **Tema**: ERR_TAM_VALOR. Trata-se de uma constante global declarada através de um *#define*. Reconhece-se isto pelo fato do nome ser redigido somente com letras maiúsculas. Reconhece-se que a constante é global pelo fato de iniciar com um domínio.

Nas seções a seguir descreveremos o uso desta regra geral nas diferentes situações que ocorrem ao se escrever um programa. Cada seção definirá uma variante desta regra geral, que, na realidade, corresponde a uma simplificação específica para a situação de uso.

Módulos e produtos

Ao iniciar a criação de um novo módulo defina um nome prefixo de domínio para este módulo. Da mesma forma, ao criar um novo produto, defina um nome prefixo de produto para este nome. Usualmente estes nomes serão criados pela gerência de desenvolvimento. Os nomes dos módulos e dos produtos e os correspondentes prefixos de domínio e de produto devem ser incorporados ao *Catálogo de Nomes Padronizados*. O Anexo B é um exemplo deste catálogo.

Assegure-se que o nome é único no conjunto de todos os nomes de prefixo de produto e de domínio conhecidos. Assegure-se, também, que o nome escolhido para o prefixo reflita bem o significado do produto ou módulo.

Os nomes externos ou públicos de um produto serão todos declarados no módulo de definição deste produto. Os nomes externos ou públicos de um módulo serão todos declarados no módulo de definição deste módulo exceto quando forem nomes públicos de produto.

Classes

Os nomes de classes devem ter a seguinte estrutura

{<Produto>|<Domínio>}_C<Tema>

onde:

Produto quando presente, identifica o produto que torna pública a classe.

Domínio quando presente, determina o módulo onde a classe pública está declarada. Produto e domínio são mutuamente exclusivos. Um dos dois deve estar presente.

Prefixo tipo é sempre a letra **C** maiúscula.

Tema denomina a classe designada pelo nome. O tema deve refletir precisamente o significado da abstração implementada pela classe. A primeira palavra do tema deve ser um substantivo. Procure utilizar nomes curtos – sintéticos – para classes

Exemplos de declaração e de uso de classes:

ABC_CClipboard	**Domínio**: *ABC*; **Prefixo de tipo**: *C* – classe –; **Tema**: Clipboard. Note que o prefixo de tipo e o tema iniciam com a mesma letra, "C", o que provoca a sua duplicação no nome.
pCClipboardCorr	ponteiro para um objeto da classe *Clipboard*.

Funções externas, APIs

Uma função é externa se ela não pertence a uma classe e não é definida `static`. Os nomes de funções externas devem ter a seguinte estrutura

{<Produto>|<Domínio>}_<Prefixo tipo><Tema><Sufixo Tipo>

onde:

Produto quando presente, identifica o produto que torna pública a função (API).

Domínio quando presente, determina o módulo onde a função está declarada. Produto e domínio são mutuamente exclusivos. Um dos dois prefixos produto ou domínio deverá estar presente.

Prefixo tipo quando presente, identifica o *tipo computacional* retornado pela função. Quando não existe, a função retorna `void`.

Tema identifica precisamente a ação realizada pela função. A primeira palavra do tema deve ser um verbo no infinitivo.

Sufixo tipo quando presente, complementa o *tipo computacional* da função.

Com relação ao prefixo tipo da função, deve-se observar o seguinte:

- Todas as funções devem identificar, o tipo retornado, exceto quando retornar nada (`void`).
- O tipo computacional denotado pelo prefixo tipo deve refletir a abstração de dados do valor retornado. Ele é redigido sempre em letras minúsculas e não possui separador.
- O prefixo tipo da função é formado por uma seqüência de um ou mais prefixos de tipo elementares. Estes são usualmente escolhidos no catálogo de prefixos tipo mantido pela empresa, ou pela documentação dos produtos utilizados para desenvolver o sistema, ver Anexo B.
- Caso não exista um identificador de tipo declarado para o valor retornado pela função, procure incorporar no tema da função o tema do tipo ou da classe retornado.
- Ao utilizar compiladores para modelos de memória de 16 bits, por exemplo o compilador Visual C++, evite declarações FAR e, conseqüentemente, designadores de tipo "lp" *long pointer*. Ao invés disso utilize sempre o modelo de memória *large* ao compilar os módulos.

