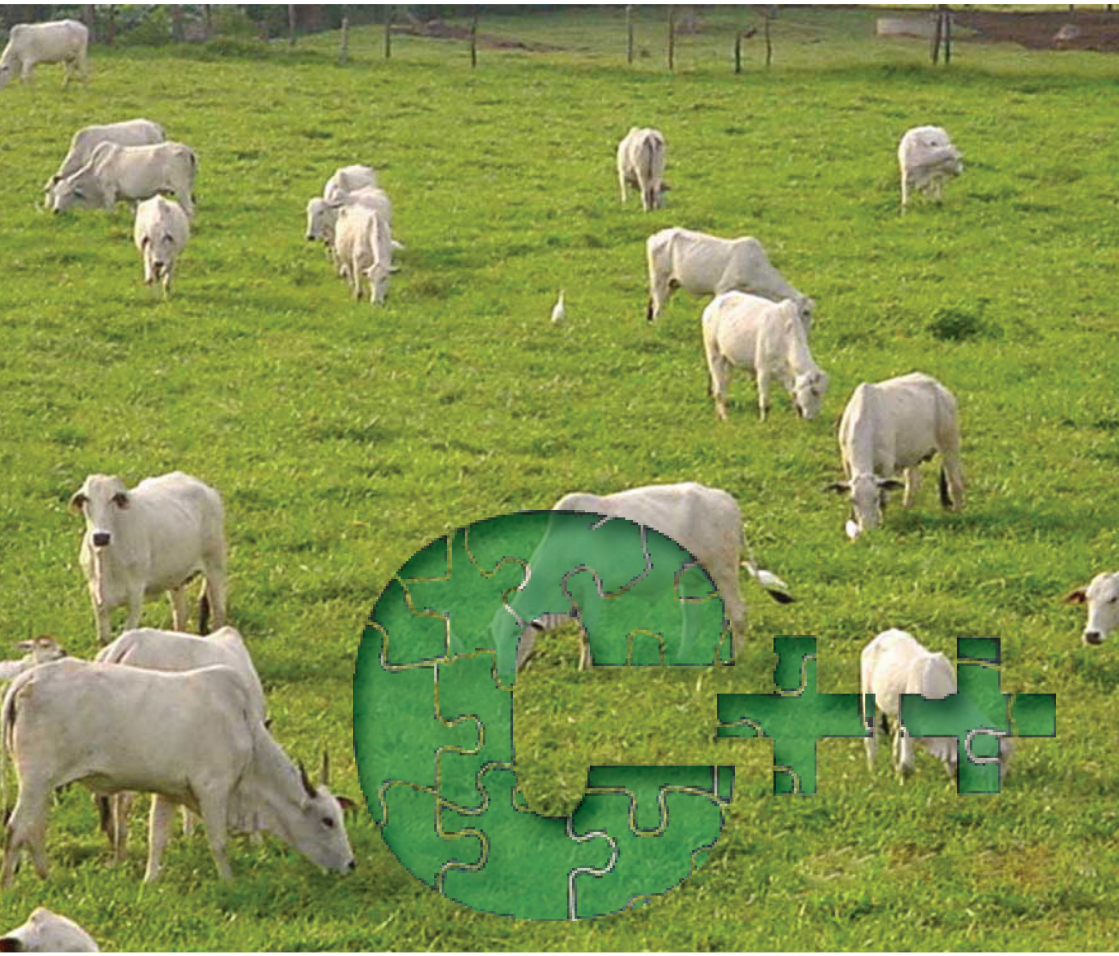
Documentos

ISSN 1677-9274
Dezembro, 2012
124

Proposta de padronização de código-fonte C++ no escopo do projeto Pecus



Embrapa

ISSN 1677-9274
Dezembro, 2012

Empresa Brasileira de Pesquisa Agropecuária
Embrapa Informática Agropecuária
Ministério da Agricultura, Pecuária e Abastecimento

