Aquí tienes un resumen punto por punto de la normativa de código proporcionada:

La Norma (The Norm) - Versión 4.1 [1111]

Esta normativa describe un estándar de programación con un conjunto de reglas para escribir código, aplicando por defecto a todos los proyectos C dentro del Common Core y a cualquier proyecto donde se especifique[2]. La "norminette" es un código Python de código abierto que verifica el cumplimiento de la Norm, aunque no todas sus restricciones (ej. las subjetivas)[3]. Las reglas no verificadas por la norminette están marcadas con

$(*)$ y pueden causar el fallo del proyecto si son descubiertas por el evaluador[4].

I. Prólogo (Foreword)
[5]

- La "norminette" es una herramienta de código abierto en Python que verifica la conformidad del código fuente con la Norma[6].
- Verifica muchas restricciones, pero no todas (ej. restricciones subjetivas)[7].
- Salvo regulaciones locales específicas, la "norminette" prevalece durante las evaluaciones[8].
- Las reglas no verificadas por la "norminette" están marcadas con $(*)$ y pueden llevar al fallo del proyecto si son descubiertas por el evaluador[9].
- El repositorio está disponible en https://github.com/42School/norminette[10].

II. ¿Por qué? cite_start [11]

- **Secuenciación:** La Norma promueve la creación de piezas de código simples, donde la tarea de cada pieza se entienda y verifique claramente, y la secuencia de instrucciones no deje dudas[12]. Por eso se limitan las funciones a un máximo de 25 líneas y se prohíben

  for, do while, o ternarios[13].
- **Apariencia (Look and Feel):** Se pide un aspecto y estilo específico para facilitar la revisión y comprensión del código entre compañeros, incluyendo reglas para nombres de funciones y variables, indentación, y uso de tabulaciones y espacios[14]. También sirve como una marca distintiva de la comunidad 42[15].
- **Visión a largo plazo:** Escribir código comprensible es la mejor manera de mantenerlo,

ahorrando tiempo al corregir errores o añadir nuevas características[16].

III. La Norma (The Norm)
[17]

III.1 Nomenclatura (Naming) [18]

- Los nombres de las estructuras deben comenzar con s_[19].
- Los nombres de los typedef deben comenzar con t_[20].
- Los nombres de las union deben comenzar con u_[21].
- Los nombres de las enum deben comenzar con e_[22].
- Los nombres de las globales deben comenzar con g_[23].
- Los identificadores (variables, funciones, tipos definidos por el usuario) solo pueden contener minúsculas, dígitos y _ (snake_case); no se permiten mayúsculas[24].
- Los nombres de archivos y directorios solo pueden contener minúsculas, dígitos y _ (snake_case)[25].
- Los caracteres que no forman parte de la tabla ASCII estándar están prohibidos, excepto dentro de cadenas literales y caracteres[26].
- $\\left(^{*}\\right)$ Todos los identificadores (funciones, tipos, variables, etc.) deben ser explícitos o mnemotécnicos, legibles en inglés, con cada palabra separada por un guion bajo[27]. Esto también aplica a macros, nombres de archivos y directorios[28].
- El uso de variables globales que no estén marcadas como const o static está prohibido y se considera un error de norma, a menos que el proyecto las permita explícitamente[29].

- El archivo debe compilar. Un archivo que no compila no se espera que pase la Norma[30].

III.2 Formato (Formatting) [31]

- Cada función debe tener como máximo 25 líneas, sin contar las llaves de la propia función[32].

- Cada línea debe tener como máximo 80 columnas de ancho, incluyendo los comentarios[33]. Una tabulación no cuenta como una sola columna, sino como el número de espacios que representa[34].

- Las funciones deben estar separadas por una línea vacía[35]. Se pueden insertar comentarios o instrucciones de preprocesador entre funciones[36]. Al menos una línea vacía debe existir[37].

- El código debe indentarse con tabulaciones de 4 caracteres de largo (ASCII char número 9)[38].