O sufixo tipo complementa a semântica computacional da função. Deve sempre ser escolhido do catálogo de nomes de sufixos tipo.

Exemplos

ABC_pObterItemEstoque	designa uma função global pública, pertencente ao módulo ABC, que, dada uma identificação de um item de estoque, torna disponíveis os dados do correspondente item de estoque. Esta função retorna um ponteiro para um item de estoque.
GCC_fEnviarPacoteAssinc	identifica a função *EnviarPacote*, que retorna uma condição de retorno – *f* –, assíncrona, sufixo tipo: *Assinc*. A função pertence ao produto *GCC*.

Funções encapsuladas

Uma função é encapsulada se ela não pertence a uma classe e é definida `static`. Os nomes de funções encapsuladas devem ter a seguinte estrutura

st_<Prefixo tipo><Tema><Sufixo Tipo>

onde:

st	é o par de letras minúsculas *st*, informando tratar-se de uma função encapsulada.
Prefixo tipo	quando presente, identifica o *tipo computacional* retornado pela função. Quando não existe, a função retorna `void`.
Tema	identifica precisamente a ação realizada pela função. A primeira palavra do tema deve ser um verbo no infinitivo.
Sufixo tipo	quando presente, complementa o *tipo computacional* do elemento.

Os demais componentes do nome seguem os mesmos princípios que os descritos na seção Funções externas, APIs.

Exemplo

st_idGerarIdItemEstoque	função encapsulada que retorna a identificação – *id* – de um novo item de estoque.

Métodos

Uma função é um método se ela é membro de uma classe. Os nomes de métodos encapsulados devem ter a seguinte estrutura

<Tema><Sufixo Tipo>

onde:

Tema	identifica precisamente a ação realizada pela função. A primeira palavra do tema deve ser um verbo no infinitivo, ou, caso o método tratar um evento a primeira palavra é **On** seguida de um verbo no infinitivo, neste caso o nome deveria ser todo em inglês.
Sufixo tipo	quando presente, complementa o *tipo computacional* do elemento.

Os componentes do nome seguem os mesmos princípios que os descritos na seção Funções externas, APIs.

Exemplos:

OnOpenFile	método processador do evento abrir arquivo.
ObterPortador	método que obtém um portador. *Portador* deve ser o tema da classe da qual este método é membro. Note que o tipo retornado pelo método nunca é explicitado. Tampouco é explicitado o escopo do método.

Tipos globais externos

Um tipo é externo se não pertencer a uma classe e for declarado no módulo de definição. Tipos em C++ correspondem a `struct`'s, `union`'s e `typedef`'s. Os nomes de tipos externos devem ter a seguinte estrutura:

{<Produto>|<Domínio>}_tp<Tema><Sufixo Tipo>

onde:

Produto quando presente, identifica o produto que torna público o tipo.

Domínio quando presente, determina onde o módulo onde o tipo está declarado. Produto e domínio são mutuamente exclusivos. Um dos dois prefixos deverá estar presente.

tp é o par de letras *tp*, indicando que se trata da definição de um tipo declarado pelo programador.

Tema identifica precisamente a classe de valores do tipo ou os valores contidos na variável. A primeira palavra do tema deve ser um substantivo. Procure escolher um tema curto, uma vez que este tema deverá ser incluído nos nomes das variáveis e funções que contém, referenciam ou retornam valores deste tipo. Caso o tema seja longo, inclua um comentário definindo o nome abreviado a ser utilizado.

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

O sufixo tipo complementa o tipo computacional da variável ou tipo. Deve sempre ser escolhido do catálogo de nomes de sufixos de tipo.

Exemplos de declarações em C++

struct JAN_tpElemListaJan { ... } ; declaração de um tipo global público definido no módulo *JAN*. Define um elemento de uma lista de janelas. Esta declaração estará contida no módulo de definição.

