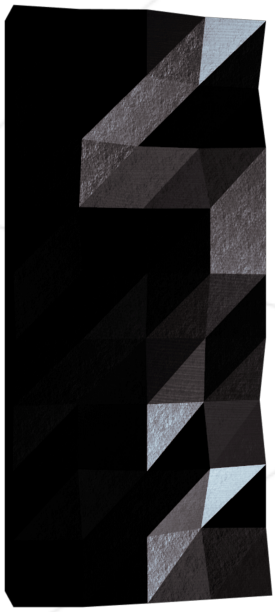

The Norm

Version 4.1

Summary: このドキュメントは、コーディングの際に従うべき一連の規則を定義した、*42* において適用されるプログラミング規格 *(Norm)* について説明したものである。 *Norm* は、 *Common Core* 内のすべての *C*言語プロジェクトには無条件に適用され、このほか *Norm* が適用される旨の記載があれば、それらのプロジェクトにも適用される。

Contents

Chapter I

前書き

`norminette` は、ソースコードが Norm を遵守しているかを判定するためのオープンソースソフトウェアな Python プログラムである。このプログラムは、Norm で規定した多くの制約について検査するが、全部ではない（例えば主観が関わるもの）。キャンパス固有に規則を改変していない限りは、レビューにおいて `norminette` が優先される。以下のページでは、`norminette` では確認できない検査項目を *(*)*印で示しており、レビュワーがコードレビュー中に発見した場合には、そのプロジェクトは不合格になり得る（ Norme フラグを使用する）。

リポジトリは `https://github.com/42School/norminette` から利用可能である。

プルリクエスト、提案、バグ報告も受け付けている。

Chapter II

目的

Norm は教育上の多くの要求を満たすために丁寧に作られている。すべての理由の中で最も重要なものを以下に示す。:

- 優先順位づけ：コーディングとは、大きくて複雑なタスクを、一連の単純な処理に分割することを意味する。これらすべての処理は次から次へと一つずつ順次実行される。ソフトウェア作りを始めたばかりの初心者は、すべの個別の処理と正確な実行順序を完全に理解しながら、単純かつ明快なアーキテクチャを必要とする。複数の処理をあたかも同時に行うような難解な構文のコードは理解しづらく、ソースコードの同じ区画に複数のタスクを割り当てるような関数はエラーの原因となりうる。
  Norm は、コード各部分に固有のタスクが明確に理解及び検証でき、実行される一連の処理に疑いの余地を残さないような、複雑でないコードを作成することを求めている。そのため、関数内の行数を最大 25 行に制限しており、`for` や `do...while`、三項演算子の使用を禁止している。

- 見た目と印象（スタイル）: 普段のピア学習やピアレビューにおいて友人や同僚とコードを共有する際、コードを解読することに時間を費すよりも、コードのロジックについて直接議論できた方が良い。
  Norm は、関数や変数の命名、インデント、括弧の規則、様々な場所におけるタブや空白、その他に関して、特定のスタイルを使用することを求めている。これにより、他の人のコードをスムーズに確認でき、理解の前段階のコードを読むことに時間を費すことなく、直接要点をつかむことができる。 Norm はまた、トレードマークとしても機能する。あなたがいずれ労働市場に出た際に、42コミュニティの一員として、他の 42 学生や 42 出身者が書いたコードを認識できるようになる。

- 長期的な視点：理解しやすいコードを書く努力をすることは、それを改修するための最善の方法である。あなたを含め誰かがバグ修正や機能追加をしなければならない際に、以前に正しい方法で物事を行っていれば、何をしているのかを理解するために貴重な時間を失うことはない。これにより、時間がかかるという理由だけでコードの保守が停止するような状況を避けることができ、市場で成功する製品を持つことにおいて、違いを生む。このことを早く学ぶほど良い。

- 参考文献：Norm に含まれる規則の一部またはすべてが恣意的だと思うかも知れないが、何をすべきか、どのようにすべきかについて私たちは実際に考え、勉強を重ねてきた。何故関数が短く単一の処理のみを行うべきなのか、何故変

数名が意味を持つべきなのか、何故各行を 80 文字以下にすべきか、何故関数が取る引数の数を抑えるべきなのか、何故コメントが有用であるべきかなどについて、Google で検索することを強く推奨する。

