Estilos de Codificação

Prof. André Grégio

Um estilo de codificação estabelece algumas regras para escrever código que ajudam os desenvolvedores a organizar sua produção. É importante seguir um estilo para manter a clareza dos códigos escritos. A indústria em geral escolhe algum estilo de codificação apropriado para adotar, de forma a manter a consistência e facilitar o desenvolvimento conjunto de sistemas e programas complexos. O kernel do Linux tem um estilo de codificação bastante simples e específico que pode servir de exemplo para quem está aprendendo a programar em C.

A seguir, são listadas as principais regras que o estilo de codificação do kernel do Linux [1] estabelece para programação em Linguagem C, adaptadas, quando oportuno, com regras do estilo de codificação do Google [2] (mais voltado para C++, mas ainda assim com regras aproveitáveis em C):

1. **Indentação**: Use TAB com comprimento de 8 caracteres para a indentação. Não use espaços para indentação.

```
if (condicao) {
     minhaFuncao();
}
```

2. **Linhas e Comprimento**: Não se recomenda que cada linha de código ultrapasse 80 caracteres. Se uma linha de código parecer muito longa (veja a parte em **negrito** abaixo), é um bom indício de que a função esteja muito complexa e que você deve considerar refatorá-la (veremos isso futuramente).

```
for (int i = 0; i < conta; i++) {
     var += umaFuncao(argumento1, argumento2, argumento3, arg4,
                      algum_argumento_funcao_ou_estrutura,
                      outro_argumento_grande, argumento7);
```

3. **Comentários**: Use comentários de forma que façam sentido. Eles devem explicar o "porquê" e não o "como". Comentários devem ser escritos antes do código a que se referem.

```
/* Calcula a média, assumindo que a soma não irá ultrapassar o
limite do int e que o total seja diferente de zero */
int media = soma / total;
```

4. **Espaçamento**: Inclua um espaço depois de palavras-chave como "if", "switch", "case", "for", "do", "while". Não coloque espaços ao redor de operadores dentro de parênteses.

```
/* Correto */
if (a > b) {
…
}

/* Não recomendado */
if(a>b){
…
}
```

5. **Chaves**: As chaves de abertura para estruturas de controle ("if", "for", "while", etc.) devem ficar na mesma linha que a instrução. As chaves de fechamento devem começar uma linha nova. Funções tanto podem ter suas chaves de abertura na mesma linha (estilo de codificação do Google), quanto na próxima linha (estilo de codificação do kernel do Linux) no mesmo nível da indentação.

```
/* Correto */
if (a > b) {
     fazAlgo();
}

/* Não recomendado */
if (a > b)
{
     fazAlgo();
}
```

6. **Nomeação de Variáveis**: Prefira nomes de variáveis e funções em minúsculas. Você pode utilizar *underscores* (o caracter "_") para separar palavras em nomes de variáveis, que é a chamada notação *snake case*. Você também pode utilizar a notação *camel case*, na qual a primeira palavra do nome da variável começa com letra minúscula e as demais são juntas, isto é, sem espaços ou qualquer outro separador, e iniciam com a primeira letra em maiúsculo. Mais informações sobre estilos de nomeação em [3]. De qualquer forma, **escolha um estilo de nomeação e seja consistente** ao longo de seu código.

```
/* Notação snake case */
int minha_variavel = 10;

/* Notação camel case */
int minhaVariavel = 10;
```

7. **Operadores e Expressões**: Adicione espaços ao redor dos operadores (exceto operadores de precedência). Não deixe espaços em branco no final das linhas.

```
/* Correto */
a = b + c;

/* Não recomendado */
a=b+c;
```

8. **Estrutura de Programa**: Cada programa deve ter um único propósito. Se houver várias tarefas a serem realizadas, elas devem ser divididas em funções. Um programa pode ser segmentado em várias bibliotecas. Veremos exemplos mais adiante no curso.

```
/* Em vez de fazer: */

int main() {
     /* Código para ler dados */
     /* Código para processar dados */
     /* Código para mostrar dados processados */
     …
}

/* Prefira estruturar melhor seu código: */

void lerDados() {
     /* Código para ler dados */
}

void processarDados() {
     /* Código para processar dados */
}

void mostrarResultados() {
     /* Código para mostrar os dados processados */
}

int main() {
     lerDados();
     processarDados();
     mostrarResultados();
     …
}
```

9. **Funções**: Tente manter as funções pequenas e com um único propósito. Uma função que executa muitas tarefas diferentes é difícil de entender e manter.

```
/* Em vez de fazer uma função assim: */

void fazerTudoComMaisUmPouco() {
     /* Código que faz muitas, mas muitas coisas diferentes */
}

/* Prefira compartimentalizar: */

void fazUmaCoisa() {
     /* Código que faz apenas uma coisa */
}

void fazOutraCoisa() {
     /* Código que faz somente outra coisa */
}

int main() {
     fazUmaCoisa();
     fazOutraCoisa();
     …
}
```

10. **Typedef**: Não use *typedef* neste curso. Você poderá usá-lo mais para frente. Talvez você ainda não saiba o que é *typedef*, e nem deve se preocupar com isso neste momento. Aqui, você só utilizará *structs* (um tópico a ser discutido em aulas futuras) e elas não necessitarão de opacidade, pois todos os seus membros ou elementos serão diretamente acessíveis. Se você não entendeu nada do que está escrito nesse parágrafo (**ainda!**), apenas lembre-se: neste curso, **NÃO USE *typedef***. Seu código ficará mais legível e possíveis usos futuros de *typedef* ficarão mais claros no momento certo.

IMPORTANTE: A legibilidade do código é fundamental. Outros programadores (ou você no futuro) devem ser capazes de entender o que seu código sem dificuldades. Quando estiver em dúvida, tente focar na clareza e simplicidade de seu código. Se um dia você for trabalhar na indústria como desenvolvedor, é altamente provável que um estilo de codificação deva ser seguido… =)

Referências

[1] https://www.kernel.org/doc/html/latest/process/coding-style.html
[2] https://google.github.io/styleguide/cppguide.html
[3] https://www.freecodecamp.org/news/snake-case-vs-camel-case-vs-pascal-case-vs-kebab-case-whats-the-difference/