JAN_tpElemListaJan ElemListaJanOrg ; declaração de uma variável utilizando o tipo acima declarado. Note a inclusão do tema *ElemListaJan* no nome da variável.

ABC_tpItemEstoque designa um tipo público, tipicamente um `struct` que contém os dados de um item de estoque. Note que o nome do tema do tipo é *ItemEstoque*.

As mesmas declarações em C seriam:

typedef struct JAN_tagElemListaJan { ... } JAN_tpElemListaJan ;

struct JAN_tagElemListaJan * ElemListaJanProx ; um campo da estrutura ElemListaJan que aponta para o próximo elemento da lista.

JAN_tpElemListaJan * ElemListaJanOrg ; uma variável ponteiro para uma lista formada por ElemListaJan. Esta variável não faz parte da declaração da estrutura.

Variáveis globais externas

Uma variável é externa se não pertencer a uma classe, nem a uma função, nem for definida `static`. Os nomes de tipos e de variáveis externas devem ter a seguinte estrutura:

{<Produto>|<Domínio>}_<Prefixo tipo><Tema><Sufixo Tipo>

onde:

Produto quando presente, identifica o produto que torna público o tipo ou a variável. Caso se trate de um tipo, deve-se definir um identificador de prefixo tipo para ele.

Domínio quando presente, determina onde o módulo onde o tipo ou a variável estão declarados. Produto e domínio são mutuamente exclusivos. Um dos dois prefixos de produto ou de domínio deverá estar presente.

Prefixo tipo identifica o seu *tipo computacional*.

Tema identifica precisamente a classe de valores do tipo ou os valores contidos na variável. A primeira palavra do tema deve ser um substantivo.

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

Com relação ao prefixo de tipo do elemento, deve-se observar o seguinte:

- Todas as variáveis devem identificar o tipo computacional do valor que poderão conter.
- Caso o tipo da variável seja primário, o prefixo tipo deve refletir integralmente o correspondente tipo computacional.
- Prefixos de tipo são sempre redigidos em letras minúsculas e não possuem separador.
- O prefixo de tipo de uma variável é formado por uma seqüência de um ou mais prefixos de tipo elementares. Estes são escolhidos do *Catálogo de Prefixos de Tipo* mantido pela empresa. O Anexo B contém o catálogo de tipos mais comuns.
- Ao declarar uma variável que contém o valor de um tipo declarado no programa, inclua o nome do tema deste tipo no tema da variável.

O sufixo tipo complementa o tipo computacional da variável ou tipo. Deve sempre ser escolhido do catálogo de nomes de sufixos de tipo.

Exemplos de declarações de variáveis globais públicas

struct JAN_tpElemListaJan { ... } ; declaração C++ de um tipo global público definido no módulo *JAN*. Define um elemento de uma lista de janelas. Esta declaração estará contida no módulo de definição.

JAN_tpElemListaJan * JAN_pElemListaJanOrg ; declaração de uma variável global pública contendo um ponteiro para o tipo acima declarado.

char JAN_vtszTabNomes[JAN_DIM_NOME][JAN_DIM_TAB_NOMES] ;
declaração de um vetor de strings terminados em zero. As constantes *JAN_DIM_NOME* e *JAN_DIM_TAB_NOMES* devem estar definidas (`#define`) no módulo de definição.

Tipos globais encapsulados

Um tipo é global encapsulado se não pertencer a uma classe e for declarado no módulo de implementação. Os nomes de tipos encapsulados devem ter a seguinte estrutura:

st_tp<Tema><Sufixo Tipo>

onde:

st é o par de letras minúsculas st, denotando tratar-se de um tipo ou variável encapsulada.

tp identifica o que se trata de uma declaração de tipo.

Tema identifica precisamente a classe de valores do tipo ou os valores contidos na variável. A primeira palavra do tema deve ser um substantivo.

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

Os componentes do nome seguem os mesmos princípios que os descritos na seção Tipos globais externos.