Documentos 124

Proposta de padronização de código-fonte C++ no escopo do projeto Pecus

Adauto Luiz Mancini

Embrapa Informática Agropecuária
Campinas, SP
2012

Embrapa Informática Agropecuária
Av. André Tosello, 209 - Barão Geraldo
Caixa Postal 6041 - 13083-886 - Campinas, SP
Fone: (19) 3211-5700 - Fax: (19) 3211-5754
www.cnptia.embrapa.br
sac@cnptia.embrapa.br

1ª edição on-line 2012

Dados Internacionais de Catalogação na Publicação (CIP)
Embrapa Informática Agropecuária

Mancini, Adauto Luiz.
Proposta de padronização de código-fonte C++ no escopo do projeto Pecus / Adauto Luiz Mancini. - Campinas : Embrapa Informática Agropecuária, 2012.
24 p. : il. - (Documentos / Embrapa Informática Agropecuária , ISSN 1677-9274 ; 124).

1. Linguagem de programação C++. 2. Padronização de código-fonte. 3. Projeto Pecus. I. Embrapa Informática Agropecuária. II. Título. III. Série.

CDD (21. ed.) 005.13

Autor

Adauto Luiz Mancini

Mestre em Ciência da Computação e Matemática Computacional
Pesquisador da Embrapa Informática Agropecuária
Av. André Tosello, 209, Barão Geraldo
Caixa Postal 6041 - 13083-886 - Campinas, SP
Telefone: (19) 3211-5771
e-mail: adauto.mancini@embrapa.br

Apresentação

A adoção de linguagem comum entre os membros de uma equipe é essencial para o sucesso de um projeto de engenharia de software. Esta linguagem comum é facilitada com o uso de uma padronização sintática, pois esta estabelece um protocolo de comunicação, facilitando o reúso dos objetos criados e nomeados de acordo com o padrão. A padronização de código-fonte foi o primeiro processo escolhido pelos membros do Laboratório de Matemática Computacional(LabMaC) envolvidos no projeto componente do projeto MP1 Pecus.

Neste trabalho, os autores propõem um padrão de codificação para a linguagem C++ tomando como base as diretrizes da comunidade “possibility”.

Kleber Xavier Sampaio de Souza
Chefe-Geral
Embrapa Informática Agropecuária

Sumário

Proposta de padronização de código-fonte C++ no escopo do projeto Pecus

Adauto Luiz Mancini

Convenção, siglas

Esta seção tem por objetivo apresentar uma notação para auxiliar na padronização deste texto e explanação de siglas e conceitos frequentes.

Notação

negrito: enfatizar fortemente o texto grifado do texto corrente.

negrito itálico: enfatizar moderadamente o texto grifado do texto corrente.

itálico: enfatizar fracamente o texto grifado do texto corrente.

[<referência>]: referência bibliográfica, podendo ser um acrônimo para as referências mais usadas, ou uma referência a um trabalho ou link. Com menor frequência pode ter outros significados, como uma referência a uma tabela, ilustração ou a algum ponto do texto geral.

“<texto>”: texto copiado ou traduzido na íntegra, com poucas modificações.

Ex: “<tradução livre>”[<referência>] simboliza a substituição de um texto contendo uma tradução livre, escrito entre aspas, seguido de uma referência (provavelmente uma indicação à referência bibliográfica ou autoria do texto original traduzido. Assim no texto pode aparecer:

“Tente limitar seu uso de abreviações em nomes simbólicos.”[gnu]

As regras de codificação são escritas usando a notação apresentada na Tabela 1.

Tabela 1. Notação para regras de codificação.

Notação	Conteúdo	Exemplo
►<recomendação >	Regra ou recomendação sobre algum aspecto de codificação a ser seguido.	► O código-fonte deve ser escrito em inglês.
± <exceção>	Enfraquecimento da recomendação, justificando casos de exceção do uso da regra ou transferindo para o programador a escolha pelo uso ou não uso da recomendação.	**±** Nomes e termos regionais sem correspondentes em inglês, ou de pessoas e localidades devem ser escritos no idioma de contexto do termo.
■ <explicação>	Justificativa para a adoção da regra ou da recomendação.	■ Atualmente inglês é a linguagem padrão para comunicação internacional.
<exemplo>	Código-fonte em C++ contendo exemplo	*enum* Enu_Colors {BLACK, BLUE};

