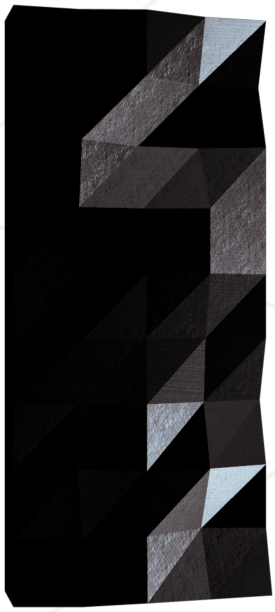

A Norma

Versão 4.1

Sumário: Este documento descreve o padrão aplicável (Norma) na 42. Um padrão de programação define um conjunto de regras a seguir ao escrever um código. A Norma aplica-se a todos os projetos C dentro do Common Core por padrão, e para qualquer projeto onde é especificado.

Conteúdo

Capítulo I

Introdução

A `norminette` é um código Python e open source que verifica a conformidade do seu código-fonte com a Norma. Ela verifica muitas restrições da Norma, mas não todas (por exemplo, restrições subjetivas). A menos que haja regulamentos locais específicos em seu campus, a `norminette` prevalece durante avaliações nos itens controlados. Nas páginas a seguir, as regras que não são verificadas pela `norminette` são marcadas com *(*)*, e podem levar à reprovação do projeto (usando a flag da Norma) se descobertas pelo avaliador durante uma revisão de código.

Seu repositório está disponível em https://github.com/42School/norminette.

Pull requests, sugestões e issues são bem-vindos!

Capítulo II

Por quê ?

A Norma foi cuidadosamente elaborada para suprir diversas necessidades pedagógicas. Aqui estão alguns dos motivos mais importantes por trás das escolhas abaixo:

- Sequenciamento: programar implica dividir uma tarefa grande e complexa em uma série de instruções elementares. Todas essas instruções vão ser executadas em sequência: uma após a outra. Um iniciante, ao começar a criar software, precisa de uma arquitetura simples e clara para seu projeto, tendo o entendimento completo de todas as instruções individuais e da exata ordem de execução. Sintaxes de linguagens crípticas que aparentam executar múltiplas instruções ao mesmo tempo são confusas, funções que buscam abordar múltiplas tarefas misturadas na mesma porção de código são fontes de erros.
  A Norma pede que você escreva trechos simples de código cujas tarefas possam ser entendidas e verificadas facilmente, em que a sequência de execução das instruções não deixa dúvidas. Por este motivo que há o limite máximo de 25 linhas por função, e o porquê de `for`, `do .. while`, ou ternários serem proibidos.

- Estética: enquanto se relaciona com seus colegas durante o processo natural de aprendizado entre pares, e também durante as avaliações entre pares, você não quer gastar tempo decifrando o código deles, mas falar diretamente sobre a lógica daquele trecho de código.
  A Norma pede por uma estética específica, provendo instruções para nomear funções e variáveis, indentação, utilização das chaves, tabulações e espaços em diversos lugares... . Isso vai permitir que você olhe brevemente para o código de outros e o ache familiar, podendo ir direto ao assunto, ao invés de gastar tempo lendo o código antes de entendê-lo. A Norma também se caracteriza como uma marca registrada. Como parte da comunidade 42, você vai poder reconhecer código escrito por outro cadete ou alumni da 42 quando estiver no mercado de trabalho.

- Visão de longo prazo: esforçar-se para escrever um código compreensível é a melhor maneira de administrá-lo. Toda vez que alguém, incluindo você, precisar consertar um bug ou adicionar uma nova funcionalidade, não será necessário gastar tempo tentando entender o funcionamento se você escreveu seu código da maneira correta. Isso vai evitar situações em que trechos de código deixam de ser atualizados apenas por tomarem tempo, o que vai fazer a diferença ao falarmos sobre ter um produto bem sucedido no mercado. Quanto mais cedo aprender, melhor.

- Referências: você pode pensar que algumas, ou todas, as regras inclusas na Norma são arbitrárias, mas nós pensamos cuidadosamente e pesquisamos como elaborá-la. Nós encorajamos fortemente que você pesquise o porquê de funções precisarem ser curtas e possuir apenas uma tarefa, o porquê de nomes de variáveis precisarem ser compreensíveis, o porquê de linhas não poderem extrapolar o limite de 80 colunas de largura, o porquê de uma função não poder receber vários parâmetros, o porquê de comentários serem úteis, etc.