Exemplo de declaração:

struct st_tpListaJanelas { ... } ; declara um tipo encapsulado correspondente a uma lista de janelas. Para poder ser um tipo encapsulado, deve estar declarado no módulo de implementação.

Variáveis globais encapsuladas

Uma variável é global encapsulada se não pertencer a uma classe nem a uma função e for definida `static`. Variáveis encapsuladas devem ser declaradas no módulo de implementação. Os nomes de tipos e de variáveis encapsulados devem ter a seguinte estrutura:

st_<Prefixo tipo><Tema><Sufixo Tipo>

onde:

st é o par de letras minúsculas *st*, denotando tratar-se de uma variável encapsulada.

Prefixo tipo identifica o seu *tipo computacional*.

Tema identifica o significado dos valores contidos na variável. A primeira palavra do tema deve ser um substantivo.

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

Os componentes do nome seguem os mesmos princípios que os descritos na seção Variáveis globais externas.

Exemplo de declaração

```
static int st_NumJanAbertas ;    declara um valor inteiro.
```

Tipos membro de classes

Um tipo é membro de uma classe se for declarada no corpo da classe. Os nomes de tipos membros de classes devem ter a seguinte estrutura:

<Visibilidade>tp<Tema><Sufixo Tipo>

onde:

Visibilidade determina se o tipo é público, protegido ou privado.

tp identifica tratar-se da declaração de um tipo definido pelo programador.

Tema identifica precisamente a classe de valores do tipo. A primeira palavra do tema deve ser um substantivo.

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

O componente **visibilidade** pode assumir um dos seguintes valores:

mpv_ neste caso o tipo é privado.

mpr_ neste caso o tipo é protegido.

m_ neste caso o tipo é público.

Os demais componentes do nome seguem os mesmos princípios que os descritos na seção Tipos globais externos.

Variáveis membro de classes

Uma variável é membro de uma classe se for declarada no corpo da classe e não pertencer a um método. Os nomes de variáveis membros de classes devem ter a seguinte estrutura:

<Visibilidade><Prefixo tipo><Tema><Sufixo Tipo>

onde:

Visibilidade determina se a variável é pública, protegida ou privada.

Prefixo tipo identifica o seu *tipo computacional*.

Tema identifica os valores contidos na variável. A primeira palavra do tema deve ser um substantivo.

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

O componente **visibilidade** pode assumir um dos seguintes valores:

mpv_ neste caso a variável é privado.

mpr_ neste caso a variável é protegida.

m_ neste caso a variável é pública.

Os componentes do nome seguem os mesmos princípios que os descritos na seção Variáveis globais externas.

Variáveis e parâmetros locais

Uma variável é local caso seja declarada no corpo de uma função ou método. Ela é um parâmetro caso seja declarada na lista de parâmetros formais de uma classe, função ou método. Os nomes de variáveis locais devem ter a seguinte estrutura:

<Prefixo tipo><Tema><Sufixo Tipo>

onde:

Prefixo tipo identifica o seu *tipo computacional*.

Tema identifica precisamente a classe de valores do tipo ou os valores contidos na variável. A primeira palavra do tema deve ser um substantivo.

Sufixo tipo quando presente, complementa o *tipo computacional* do elemento.

Os componentes do nome seguem os mesmos princípios que os descritos na seção Variáveis globais externas.

Variáveis de rascunho

Uma variável é de *rascunho* se satisfaz a todas as condições a seguir:

- é local a uma função ou método.
- não é parâmetro desta função ou método.
- se for passada como parâmetro, é um parâmetro `const`.
- não é inicializada na declaração.
- possui pequenos domínios de definição.

Entende-se por *domínio de definição* o número de linhas de código iniciando em uma atribuição à variável até o seu último uso antes da próxima atribuição ou do final do bloco em que é declarada. Assume-se nesta contagem que o programa é estruturado.