Siglas e acrônimos

[fp]: fonte padrão(fp) de informação, no caso o padrão estabelecido em[1]

LabMaC: Laboratório de Matemática Computacional[2] situado na Embrapa Informática Agropecuária[3]

MP1 Pecus: projeto Pecus do Macroprograma 1 da Embrapa

[1] Disponível em: <http://www.possibility.com/Cpp/CppCodingStandard.html#names.
[2] Disponível em: <http://cnptia.embrapa.br/content/matematica-computacional-labmac.html>.
[3] Disponível em: <http://cnptia.embrapa.br/>.

Objetivo

Apresentar uma proposta de padronização de código-fonte escrito na linguagem de programação C++, validando seu uso na implementação do *framework* de suporte à simulação MacSim (“simulador da matemática computacionas”) dentro das atividades do projeto Pecus, e adequando a especificação em função de seu uso prático durante o desenvolvimento do *framework*.

Contexto

O uso de padronização permite a criação ou o uso de protocolos que facilitam a comunicação entre os membros de uma equipe e reúso dos objetos[4] criados com o uso da padronização. A padronização de código-fonte foi o primeiro processo escolhido pelos membros do LabMaC envolvidos no projeto componente do projeto MP1 Pecus. Os motivos da escolha desse processo foram:

- conseguir desenvolver software com qualidade, eficiência e reúso;
- a existência de vasta literatura sobre o assunto;
- evitar que cada membro da equipe, permanente ou temporário, exerça sua preferência individual de desenvolvimento, possivelmente com pouca ou nenhuma organização, gerando uma grande quantidade de formatos, prejudicando uma forma comum de entendimento dos produtos gerados ou em desenvolvimento;
- rápida inclusão de novos membros na equipe com relação ao entendimento e à produção de código-fonte;
- aquisição de conhecimento na escolha ou desenvolvimento de padrões, que poderá ser aplicado posteriormente a outros processos.

[4] Que no LabMaC são código-fonte, documentos, programas de computador, processos.

Dentro desse contexto, este documento trata apenas da padronização de código-fonte, especificamente escrito na linguagem de programação C++. A codificação é apenas uma das atividades do processo de desenvolvimento de software. Posteriormente, pretende-se gerar diretrizes para outros processos. O estabelecimento dessas diretrizes tem baixa prioridade, desejando-se que elas sejam subprodutos complementares criados em paralelo às atividades diárias, sem prejudicar a produção. Ainda que tais diretrizes tenham como resultado desejado um aumento na qualidade dos produtos, o tempo necessário para sua formulação não foi previsto nas atividades programadas, de modo que tais documentos são vistos como um bônus dentro das atividades do grupo de trabalho. Essas diretrizes, para serem consolidadas, precisam ser validadas pelo uso após sua formulação, num processo de melhoramento contínuo até sua aceitação.

Introdução

Não há um padrão de codificação para a linguagem C++ recomendado por um órgão internacional de padronização[5], mas existem diversos padrões propostos por empresas (ex: Google) e organizações (ex: GNU). O padrão proposto neste trabalho é baseado principalmente nas diretrizes da comunidade *possibility*[6] com possíveis inclusões de algumas diretrizes do padrão recomendado pela Google[7], pela geosoft[8] e pela GNU[9], além de adaptações próprias sugeridas pelo autor ou pela equipe de revisão.

[5] Ao menos não foi encontrado na pesquisa bibliográfica.
[6] Disponível em: <http://www.possibility.com/Cpp/CppCodingStandard.html>.
[7] Disponível em: <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>.
[8] Disponível em: <http://geosoft.no/development/cppstyle.html#introduction>.
[9] Disponível em: <http://www.gnu.org/prep/standards/standards.html>.

Idioma

► O código-fonte deve ser escrito em inglês.

fileName; // NOT: nomeArquivo

± Nomes e termos regionais sem correspondentes em inglês, ou de pessoas e localidades, devem ser escritos no idioma de contexto do termo.