Chapter III

The Norm

III.1　命名規則

- 構造体 (struct) 名の先頭は `s_` とする。
- typedef 名の先頭は `t_` とする。
- 共用体 (union) 名の先頭は `u_` とする。
- 列挙型 (enum) 名の先頭は `e_` とする。
- グローバル変数名の先頭は `g_` とする。
- 変数、関数名、ユーザー定義型などの識別子は、小文字、数字、アンダースコアのみを含むものとする（スネークケース)。大文字は使用しない。
- ファイル名とディレクトリ名は、小文字、数字およびアンダースコアのみを含むものとする（スネークケース)。
- ASCII コード表に含まれない文字は、リテラル文字列を除き使用不可とする。
- *(*)* すべての識別子（関数、型、変数など）の名前は、英語として読めるものかつ明示的又は覚えやすいものとし、各単語はアンダースコアで区切るものとする。この規則は、はマクロ、ファイル名、ディレクトリにも適用される。
- `const` または `static` でマークされていないグローバル変数の使用は禁止とし、Norm エラーとみなされる（ただし、プロジェクトで明示的に許可されている場合は除く）
- ファイルはコンパイル可能である必要がある。コンパイルできないファイルは Norm の基準に不適合とみなされる。

III.2 フォーマット

- 各関数は、関数自身の波括弧記号の行を除き、最大 25 行とする。
- 各行は、コメントを含めて最大 80 文字とする。但し、水平タブ記号は1文字としてカウントされるのではなく、それが表す半角スペースの文字数分がカウントされる。
- 関数の前後は空行で区切る。関数間にコメントやプリプロセッサ命令を挿入することは許容されるが、少なくとも1つの空行をそれらの間に挿入するものとする。
- コードは、4文字分の長さの水平タブ記号（ASCIIコード表で *0x09* にマップされる）でインデントする。これは4つの半角空白記号 (*0x20* にマップされる) とは異なる。`norminette` によって検証される適切なインデントを視覚的に得るために、コードエディタが正しく設定されているか確認すること。
- 波括弧内のブロックはインデントする。波括弧はそれ単独の行とする。ただし、 struct、enum、union の宣言部分は除く。
- 空行は文字を含んではならない。即ち、空白記号やタブ記号を含んではならない。
- 行末が空白記号や水平タブ記号であってはならない。
- 空行は連続してはならない。空白記号は連続してはならない。
- 宣言は関数の冒頭に配置する。
- 変数名は、そのスコープ内ですべて同じ列にインデントする。注意：型名は、それが含まれるブロックごとに既にインデントされた状態である。
- ポインタに付随するアスタリスク記号は、その変数名に隣接させる。即ち、アスタリスク記号と変数名の間に空白記号または水平タブ記号を挿入してはならない。
- 変数宣言は、一行につき1つの変数とする。
- 宣言と初期化を同一行内で行ってはならない。ただし、グローバル変数（許可されている場合)、静的変数、定数を除く。
- 関数内の変数宣言と関数の残りの部分との間には、空行を1行挿入する。関数内に他の空行を入れてはならない。
- 一行につき1つの命令または制御構造 (`if` や `while`) のみが許可される。例えば、制御構造内での代入は禁止とし、同一行内での2つ以上の代入も禁止とし、制御構造の終端は必ず改行する。
- 命令または制御構造を、必要に応じて複数行に分割しても良い。追加される行は最初の行と比較してインデントされ、演算子の前後で改行する場合は演算子を前の行の末尾ではなく新しい行の先頭に配置する。
- 行末を除き、各カンマ記号またはセミコロン記号の後には空白記号が必要である。

- 各演算子または被演算子は、空白記号で区切られるものとする。
- 各 C キーワードの後には単一の空白記号を挿入する。ただし、型名（int、char、float等）および sizeof 演算子を除く。
- 制御構造（if や while）内は、単一行に単一の命令を含む場合を除き、波括弧で囲むものとする。

一般的な例：

```
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 関数パラメータ

- 関数が引数を取る場合、最大4つの名前付きパラメータとする。
- 関数が引数を取らない場合、プロトタイプの引数に `void` キーワードを明示的に記述する。
- 関数のプロトタイプのパラメータには名前が必要である。
- 各関数内に宣言可能な変数は、最大5つとする。
- 関数の `return` 文は、戻り値を丸括弧で囲むものとする。ただし、戻り値を返さない関数を除く。
- 関数のプロトタイプでは、その戻り値の型と関数名の間に単一の水平タブ記号を挿入する。

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

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

III.4 typedef, struct, enum, union

- 他の C キーワードと同様、struct 宣言する際は `struct` と名前の間に単一の半角空白記号を挿入する。enum および union においても同様とする。
- struct 型の変数を宣言する際は、通常の型の変数と同様にインデントする。enum および union においても同様とする。
- struct、enum、union の波括弧内では、他のブロックと同様に通常のインデントを適用する。
- 他の C キーワードと同様、`typedef` の後に単一の半角空白記号を挿入し、新しく定義された名前に通常のインデントを適用する。
- すべての構造体の名前は、そのスコープ内で同じ列にインデントする。
- .c ファイル内で、構造体を宣言してはならない。

III.5 ヘッダー - インクルードファイル

- *(*)* ヘッダーファイルで許可される要素: ヘッダーファイルのインクルード（システムまたはそれ以外）、宣言、プリプロセッサ命令、プロトタイプ、マクロ。
- インクルードはすべてファイルの先頭で行うものとする。
- ヘッダーファイルまたは .c ファイル内で、他の .c ファイルをインクルードしてはならない。
- ヘッダーファイルをインクルードする際は、必ず二重インクルードから保護する。ファイル名が `ft_foo.h` の場合、そのインクルードガード用のマクロ名は `FT_FOO_H` である。
- *(*)* 未使用のヘッダーファイルをインクルードしてはならない。
- ヘッダーファイルのインクルードは、.c ファイルと .h ファイル自体でコメントを使用して正当化することができる。

