UNIVERSIDAD TECNOLÓGICA DE QUERÉTARO

Diplomado en Software Embebido

Módulo 3.4: Estándares de programación
Tema 3.4.1 Introducción a la calidad
Tema 3.4.2 Normas ISO/IEC.
Tema 3.4.3 Misra.
Tema 3.4.4 Herramienta para revisión de estándares de código.
Tema 3.4.5 Generación de documentación de código

Adbeel Alejandro Pérez Martínez

Octubre 2018

INDICE DE TEMAS

ÍNDICE DE FIGURAS

I.

II. Desarrollo del tema.

3.4. Estándares de programación

3.4.1. Introducción a la calidad

"La calidad nunca es un accidente; siempre es el resultado de un esfuerzo de la inteligencia."
John Ruskin (1819-1900)

3.4.1.1. Definición de calidad

El término "calidad" ha evolucionado a lo largo del tiempo. Algunas de las definiciones recopiladas más representativas son las siguientes:

> *"Constitución, con la cual la mercadería satisface el empleo previsto"* [Asociación Alemana para la Calidad, DGQ, 1972];
>
> *"Conjunto de propiedades y características de un producto o servicio, que confiere su aptitud para satisfacer las necesidades dadas"* [Instituto Alemán para la Normalización, DIN 55 350-11, 1979];
>
> *"La totalidad de las características de una entidad que le confieren la aptitud para satisfacer las necesidades establecidas y las implícitas"* [Instituto Centroamericano de Tecnología Industrial – Comisión Panamericana de Normas Técnicas – Organización Internacional de Normalización ICAITI-COPANT-ISO 8402, 1995]

Tomando como base las definiciones anteriores, la calidad se relaciona más bien con las exigencias de los consumidores con respecto a la satisfacción de sus necesidades.

Las necesidades son el conjunto de todas las características de un producto o servicio que tengan importancia para el cliente, algunas de ellas pueden ser implícitas sin que el cliente las exija de manera explícita, pero de todas formas son vitales.

3.4.1.2. Definición de estándar

El término estándar tiene su origen etimológico en el vocablo inglés standard. El concepto se utiliza para nombrar a aquello que puede tomarse como referencia, patrón o modelo.

Se conoce como estandarización o normalización al proceso que apunta a la creación y la aplicación de normas que son utilizadas a nivel general en un determinado ámbito. La *Internacional Organization for Standarization* (ISO) es la entidad mundial que trabaja para el establecimiento de disposiciones diseñadas

para un uso común y repetido, lo cual permite alcanzar un determinado ordenamiento que ayuda a resolver un problema potencial o real.

Las normas de calidad ISO, por citar un caso, constituyen un estándar que certifica las cualidades de un producto. Cuando algo cuenta con una norma de calidad ISO, los compradores ya tienen una garantía respecto a un cierto estándar alcanzado por el elemento en cuestión.

3.4.1.3. Estándares ISO/IEC en software

¿Qué es ISO?

ISO (International Standard Organization) u Organización Internacional de Normalización, es un organismo que se dedica a publicar normas a escala internacional y que en este caso (el campo de la gestión de la calidad).

En esta organización internacional (ISO), se encuentran representados hoy en día alrededor de noventa países de todos los continentes a través de organismos creados con este mismo objetivo. Sus normas son el resultado de acuerdos logrados por todos los representantes, quienes defienden los intereses de los sectores industriales de cada uno de sus países al crear o modificar las normas y políticas de ISO.

¿Qué es la IEC?

La Comisión Electrotécnica Internacional (IEC) es la principal organización del mundo que prepara y publica estándares internacionales para todas las tecnologías eléctricas, electrónicas y relacionadas.

Fundada en 1906, la IEC (Comisión Electrotécnica Internacional) es la organización líder en el mundo para la elaboración y publicación de las Normas internacionales para todas las tecnologías eléctricas, electrónicas y relacionadas. Éstos se conocen colectivamente como "electrotécnica".

Más de 10 000 expertos de la industria, el comercio, el gobierno, la prueba y los laboratorios de investigación, las universidades y los grupos de consumidores participen en el trabajo de normalización IEC.

IEC proporciona una plataforma para las empresas, las industrias y los gobiernos para hacer frente, la discusión y el desarrollo de las normas internacionales que requieren.

Millones de dispositivos que contienen la electrónica, y usar o producir electricidad, se basan en normas internacionales de la IEC y los sistemas de evaluación de la conformidad para llevar a cabo, en forma y trabajar de manera segura.

Todas las normas internacionales de la IEC son totalmente basadas en el consenso y representan las necesidades de las principales partes interesadas de todas las naciones que participan en el trabajo de IEC. Cada país miembro, no importa cuán

grande o pequeña, tiene un voto y voz en lo que sucede en una norma internacional IEC.