± Quando o implementador não se sentir confortável, no caso de comentários de texto corrido (explicação de variáveis e interfaces de funções, algoritmos, etc), adicionar também uma explicação no idioma local, para que posteriormente o texto em inglês possa ser refinado. Isso é importante porque um texto escrito por alguém com pouco domínio da linguagem pode ser confuso e haver perda de informação ou significação errada do conteúdo desejado.

■ Atualmente inglês é a linguagem padrão para comunicação internacional.

Nomenclatura

A escolha adequada e consistente de nomes é o aspecto mais importante da padronização de código, uma vez que deve permitir uma compreensão fácil e sem (ou com reduzida) ambiguidade aos humanos dos elementos representados no código. Esses elementos podem ser dados ou procedimentos. Elementos de dados vão desde a declaração de tipos de dados (constantes, enumerações, registros, estruturas, classes, etc) à declaração das variáveis a serem usadas no código (variáveis e *arrays* (vetores), variáveis auxiliares, instâncias de agregações de dados como estruturas ou objetos de classe) e funções (sejam métodos de classes ou não).

“As regras de consistência mais importantes são aquelas que governam nomeação. O estilo de um nome informa-nos imediatamente sobre o tipo ou finalidade da entidade textual referenciada: um tipo, uma variável, uma função, uma constante, uma macro, etc, sem fazer-nos procurar pela declaração da entidade. O engenho de casamento de padrões de nossos cérebros funciona bem usando esse tipo de regras de nomeação.”[google]

“Um nome é o resultado de um processo de reflexão longo e profundo sobre a ecologia em que o alvo reside. Somente um programador que entende o sistema como um todo pode criar um nome que se encaixa adequadamente com o sistema. Se o nome é apropriado, tudo se encaixa junto naturalmente, relações são claras, o entendimento é derivável, e o raciocínio a partir de expectativas comuns humanas funciona como esperado.”[possibility]

Tipos de dados

► Nomes representando tipos devem começar com um prefixo dependente do tipo (Tabela 2), seguido dos termos intercalados por tipo de caixa (primeira letra maiúscula e demais minúsculas).

Tabela 2. Prefixos para tipos.

Classe	Cls_
Estrutura	Str_
União	Uni_
Enumeração	Enu_

```
class Cls_Rectangle {
    int width, height;
  public:
    void Values (int width_arg,int height_arg);
    int area () {return (width*height);}
};

struct Str_Product {
  int weight;
  float price;
};


enum Enu_Colors {BLACK, BLUE};

Cls_Rectangle rectangle; //Sabe_se que é uma variável de tipo clas-
se pelo prefixo

Str_Product product; //Sabe_se que é uma variável de tipo estrutura
pelo prefixo
```

```
Enu_Colors color; //Sabe_se que é uma variável de tipo enumeração
pelo prefixo
```

■ O uso de um prefixo permite saber na declaração da variável o tipo de agregação da informação (classe, estrutura, etc), em adição à função informada no nome (Rectangle). Esta forma mimetiza a declaração de variáveis de tipos básicos em C++.

```
int integerNumber; //tipo básico
Cls_Rectangle rectangle; //imitação da declaração do tipo básico
```

O uso de caixas maiúsculas e minúsculas para diferenciar nomes de variáveis de nomes de tipos ou de funções é prática comum na comunidade C++.

Constantes e enumerações

► Nome de constante ou de valor de enumeração deve ser escrito com letras maiúsculas. No caso do nome conter mais de uma palavra, estas devem ser separadas por sublinhado.

```
#define PI 3.14
enum Enu_ColoredFruits {YELLOW_BANANA, ORANGE_ORANGE, RED_APPLE};
```

Variáveis

1. Por um ou mais prefixos (Tabela 3) relacionados ao modelo matemático (caso se aplique), começando com letra minúscula e os demais termos começando com letra maiúscula, seguindo ordem de prioridade;
2. a função da variável começando em minúsculo e os demais termos começando com maiúsculo;
3. um ou mais sufixos (Tabela 4)relacionados à implementação (caso se aplique), começando em minúsculo e os demais termos começando com maiúsculo, seguindo ordem de prioridade.