Capítulo III

A Norma

III.1 Denominação

- O nome de um struct deve começar por `s_`.
- O nome de um typedef deve começar por `t_`.
- O nome de um union deve começar por `u_`.
- O nome de um enum deve começar por `e_`.
- O nome de uma variável global deve começar por `g_`.
- Identificadores, como nomes de variáveis, funções e tipos definidos pelo usuário, só podem conter letras minúsculas, dígitos e '_' (snake_case). Nenhuma letra maiúscula é permitida.
- Nomes de arquivos e diretórios só podem conter letras minúsculas, dígitos e '_' (snake_case).
- Caracteres que não fazem parte da tabela ASCII padrão são proibidos, exceto dentro de strings literais e caracteres.
- *(*)* Todos os nomes de identificadores (funções, tipos, variáveis, etc.) devem ser explícitos, ou mnemônicos, devem ser legíveis em inglês, com cada palavra separada por um underscore. Isso se aplica a macros, nomes de arquivos e diretórios também.
- O uso de variáveis globais que não são marcadas como const ou static é proibido e é considerado um erro de norma, a menos que o projeto as permita explicitamente.
- O arquivo deve compilar. Não se espera que um arquivo que não compila passe na Norma.

III.2 Formatação

- Cada função deve ter no máximo 25 linhas, sem contar as próprias chaves da função.
- Cada linha deve ter no máximo 80 colunas de largura, incluindo comentários. Atenção: uma tabulação não conta como uma única coluna, mas como o número de espaços que ela representa.
- Funções devem ser separadas por uma linha vazia. Comentários ou instruções de pré-processador podem ser inseridos entre funções. Pelo menos uma linha vazia deve existir.
- Você deve indentar seu código com tabulações de 4 caracteres. Isso não é o mesmo que 4 espaços, estamos falando de tabulações reais (caractere ASCII número 9). Verifique se seu editor de código está corretamente configurado para obter uma indentação visual adequada que será validada pela `norminette`.
- Blocos dentro de chaves devem ser indentados. As chaves ficam sozinhas em sua própria linha, exceto na declaração de struct, enum, union.
- Uma linha vazia deve estar realmente vazia: sem espaços ou tabulações.
- Uma linha nunca pode terminar com espaços ou tabulações.
- Nunca pode haver duas linhas vazias consecutivas. Nunca pode haver dois espaços consecutivos.
- Declarações devem estar no início de uma função.
- Todos os nomes de variáveis devem ser indentados na mesma coluna em seu escopo. Nota: os tipos já são indentados pelo bloco que os contém.
- Os asteriscos que acompanham ponteiros devem estar colados ao nome da variável.
- Apenas uma declaração de variável por linha.
- Declaração e inicialização não podem estar na mesma linha, exceto para variáveis globais (quando permitido), variáveis estáticas e constantes.
- Em uma função, você deve colocar uma linha vazia entre as declarações de variáveis e o restante da função. Nenhuma outra linha vazia é permitida dentro de uma função.
- Apenas uma instrução ou estrutura de controle por linha é permitida. Ex.: Atribuição em uma estrutura de controle é proibida, duas ou mais atribuições na mesma linha são proibidas, uma nova linha é necessária ao final de uma estrutura de controle, ...

- Uma instrução ou estrutura de controle pode ser dividida em várias linhas quando necessário. As linhas seguintes devem ser indentadas em relação à primeira linha, espaços naturais devem ser usados para cortar a linha, e, se aplicável, operadores devem estar no início da nova linha e não no final da anterior.

- A menos que seja o final de uma linha, cada vírgula ou ponto e vírgula deve ser seguido por um espaço.

- Cada operador ou operando deve ser separado por um - e apenas um - espaço.

- Cada palavra-chave do C deve ser seguida por um espaço, exceto por palavras-chave de tipos (como int, char, float, etc.), assim como sizeof.

- Estruturas de controle (if, while..) devem usar chaves, a menos que contenham uma única instrução em uma única linha.

Exemplo geral:

```
int             g_global;
typedef struct  s_struct
{
    char    *my_string;
    int     i;
}               t_struct;
struct          s_other_struct;

int     main(void)
{
    int     i;
    char    c;

    return (i);
}
```

III.3 Funções

- Uma função pode ter até 4 parâmetros definidos no máximo.
- Uma função que não tem argumentos deve ser explicitamente prototipada com a palavra "void"como o argumento.
- Parâmetros em protótipos de funções devem ser nomeados.
- Cada função deve ser separada da próxima por uma linha vazia.
- Você não pode declarar mais de 5 variáveis por função.
- O retorno de uma função deve estar entre parênteses, a menos que a função retorne nada.
- Cada função deve ter uma tabulação única entre seu tipo de retorno e seu nome.

```
int my_func(int arg1, char arg2, char *arg3)
{
    return (my_val);
}

int func2(void)
{
    return ;
}
```

III.4 Typedef, struct, enum e union

- Como outras palavras-chave do C, adicione um espaço entre “struct” e o nome ao declarar uma struct. O mesmo se aplica para enum e union.
- Ao declarar uma variável do tipo struct, aplique a indentação usual para o nome da variável. O mesmo se aplica para enum e union.
- Dentro das chaves da struct, enum, union, as regras regulares de indentação se aplicam, como em qualquer outro bloco.
- Como outras palavras-chave do C, adicione um espaço após “typedef”, e aplique a indentação regular para o novo nome definido.
- Você deve indentar todos os nomes das estruturas na mesma coluna para seu escopo.
- Você não pode declarar uma estrutura em um arquivo .c.