Os nomes de variáveis locais podem ter a seguinte estrutura simplificada:

<Tema>

Exemplos

i, j, k	contadores de ciclos
Buffer	armazenamento temporário de dados de entrada
Temp	valor temporário

Constantes `#define`

Constantes declaradas em `#define` seguem os mesmos princípios que variáveis. Constantes podem ser globais externas, caso estejam declaradas no módulo de definição, ou podem ser globais encapsuladas, caso estejam declaradas no módulo de implementação. O nome de uma constante é redigido todo em letras maiúsculas, separando-se os componentes do nome e as palavras do tema com um caracter sublinhado.

Exemplos:

JAN_ID_JANELA_NIL	Uma constante global definida no módulo *JAN*, que contém um identificador de janela nulo — prefixo de tipo: *ID* e sufixo de tipo: *NIL*.
ST_DIM_NOME_JANELA	Uma constante encapsulada, declarada no módulo de implementação, que contém a dimensão (comprimento máximo incluindo o zero terminal) do nome de uma janela.

Constantes `enum`

Constantes enumeradas são declaradas em um `enum`. O nome do `enum` segue os mesmos princípios que o nome de um tipo. Os nomes de constantes enumeradas ter a seguinte estrutura:

<Nome do enum><Tema>

onde:

Nome do enum	é o nome sem o prefixo de tipo do correspondente declarador de enumeração .
Tema	complementa o significado da enumeração para denotar precisamente o significado do valor da constante.

Exemplo em C++:

```
enum JAN_tpCores
{
   JAN_CoresAzul ,
   JAN_CoresVermelho ,
   JAN_CoresVerde ,
   JAN_CoresAmarelo
}
```

Exemplo em C:

```
typedef enum
{
   JAN_CoresAzul ,
   JAN_CoresVermelho ,
   JAN_CoresVerde ,
   JAN_CoresAmarelo
} JAN_tpCores
```

Anexo B Tabelas de componentes de nome padronizados

Este anexo contém um exemplo de componentes padronizados dos nomes.

Prefixos de domínio padronizados

Identificadores de domínio – escopo – com significado específico

st_	encapsulado, *static*
m_	membro (constante, variável ou tipo) público de uma classe
mpv_	membro privado de uma classe
mpr_	membro protegido de uma classe
pv_	reservado para método privado, não deve ser utilizado
pr_	reservado para método protegido, não deve ser utilizado
vazio	método ou variável local

Identificadores de domínio definidos

Identificadores de prefixos de tipo padronizados

Identificadores de tipos primários

b	tipo de dado usualmente representado por 8 bits. Em alguns casos pode representar valores com sinal ou sem sinal (***byte***).
c	caractere, usado como tal e não como um inteiro (***char***).
C	classe. Note que este prefixo de tipo é redigido em letras maiúsculas. O restante do prefixo de tipo é o identificador de prefixo de tipo da classe (***class***).
d	ponto flutuante de 64 bits (***double***).
dh	data e hora
dt	data
f	flag, usualmente um valor enumerado.
f	ponto flutuante. O tamanho do valor depende da arquitetura. A ambigüidade com *flag* é resolvida por contexto (***float***).
fn	variável referindo uma função
fp	ponteiro *far*. Evite o uso de *fp*. Defina o tamanho do ponteiro somente quando for mandatório devido a particularidades da biblioteca ou API utilizada. Ver *p*, *np*, *hp*, *fp*
h	handle
hp	ponteiro *huge*. Evite o uso de *hp*. Defina o tamanho do ponteiro somente quando for mandatório devido a particularidades da biblioteca ou API utilizada. Ver *p*, *np*, *hp*, *fp*
hr	hora
i	inteiro com sinal. Dependendo da arquitetura ou do compilador tem 16, 32 ou 64 bits. Ver *b*, *w*, *l*, *i* e *u* (**int**).
id	identificador, tipicamente um valor *unsigned*
ix	índice de um vetor. É utilizado também para referenciar elementos sucessivos relativos a um ponteiro. Tipicamente um valor *int*
l	tipo de dado usualmente representado por 32 bits (***long***).
n	contador. Tipicamente um *int*, utilizado estritamente para contar elementos.