Tabela 3. Prefixos para variáveis.

1) entrada (input)	inp_
2) saída (output)	out_
3) estado (compartiment)[10]	cpt_
4) estado de transição (transition)[11]	trs_
5) auxiliar	aux_
6) mínimo	min_
7) média (average)	avg_
8) máximo	max_
9) contador (count)	cnt_
10) chave (key)	key_
11) índice (index)	idx_

O prefixo *auxiliar* (aux_) refere-se a um cálculo intermediário do modelo matemático.

Tabela 4. Sufixos para variáveis.

1) constante[12]	_ctt
2) argumento[13] (de função)	_arg
3) ponteiro	_ptr
4) Alias ou referência	_ref
5) Encapsulamento (hidden)	_hid
6) Variável estática (static)	_stt

```
//Valor de entrada de carboidratos diários na ração em miligramas
com valor constante:
const float inp_dailyCarbohidratesInFeedInMilligrams_ctt = 456.0;
```

[10] Um compartimento (variável de estado) refere-se a um valor de comportamento dinâmico (em função do tempo) de interesse do sistema sendo estudado. Terminologia de modelos dinâmicos em simulação de sistemas.

[11] Um estado de transição refere-se a um (ou a um conjunto de) estado que em função de um evento (valor) de entrada dispara uma regra de transição adequada e muda o estado do sistema. Terminologia de automatos finitos de computação.

[12] Variáveis declaradas com a palavra-chave *const*, não referem-se à macro *#define*.

[13] O sufixo _arg deve ser usado apenas no escopo interno da função onde é declarado, evite seu uso na chamada da função.

```
//Variável auxiliar do modelo, porcentagem diária de carbohidratos
na ração:
float aux_dailyPercentageOfCarbohidratesInFeed;

//Compartimento e porta de saída para ganho total de peso diário em
miligramas:
float outCpt_totalWeightGainInMilligramsOfDay ;

//Compartimento com valor médio do ganho diário de peso em gramas
por animal no lote 53:
float cptAvg_totalWeightGainInMilligramsOfDayInBatch53;

//Estado de transição, situação reprodutiva da vaca, ponteiro cons-
tante para argumento de função:
const int* trs_reprodutiveStateOfCow_cttArgPtr;
```

■ Prática comum na comunidade de desenvolvimento C++, iniciar nomes de variáveis com minúscula permite diferenciá-las de tipos ou funções. O uso de prefixos e sufixos preestabelecidos permite fornecer informações adicionais de tipo ou comportamento da variável. A opção escolhida foi priorizar informações relacionadas ao contexto do modelo matemático na forma de prefixo, seguido pela informação da funcionalidade da variável propriamente dita, e por último (sufixo) o contexto da variável em função do tipo de comportamento na implementação. Alguns padrões de codificação (google,gnu) recomendam o uso de sublinhado como separador de palavras. Como essa prática aumenta o tamanho do nome no código e também o tempo e esforço de digitação, preferiu-se o padrão de separar as palavras por caixa alta e baixa, reservando o sublinhado apenas para separar sequências de prefixos ou sufixos, quando existirem.

```
MyExcitingClass my_exciting_local_variable ;
MyExcitingClass myExcitingLocalVariable; //mais curto e fácil de
digitar
```

►Nomes de tipos e variáveis tipicamente deveriam ser substantivos ou adjuntos adnominais(adjetivos, locuções adjetivas, artigos, pronomes adjetivos, numerais adjetivos):

```
string fileOpener;
int cnt_errors;
```

■ Os verbos que têm uma ação associada são mais apropriados para a definição de nomes de funções.

■ Não se preocupe em salvar espaço horizontal uma vez que é muito mais importante tornar seu código imediatamente entendível para um novo leitor.