- Los bloques dentro de llaves deben estar indentados[39]. Las llaves van solas en su propia línea, excepto en la declaración de

  struct, enum, union[40].

- Una línea vacía debe estar realmente vacía: sin espacios ni tabulaciones[41].

- Una línea nunca puede terminar con espacios o tabulaciones[42].

- No se pueden tener dos líneas vacías consecutivas ni dos espacios consecutivos[43].

- Las declaraciones deben estar al principio de una función[44].

- Todos los nombres de variables deben estar indentados en la misma columna dentro de su ámbito[45].

- Los asteriscos que acompañan a los punteros deben ir pegados a los nombres de las variables[46].

- Una única declaración de variable por línea[47].

- La declaración y la inicialización no pueden estar en la misma línea, excepto para

variables globales (cuando estén permitidas), variables estáticas y constantes[48].

- En una función, debe haber una línea vacía entre las declaraciones de variables y el resto de la función[49]. No se permiten otras líneas vacías en una función[50].

- Solo se permite una instrucción o estructura de control por línea[51]. Por ejemplo, la asignación en una estructura de control está prohibida, dos o varias asignaciones en la misma línea están prohibidas, se necesita un salto de línea al final de una estructura de control[52].

- Una instrucción o estructura de control puede dividirse en varias líneas cuando sea necesario[53]. Las líneas siguientes deben estar indentadas con respecto a la primera línea, se utilizarán espacios naturales para cortar la línea, y si aplica, los operadores irán al principio de la nueva línea y no al final de la anterior[54].

- A menos que sea el final de una línea, cada coma o punto y coma debe ir seguido de un espacio[55].

- Cada operador u operando debe estar separado por uno y solo un espacio[56].

- Cada palabra clave de C debe ir seguida de un espacio, excepto las palabras clave para tipos (como
  int, char, float, etc.), así como sizeof[57].

- Las estructuras de control (
  if, while..) deben usar llaves, a menos que contengan una sola instrucción en una sola línea[58].

III.3 Funciones (Functions) [59]

- Una función puede tomar como máximo 4 parámetros con nombre[60].

- Una función que no toma argumentos debe ser prototipada explícitamente con la palabra "void" como argumento[61].

- Los parámetros en los prototipos de las funciones deben tener nombre[62].

- No se pueden declarar más de 5 variables por función[63].

- El

return de una función debe ir entre paréntesis, a menos que la función no devuelva nada[64].

- Cada función debe tener una sola tabulación entre su tipo de retorno y su nombre[65].

III.4 Typedef, struct, enum y union [66]

- Como otras palabras clave de C, añadir un espacio entre "struct" y el nombre al declarar una estructura[67]. Lo mismo aplica a

  enum y union[68].

- Al declarar una variable de tipo
  struct, aplicar la indentación habitual para el nombre de la variable[69]. Lo mismo aplica a

  enum y union[70].

- Dentro de las llaves de
  struct, enum, union, se aplican las reglas de indentación regulares, como en cualquier otro bloque[71].

- Como otras palabras clave de C, añadir un espacio después de "typedef", y aplicar la indentación regular para el nuevo nombre definido[72].

- Todas los nombres de estructuras deben indentarse en la misma columna para su ámbito[73].

- No se puede declarar una estructura en un archivo
  .c[74].

III.5 Cabeceras (Headers) - también conocidos como archivos include [75]

- $\\left(^{*}\\right)$ Los elementos permitidos en un archivo de cabecera son: inclusiones de cabeceras (del sistema o no), declaraciones, defines, prototipos y macros[76].

- Todos los
  includes deben estar al principio del archivo[77].

- No se puede incluir un archivo
  .c en un archivo de cabecera o en otro archivo .c[78].

- Los archivos de cabecera deben estar protegidos contra dobles inclusiones[79]. Si el archivo es

  ft_foo.h, su macro de guarda es FT_FOO_H[80].

- $\\left(^{*}\\right)$ La inclusión de cabeceras no utilizadas está prohibida[81].