np	ponteiro *near*. Evite o uso de *np*. Defina o tamanho do ponteiro somente quando for mandatório devido a particularidades da biblioteca ou API utilizada. Ver *p, np, hp, fp*
o	deslocamento (*offset*) em bytes, da origem de um elemento relativo a um ponteiro ou referência. Para poder ser utilizado o deslocamento, o ponteiro deve ser do tipo *char* *. Tipicamente um valor *int*
p	ponteiro. O tamanho depende da arquitetura ou do compilador. Evite utilizar ponteiros com tamanho definido, ex. np - *near*, fp - *far* e hp *huge*. O restante do prefixo de tipo determina o tipo do elemento para o qual aponta o ponteiro. Ver *p, np, hp, fp*
pvt	ponteiro para a origem de um vetor. Ver ponteiro e vetor
r	referência, ver ponteiro
rvt	referência para a origem de um vetor. Ver ponteiro e vetor
s	string de caracteres, sem terminador
sz	string de caracteres, terminado com zero
tag	denota que o nome define um rótulo (tag) de construtor de tipo. Pode ocorrer em declarações `struct`, `union`, e `enum` ao programar em C.
tp	denota que o nome define um tipo. Ocorre em nomes de `struct`, `union`, `enum` ou `typedef`. Ao programar em C, o construtor de tipo deverá ser precedido de `typedef` e o nome do tipo estará ao final da declaração.
u	inteiro sem sinal (***unsigned***). Dependendo da arquitetura ou do compilador tem 16 32 ou 64 bits. Ver *b, w, l, i* e *u*.
vt	vetor (*array*). Implicitamente corresponde a um ponteiro constante para a origem de um vetor. Vetor deve ser utilizado como prefixo.
w	tipo de dado usualmente representado por 16 bits. Em alguns casos pode representar valores com sinal ou sem sinal (***word***).

Abreviações e palavras padronizadas

Substantivos e adjetivos

Ant	elemento anterior (ponteiro)
Buffer	área de armazenamento para trabalho. Ex. registro de arquivo em memória real; cópia de um elemento de lista em uma variável local.
Corr	elemento corrente (ponteiro)
Dim	dimensão, tamanho declarado. É tipicamente um componente do nome de uma constante inteira utilizada para declarar um vetor ou array.
Erro	mensagem de erro. Ver mensagem. Note a diferença de redação e significado do sufixo de tipo Err
Lim	limite. valor máximo permitido. Em algumas situações o valor da dimensão declarada é maior do que o maior índice permitido. Ex. considere o tamanho (*strlen*) máximo que um *string* pode ter, em um *string* terminado com zero, este valor limite é pelo menos uma unidade menor que a dimensão.
Lst	lista (ponteiro para origem da lista)
Max	valor limite superior. Deve ser utilizado para registrar o maior valor de uma coletânea de valores. Não use para denotar a dimensão de um vetor ou *string*. Para estes utilize *Dim* ou *Lim*.
Min	valor limite inferior
Msg	mensagem (*char[ ]*) tipicamente contém marcadores de campo para a inclusão de valores ao montar, ou exibir a mensagem.
Nome	identificador, seqüência de caracteres de um elemento (*char[ ]*).
Num	número, contagem, de elementos (*int*).
Org	origem, ex. origem de uma lista, coordenada origem de uma janela
Pag	página

Prox	elemento a seguir (ponteiro)
Tam	tamanho (*unsigned*)