III.5 Headers - ou arquivos de inclusão

- *(*)* Os elementos permitidos em um arquivo header são: inclusões de headers (sistema ou não), declarações, defines, protótipos e macros.
- Todas as inclusões devem estar no início do arquivo.
- Você não pode incluir um arquivo C em um header ou em outro arquivo C.
- Arquivos header devem ser protegidos contra inclusões duplas. Se o arquivo for `ft_foo.h`, sua macro de proteção deve ser `FT_FOO_H`.
- *(*)* A inclusão de headers não utilizados é proibida.
- A inclusão de headers pode ser justificada no arquivo .c e no próprio arquivo .h usando comentários.

```
#ifndef FT_HEADER_H
# define FT_HEADER_H
# include <stdlib.h>
# include <stdio.h>
# define FOO "bar"

int     g_variable;
struct  s_struct;

#endif
```

III.6 O header da 42 - ou comece um arquivo com estilo

- Todo arquivo .c e .h deve começar imediatamente com o header padrão da 42: um comentário multilinha com um formato especial incluindo informações úteis. O header padrão está disponível nos computadores dos clusters para vários editores de texto (emacs: usando `C-c C-h`, vim usando `:Stdheader` ou `F1`, etc...).

- *(*)* O header da 42 deve conter várias informações atualizadas, incluindo o criador com login e e-mail estudantil (@student.campus), a data de criação, o login e a data da última atualização. Cada vez que o arquivo for salvo no disco, as informações devem ser atualizadas automaticamente.

O header padrão pode não estar automaticamente configurado com suas informações pessoais. Você pode precisar alterá-lo para seguir a regra anterior.

III.7 Macros e Pré-processadores

- *(*)* Constantes de pré-processador (ou #define) que você criar devem ser usadas apenas para valores literais e constantes.
- *(*)* Todo #define criado para burlar a norma e/ou ofuscar o código é proibido.
- *(*)* Você pode usar macros disponíveis em bibliotecas padrão, somente se estas forem permitidas no escopo do projeto em questão.
- Macros multilinha são proibidas.
- Nomes de macros devem estar todos em maiúsculas.
- Você deve indentar diretivas de pré-processador dentro de blocos #if, #ifdef ou #ifndef.
- Instruções de pré-processador são proibidas fora do escopo global.

III.8 Coisas proibidas!

- Você não tem permissão para usar:
  - for
  - do...while
  - switch
  - case
  - goto
- Operadores ternários como '?'.
- VLAs - Arrays de comprimento variável.
- Tipo implícito em declarações variáveis.

```
int main(int argc, char **argv)
{
    int     i;
    char    string[argc]; // This is a VLA

    i = argc > 5 ? 0 : 1 // Ternary
}
```

III.9 Comentários

- Comentários não podem estar dentro do corpo das funções. Comentários devem estar no final de uma linha ou em sua própria linha.
- *(*)* Seus comentários devem estar em inglês e ser úteis.
- *(*)* Um comentário não pode justificar a criação de uma função genérica/faz-tudo ou ruim.

Uma função genérica/faz-tudo ou ruim geralmente possui nomes que não são explícitos, como f1, f2... para funções e a, b, c,... para nomes de variáveis. Uma função cujo único objetivo é evitar a norma, sem um propósito lógico único, também é considerada uma função ruim. Lembre-se de que é desejável ter funções claras e legíveis que realizem uma tarefa clara e simples. Evite qualquer técnica de ofuscação de código, como one-liner, ...

III.10 Arquivos

- Você não pode incluir um arquivo .c em um arquivo .c.
- Você não pode ter mais de 5 definições de função em um arquivo .c.

III.11 Makefile

Makefiles não são verificados pela `norminette` e devem ser checados durante a avaliação pelo estudante quando solicitado pelas diretrizes de avaliação. A menos que haja instruções específicas, as seguintes regras se aplicam aos Makefiles:

- As regras *$(NAME)*, *clean*, *fclean*, *re* e *all* são obrigatórias. A regra *all* deve ser a padrão e executada ao digitar apenas `make`.

- Se o makefile fizer relink quando não for necessário, o projeto será considerado não funcional.

- No caso de um projeto com múltiplos binários, além das regras acima, você deve ter uma regra para cada binário (ex: $(NAME_1), $(NAME_2), ...). A regra "all" irá compilar todos os binários, utilizando a regra de cada binário.

- No caso de um projeto que utiliza uma função de uma biblioteca não do sistema (ex.: `libft`) que existe junto ao seu código fonte, seu makefile deve compilar essa biblioteca automaticamente.

- Todos os arquivos fonte necessários para compilar seu projeto devem ser explicitamente nomeados no seu Makefile. Ex.: nada de "*.c", nada de "*.o", etc ...