```
#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 42ヘッダー - スタイルのあるファイルの開始

- すべての .c と .h ファイルの先頭には、42ヘッダーを配置する（有用な情報を含む特別な形式の複数行コメント）。42ヘッダーは標準でクラスターの PC の様々なテキストエディタ（emacs: `C-c C-h` を入力、vim: `:Stdheader` または `F1` を入力、など）で利用可能である。
- *(*)* 42ヘッダーには、作成者（login 名と学生メール (*login@student.42tokyo.jp*)）、作成および最終更新の login 名と日時を含む情報を含めるものとする。ファイルがディスクに保存される度に、情報が自動的に更新されるべきである。

デフォルトの 42ヘッダーは、自動的にはあなたの個人情報が設定されない可能性がある。上記の規則に従うためには、設定する必要があるかも知れない。

III.7 マクロとプリプロセッサ

- *(*)* プリプロセッサ定数（または `#define`）の作成は、リテラルまたは定数値のために使用するもののみとする。
- *(*)* Norm を回避、またはコードを難読化する目的で `#define` を用いてはならない。
- *(*)* 標準ライブラリから利用可能なマクロは、それらがプロジェクトで許可されている範囲でのみ使用しても良い。
- 複数行にわたるマクロを記述してはならない。
- マクロ名はすべて大文字とする。
- `#if`、`#ifdef`、または `#ifndef` ブロック内のプリプロセッサ命令はインデントする。
- グローバルスコープの外では、プリプロセッサ命令を使用してはならない。

III.8 禁止事項！

- 下記を使用してはならない：
  - `for`
  - `do...while`
  - `switch`
  - `case`
  - `goto`
- '?' のような三項演算子
- 可変長配列 (VLA)
- 変数宣言における暗黙的な型

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

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

III.9 コメント

- コメントは、関数内に配置してはならない。コメントは、行末に、またはコメント単独の行として配置するものとする。
- *(*)* コメントは英語で記述し、有用である必要がある。
- *(*)* コメントは、キャリーオールまたは悪い関数の作成を正当化する手段にはならない。

一般的にキャリーオールまたは悪い関数は、関数名においては `f1`、`f2...` のような、変数名においては `a`、`b`、`c...` のような明示的でない名前を伴う。特異的で論理的な理由なく、`Norm` の回避のみを目的とした関数も、悪い関数とみなされる。それぞれが明確で単純なタスクを達成する、明確で読みやすい関数を持ったコードが望ましいことを心に留めて欲しい。ワンライナーのようなコード難読化テクニックは避けること。

III.10 ファイル

- .c ファイルで .c ファイルをインクルードしてはならない。
- 一つの .c ファイルに含めることができる関数定義は、最大5つとする。

III.11 Makefile

Makefile は `norminette` によって検査されず、評価ガイドラインで要求された場合、レビュー中にレビュワーが直接確認する必要がある。特に指示がない限り、下記の規則が Makefile に適用される：

- *$(NAME)*、*clean*、*fclean*、*re*、 *all* ルールを必須とする。*all* ルールをデフォルトのルールとし、単に `make` と入力したときに *all* ルールが実行される必要がある。
- 再コンパイルまたは再リンク不要にもかかわらず Makefile が再リンクする場合、そのプロジェクトは要件を満たしていないとみなされる。
- マルチバイナリプロジェクトの場合、上記に加えて各バイナリのルール（例：*$(NAME_1)*、*$(NAME_2)*、...）も作成する。 *all* ルールは、各バイナリルールを使用してすべてのバイナリをコンパイルする。
- 非システムライブラリ（例：`libft`）がソースコードと共に存在し、そのライブラリから関数を呼び出すプロジェクトの場合、Makefile はこのライブラリも含めて自動的にコンパイルする必要がある。
- プロジェクトをコンパイルするために必要なすべてのソースファイルについて、 Makefile に名前を明示的に列挙するものとする。即ち、`*.c`、 `*.o` 等で記述してはならない。