- La inclusión de cabeceras puede justificarse en el archivo
  .c y en el propio archivo .h utilizando comentarios[82].

III.6 La cabecera 42 (The 42 header) - también conocido como empezar un archivo con estilo [83]

- Todo archivo
  .c y .h debe comenzar inmediatamente con la cabecera estándar 42: un comentario de varias líneas con un formato especial que incluye información útil[84].

- $\\left(^{*}\\right)$ La cabecera 42 debe contener información actualizada, incluyendo el creador con el login y el correo electrónico del estudiante (@student.campus), la fecha de creación, el login y la fecha de la última actualización[85]. Cada vez que el archivo se guarda en disco, la información debe actualizarse automáticamente[86].

III.7 Macros y Preprocesadores (Macros and Pre-processors) [87]

- $\\left(^{*}\\right)$ Las constantes del preprocesador (o #define) que se creen deben usarse solo para valores literales y constantes[88].

- $\\left(^{*}\\right)$ Todas las #define creadas para eludir la norma y/o ofuscar código están prohibidas[89].

- $\\left(^{*}\\right)$ Se pueden usar macros disponibles en bibliotecas estándar, solo si estas están permitidas en el ámbito del proyecto dado[90].

- Las macros multilinea están prohibidas[91].

- Los nombres de las macros deben estar en mayúsculas[92].

- Las directivas del preprocesador deben indentarse dentro de bloques #if, #ifdef o #ifndef[93].

- Las instrucciones del preprocesador están prohibidas fuera del ámbito global[94].

III.8 ¡Cosas prohibidas! (Forbidden stuff!)
[95]

- No se permite usar:
  - for
  - do...while
  - switch
  - case
  - goto
- Operadores ternarios como ?[96].

- VLAs - Arrays de Longitud Variable (Variable Length Arrays)[97].

- Tipo implícito en las declaraciones de variables[98].

III.9 Comentarios (Comments) [99]

- Los comentarios no pueden estar dentro de los cuerpos de las funciones[100]. Los comentarios deben estar al final de una línea o en su propia línea[101].

- $\\left(^{*}\\right)$ Los comentarios deben estar en inglés y ser útiles[102].

- $\\left(^{*}\\right)$ Un comentario no puede justificar la creación de una función "cajón de sastre" o una mala función[103]. Una función cuyo único objetivo es evitar la norma, sin un propósito lógico único, también se considera una mala función[104]. Se desea tener funciones claras y legibles que realicen una tarea clara y simple cada una[105]. Se deben evitar técnicas de ofuscación de código, como las de una sola línea[106].

III.10 Archivos (Files) [107]

- No se puede incluir un archivo .c en otro archivo .c[108].

- No se pueden tener más de 5 definiciones de función en un archivo .c[109].

III.11 Makefile [110]

- Los Makefile no son verificados por la "norminette", y deben ser verificados durante la evaluación por el estudiante cuando lo soliciten las directrices de evaluación[111].

- Salvo instrucciones específicas, se aplican las siguientes reglas a los Makefile:
    - Las reglas $(NAME), clean, fclean, re y all son obligatorias[112]. La regla all debe ser la predeterminada y ejecutarse al escribir solo make[113].
    - Si el makefile vuelve a enlazar cuando no es necesario, el proyecto se considerará no funcional[114].
    - En el caso de un proyecto multi-binario, además de las reglas anteriores, debe haber una regla para cada binario (ej: $(NAME_1), $(NAME_2), ...)[115]. La regla "all" compilará todos los binarios, utilizando cada regla de binario[116].
    - En el caso de un proyecto que llama a una función de una biblioteca no del sistema (ej: libft) que existe junto con el código fuente, el makefile debe compilar esta biblioteca automáticamente[117].
    - Todos los archivos fuente necesarios para compilar el proyecto deben estar explícitamente nombrados en el Makefile[118]. Ej: no *.c, no *.o, etc[119].