Exemplos de nomes bem escolhidos:

```
int cnt_errors;          // Bom
int cnt_completedConnections;  // Bom
```

Nomes pobremente escolhidos usam abreviações ambíguas ou caracteres arbitrários que não transmitem significado:

```
int n;              // Ruim: pouco significativo
int nErr;             // Ruim: abreviação ambígua
int nCompConns;         // Ruim: abreviação ambígua
```

►"Nomes de variáveis locais podem ser curtos, porque são usados somente dentro de um contexto, onde comentários (presumivelmente) explicam seu propósito."[gnu]

```
int i,j; //contador usado para incrementos em laços
for (i=0; i<10; i++) {j = i*2;}
```

►Variáveis genéricas devem ter o nome semelhante ao nome de seu tipo.

```
Cls_Database database;
Cls_Database* database_ptr;
```

■ "Reduza a complexidade reduzindo o número de termos e nomes usados. Também facilita deduzir o tipo dado apenas pelo nome da variável.

Se por alguma razão esta convenção parece não se encaixar é uma forte indicação de que o nome do tipo está ruim.

Variáveis não genéricas tem um papel. Estas variáveis geralmente podem ser nomeadas pela combinação do papel e tipo:

```
Point startingPoint, centerPoint;
Name loginName;” [geosoft]
```

►“Abreviações: não use abreviações a menos que elas sejam extremamente bem conhecidas fora do seu projeto.

```
int cnt_dnsConnections;  // Muitas pessoas sabem o que “DNS” signi-
fica.
int wgcConnections;  // Apenas seu grupo de trabalho conhece a si-
gla wgc
int pcReader;        // Muitas coisas podem ser abreviadas por
“pc”.
```

Nunca abrevie deixando letras de fora:

```
string chemistryDepartmentManager; // Bom
int chemDeptManager; // Ruim
```

►“Tente limitar o uso de abreviações em nomes simbólicos.

■ Fazer umas poucas abreviações é aceitável, mas explique o que elas significam e então use-as frequentemente, mas não use grandes quantidades de abreviações obscuras. ”[gnu]

►“Abreviações e acrônimos não devem ser em maiúscula quando usados como um nome.

```
ExportHtmlSource(); // Não use: ExportHTMLSource();
OpenDvdPlayer(); // Não use: OpenDVDPlayer();” [geosoft]
```

►“Variáveis globais devem sempre ser referenciadas explicitamente usando o operador :: [geosoft]

```
::mainWindow.Open()
::applicationContext.GetName()
```

►Variáveis privadas ou protegidas de classe devem ter o sufixo _hid

```
class Cls_SomeClass {
  private:
    int length_hid;
}
```

■ A indicação explícita no nome de que a variável é estática, funciona como um lembrete ao programador de que a variável tem um único valor para todas as instâncias da classe.

►O prefixo cnt_ deve ser usado para variáveis representando um número de objetos.

■ A indicação deixa claro que a variável representa o tamanho de uma população.

►Variáveis booleanas devem ser iniciadas por um verbo, quando adequado.

```
isSet = IsSet();
hasLicense = HasLicense();
mustDestroy = MustDestroy();
```

Funções

►Nomes de funções devem começar com um verbo imperativo (expressa comando) em maiúscula, sem prefixos ou sufixos. Nomes de funções (método que retornam um valor) devem ter um nome representativo do que retornam. Procedimentos (métodos void, que não retornam valor) devem ter um nome representativo da ação ou processo executado pelo método. Na definição da função os parâmetros devem ter sufixo _arg.

```
void OpenFile();
isOperationValid = CheckFlight (flightNumber, cardCreditOfClient);
aux_PercentOfFat = ComputeFatIndice (totalWeightInOnces, ageInWe-
eks);

void ComputeGeometricCenter (Cls_Rectangle rectangle_arg, float x_
arg, float y_arg);
```

```
ComputeGeometricCenter (rectangle, x, y); //Chamada da função sem o
sufixo _arg
```

■ Funções têm o objetivo de processar dados e têm um caráter ativo de executar ações, que são melhor representadas por verbos. Embora nomes de funções possam ser diferenciados de outros nomes porque possuem parênteses. O uso do sufixo _arg na definição da função permite saber dentro do corpo da função quando uma variável é um argumento da função, útil quando o corpo tem tamanho grande e muitas variáveis são declaradas, ou quando são declaradas variáveis internas com nomes semelhantes aos nomes dos argumentos.