Los estándares ISO/IEC para software
Los estándares de calidad de software hacen parte de la ingeniería de software, utilización de estándares y metodologías para el diseño, programación, prueba y análisis del software desarrollado, con el objetivo de ofrecer una mayor confiabilidad, mantenibilidad en concordancia con los requisitos exigidos, con esto se eleva la productividad y el control en la calidad de software, parte de la gestión de la calidad se establecen a mejorar su eficacia y eficiencia.

En un escenario en el que los sistemas de software se desarrollan y construyen por terceros proveedores, el contratante del servicio, como primer receptor del mismo, en muchos casos debe confiar en el buen hacer del proveedor seleccionado, especialmente si nos dispone de los medios apropiados para auditar la entrega y en su caso argumentar defectos en el proceso de desarrollo.

En general, una vez validado que el sistema responde a los principales requisitos funcionales especificados, el usuario realizará las pruebas de aceptación, corrigiendo los errores encontrados y pasándose al fin del entorno de producción. Sin embargo, en muy pocas ocasiones se validan de manera rigurosa los requisitos funcionales y los no funcionales, o se ejecutan validaciones que aseguren que el sistema es lo suficientemente robusto y estable como para pasar a un entorno productivo con las garantías adecuadas.

3.4.1.4. Dinámica grupal.

1. Conteste las siguientes preguntas.
    a. ¿Compila el programa?
    b. ¿Durante la compilación se generarán warnings?
    c. Encuentre los errores que mejoran la calidad del SW y justifique su respuesta.
2. Escriba las soluciones que corrijan el código.

```
#include <stdio.h>


char var[30] = "hola que tal "
                  "que dices "
                  "que que";
char inc(char var1);

int main()
{
      int value;
      char *ptr;
      ptr=var;
```

```
	printf(var);
	printf("\n");
	printf("length %i\n",sizeof(var));
	printf("strange value %i \n",inc(value));
	for (unsigned int i=0;i<sizeof(var);i++)
	{
		char tmp=inc(*(ptr+i));
		*(ptr+i)=tmp;
		printf("%c\n",*(ptr+i));
	}

	return 0;
}

char inc(char var1)
{
	return var1+1;
}
```

3.4.2. Normas ISO/IEC

"La salud de una sociedad democrática puede medirse por la calidad de las funciones desempeñadas por los particulares."
Alexis de Tocqueville

3.4.2.1. Norma ISO/IEC 9126 – Evalúa los Productos de Software

La norma ISO/IEC 9126 de 1991, es la norma para evaluar los productos de software, esta norma nos indica las características de la calidad y los lineamientos para su uso, las características de calidad y sus métricas asociadas, pueden ser útiles tanto como para evaluar el producto como para definir los requerimientos de la calidad y otros usos. Esta norma definida por un marco conceptual basado en los factores tales como Calidad del Proceso, Calidad del Producto del Software y Calidad en Uso; según el marco conceptual, la calidad del producto, a su vez, contribuye a mejorar la calidad en uso.

Parte beneficiaria / Características	Usuario Final	Organización	Soporte técnico
Efectividad	Efectividad del usuario	Efectividad de las tareas	Efectividad del mantenimiento
Recursos	Productividad del usuario (tiempo)	Coste-Eficiencia (dinero)	Coste del mantenimiento
Consecuencias adversas	Riesgos para el usuario (salud y seguridad)	Riesgo comercial	Corrupción o fallos del software
Satisfacción	Satisfacción del usuario	Satisfacción en la gestión	Satisfacción del mantenimiento

Figura 1. ISO/IEC 9126, Tabla de calidad desde el punto de vista usuario

La norma ISO/IEC 9126 define la calidad en uso como la perspectiva del usuario de la calidad del producto software cuando éste es usado en un ambiente específico y un contexto de uso específico. Éste mide la extensión para la cual los usuarios pueden conseguir sus metas en un ambiente particular, en vez de medir las propiedades del software en sí mismo.

El modelo de la calidad en uso muestra un conjunto de 4 características: efectividad, productividad, integridad, y satisfacción.

3.4.2.2. Norma ISO / IEC 9899 – Estándares de codificación

Actualmente hay dos versiones principales de la Norma: ISO / IEC 9899: 1990 (comúnmente denominada C89 o C90) e ISO / IEC 9899: 1999 (comúnmente denominada C99). Como lo indica la denominación, C99 se ratificó en 1999 y reemplaza a C89 / C90, que se ratificó en 1990.

C99 no se ha implementado ampliamente por completo, por lo que no es prudente confiar en sus características para la portabilidad, aunque la mayoría de los compiladores de C implementan C89 por completo.