Verbos identificadores de funções e métodos

Abrir	estabelecer direito de acesso a um arquivo ou serviço. Ex. AbrirJan, AbrirArqAlunos.
Alocar	tornar disponível espaço de armazenamento, ou meio para comunicação.
Calcular	efetuar um cálculo genérico. O restante do tema deve identificar o que será calculado. Utilize sempre Calcular e nunca Computar.
Copiar	gerar uma duplicata de um elemento, tornando-a disponível em um local destino. O elemento origem não é afetado pela cópia.
Criar	criar um novo elemento, objeto, ou valor de chave única.
Desalocar	eliminar espaço disponível ou meio de comunicação alocado.
Desmarcar	eliminar uma marca de um objeto. Evite a palavra *resetar*, ela não existe em português. Exemplos, DesmarcarBloqueado. Ver Marcar.
Destruir	eliminar um elemento da memória ou de um arquivo do sistema de arquivos. A diferença entre Destruir e Excluir é que a exclusão retira um elemento de um conjunto, mas não elimina o espaço de memória ocupado.
Excluir	eliminar um elemento de um conjunto. Evite a palavra *Deletar*, ela não existe em português. A palavra *Delir* existe, porém não é de uso corriqueiro.
Exibir	visualizar no terminal do usuário
Fechar	desfazer o direito de acesso a um arquivo ou serviço.
Gravar	gravar um registro ou dado em arquivo.
Ler	ler um registro ou dado para a memória. Poder ser utilizado para ler teclado ou disco.
Marcar	atribuir uma marca a um objeto. Evite a palavra *setar*, ela não existe em português. Em adição o verbo inglês *to set* tem um número muito grande de significados, cada qual correspondendo a uma palavra diferente em português. Exemplos MarcarUsado, MarcarAlterado, MarcarBloqueado. Ver Desmarcar
Montar	gerar um valor composto. Por exemplo, substituir os marcadores contidos em uma mensagem por valores fornecidos.
Mover	transferir um elemento de um local origem para um destino. O elemento deixa de existir no local origem. Se o valor origem deve permanecer, utilize Copiar.
Obter	converter um identificador no valor correspondente
Ordenar	classificar segundo uma chave
Posicionar	ajustar o valor corrente para uma determinada posição, ou torna corrente um determinado elemento em um conjunto de elementos. Pode ser utilizado, por exemplo, com cursores e estruturas de dados.
Procurar	procurar um registro contendo um valor ou chave
Registrar	atribuir um valor permanente. Este valor pode durar a sessão de uso do programa, ou até ultrapassá-la. Nesse caso será gravado em um arquivo. A diferença entre registrar e gravar é que gravar é sempre uma operação atômica envolvendo valores e chaves já conhecidos, enquanto que registrar pode requerer algum cálculo ou um conjunto de operações de saída.
Selecionar	Escolhe zero ou mais elementos de um conjunto de elementos.

Sufixos de tipo padronizado

Assinc	a função é assíncrona.
Err	indicador de erro. Err pode ser um valor qualquer, por exemplo -2. Ocorre usualmente em uma constante – ERR –, indicando que se uma variável contém este valor terá ocorrido um erro de processamento.

Nil	elemento inexistente ou indefinido. Nil pode ser valor qualquer, por exemplo -1. Ocorre usualmente em uma constante – NIL –, indicando um valor ainda não definido ou não disponível.
Null	elemento nulo. Null tem sempre o valor zero. Ocorre usualmente em uma constante NULL –, indicando um valor ainda não definido ou não disponível.
Sinc	a função é síncrona.

Bibliografia

A estrutura da presente norma segue de perto a estrutura encontrada em

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

A discussão relativa à notação húngara, bem como os identificadores padronizados para Windows, foram retirados do material encontrado nos CD-ROM's de apoio ao desenvolvedor de aplicações Windows, e no CD-ROM do compilador Visual C++, tornados disponíveis pela Microsoft. Uma introdução a este assunto pode ser encontrado também em:

Petzold, C.; *Programando para Windows*; São Paulo; Makron; 1992

Os conceitos de módulo de definição e de módulo de implementação são encontrados em:

Wirth, N.; *Programming in Modula 2*; Springer; Berlin; 1983

Uma visão geral de projeto de programas pode ser encontrada nos capítulos que tratam de projeto estruturado de:

Ambiente de Engenharia de Software Talisman, Manual do Usuário; Staa Informática; Rio de Janeiro; 1993

Uma versão preliminar desta norma, voltada para programas C, foi utilizada para o desenvolvimento de Talisman. Encontra-se descrita em:

Staa, A.v.; *Convenções para nomes de elementos de programas C*; Relatório técnico; Departamento de Informática, PUC-Rio; 1993.