►Funções de acesso a propriedades protegidas de classe devem ter o mesmo nome da propriedade, começando com maiúscula. Quando a propriedade contiver prefixos ou sufixos, não utilizar o caracter sublinhado(_) no nome da função, para manter a padronização da regra de nomeação de funções. Quando a propriedade for encapsulada (sufixo _hid), não usar hid no nome da função de acesso.

```
class SomeClass {
  public:
    int Lenght()           //Não usar: GetLenght
      {return length_hid;
      }
    void Lenght(int value_arg)      //Não usar: SetLenght
      {length_hid = value_arg;
      }
    const float * AreaPtr()
      {return area_ptr;
      }
  private:
    int length_hid;
    float * area_ptr;

}
```

■ As assinaturas diferentes das funções são suficientes para diferenciar se a variável está sendo lida ou escrita, sem necessidade de Get/Set no nome da função. Definir a função com o mesmo nome da propriedade diminui a quantidade de nomes a serem lembrados.

Arquivos fontes

►Uma classe deve ser declarada em um arquivo de interface (.h, .hpp) e definida em um arquivo fonte (.c++, .C, .cc , .cpp) com os arquivos tendo o mesmo nome da classe.

```
class Cls_Rectangle: Cls_Rectangle.h, Cls_Rectangle.cpp
```

■ Isto facilita localizar a definição de uma classe que foi referenciada em outro arquivo de código-fonte.

►As seções de acessibilidade de uma classe devem ser explícitas e seguir a ordem: public, protected e private.

■ Um usuário comum da classe lê apenas a interface pública, no início. As seções encapsuladas são de interesse apenas dos programadores responsáveis por tais métodos.

►Arquivos de cabeçalho precisam conter um guarda de include.

```
#ifndef COM_COMPANY_MODULE_CLASSNAME_H
#define COM_COMPANY_MODULE_CLASSNAME_H
:
  #endif // COM_COMPANY_MODULE_CLASSNAME_H
```

■ Evita erros de compilação ao se tentar incluir duas ou mais vezes o mesmo arquivo de cabeçalho causando erro de duplicação de declaração.

►Tipos que são locais a um arquivo somente devem ser declarados dentro do referido arquivo.

■ Reforça encapsulamento da informação.

Identação

►Usar 5 espaços para identação.

■ Facilita visualizar comandos condicionais com blocos senão.

►Alinhe { e } verticalmente.

```
{//declaração de variáveis com uso em mais de um lugar no bloco
  int weight, height, x; //variáveis obvias que não precisam expli-
cação
  Type1 complexVariable1; //comentário contendo explicação da va-
riavel não obvia
  Type1 complexVariable2; // comentário contendo explicação da va-
riavel não obvia
  :
  Type5 complexVariable19; // comentário contendo explicação da va-
riavel não obvia

if (condition)
    {command 1;
    }
else
    {command 2;
    }

if (very_long_condition1
&& second_very_long_condition)
    {//variavel que serão usadas apenas em um comando ou subbloco
  //são declaradas antes do local de uso
  Type3 veryLocalVariable;
      …
  veryLocalVariable = … ;
    }
else if (...)
    {
      ..
    }
for (int i=0; i<height; i++)
    {
    }


  {//se criar novo bloco de escopo, comentar função do novo bloco

    //declaração de variáveis óbvias
    //declaração de variáveis com uso em mais de um lugar no bloco

    //comandos locais do bloco
  }

  while (condition1
  && second_very_long_condition)
      {
      }
}
```

■ Alinhando verticalmente as chaves, por questão de simetria, fica mais fácil visualizar a estrutura do código. Quando houver aninhamento de muitas chaves, com o fechamento distante da abertura, comentar o fechamento com o comando correspondente da abertura.

```
if (condition1)
    {if (condition2)
      {if (condition3)
        {
        ...
      } //if (condition3)
    } //if (condition2)
  } //if (condition1)
```

Embrapa

Informática Agropecuária

Ministério da
Agricultura, Pecuária
e Abastecimento

GOVERNO FEDERAL
BRASIL
PAÍS RICO É PAÍS SEM POBREZA

CGPE 10233