Muchos compiladores implementan un subconjunto de características de C99, a veces como extensiones.

C89 / C90 y C99 son estándares ISO, sin embargo, C89 / C90 fue ratificado originalmente por ANSI como X3.159-1989. Los estándares ANSI e ISO C89 / C90 son técnicamente equivalentes, pero la numeración de la sección del estándar difiere en sus formularios ANSI e ISO.

Antes de la estandarización de ANSI e ISO, C fue estandarizado de manera no oficial alrededor de "El lenguaje de programación C", por *Brian Kernighan* y *Dennis Ritchie* (*Prentice Hall*, 1978).

Este estándar no oficial de C se conoce comúnmente como K&R, aunque no está claro si se puede considerar que se basa exclusivamente en la especificación de este libro (Apéndice A de la Edición 1), las implementaciones reales no siguieron exactamente las especificaciones del Apéndice A hubo varios cambios generalmente aceptados y algunos consideraron que el manual de referencia de AT&T era más autoritario; de ahí la necesidad de un estándar oficial (lo anterior se basa en una publicación *comp.std.c*, 2 de noviembre de 2005, de *Douglas Gwyn*).

3.4.2.2.1. Obtención de la Norma

El Estándar se puede comprar en papel o en formato digital descargable de las filiales nacionales, como se describe en el sitio web de WG14 (http://www.open-std.org/jtc1/sc22/wg14/). Dos de estos organismos nacionales son ANSI, a través de su tienda *eStandards*, y SAI Global (originalmente *Standards Australia*), que vende todas las publicaciones de la Norma C ISO. Una fuente internacional es *Techstreet*.

3.4.2.2.2. Norma C89

En 1983, el *American National Standards Institute* formó un comité, X3J11, para establecer una especificación estándar de C. El estándar se completó en 1989 y se ratificó como ANSI X3.159-1989 "Lenguaje de programación C". Esta versión del

lenguaje a menudo se conoce como "ANSI C". Más tarde, a veces, la etiqueta "C89" se usa para distinguirla de C99 pero utilizando el mismo método de etiquetado.

3.4.2.2.3. Norma C90

La misma norma que C89 fue ratificada por la Organización Internacional de Normalización como ISO / IEC 9899: 1990, con solo cambios de formato, [2] que a veces se denomina C90. Por lo tanto, los términos "C89" y "C90" se refieren esencialmente al mismo lenguaje.

Esta norma ha sido retirada por ANSI / INCITS [3] e ISO / IEC. [4]

3.4.2.2.4. Norma C95

En 1995, la ISO publicó una extensión, llamada Enmienda 1, para el estándar ANSI-C. Su nombre completo finalmente fue ISO / IEC 9899 / AMD1: 1995 o apodado C95.
Aparte de la corrección de errores, hubo más cambios en las capacidades del lenguaje, tales como:

- Compatibilidad con caracteres múltiples y de múltiples bytes mejorada en la biblioteca estándar, introduciendo <wchar.h> y <wctype.h> así como también E / S de múltiples bytes
- Adición de dígrafos al lenguaje.
- Especificación de macros estándar para la especificación alternativa de operadores, p. Ej. y para &&
- Especificación de la macro estándar __STDC_VERSION__

Además de la enmienda, la ISO para C90 publicó dos correcciones técnicas:
ISO / IEC 9899 TCOR1 en 1995
ISO / IEC 9899 TCOR2 en 1996

3.4.2.2.5. Norma C99

C99 (anteriormente conocido como C9X) es un nombre informal para ISO / IEC 9899: 1999, una versión anterior del estándar de lenguaje de programación C. Amplía la versión anterior (C90) con nuevas características para el lenguaje y la biblioteca estándar, y ayuda a las implementaciones a hacer un mejor uso del hardware informático disponible, como la aritmética IEEE 754-1985 y la tecnología de compilación. La versión C11 del estándar de lenguaje de programación C, publicada en 2011, reemplaza a C99.

3.4.2.2.6. Norma C03 C++

Este es el Lenguaje C++ definido en la norma “ISO/IEC 14882:2003 Programming language C++”.

3.4.2.2.7. Norma C11

C11 (anteriormente C1X) es un nombre informal para ISO / IEC 9899: 2011, un estándar anterior para el lenguaje de programación C.

Reemplazó a C99 (norma ISO / IEC 9899: 1999) y fue reemplazado por C18 (norma ISO / IEC 9899: 2018).

C11 estandariza principalmente las funciones que ya son compatibles con los compiladores contemporáneos comunes, e incluye un modelo de memoria detallado para admitir mejor múltiples subprocesos de ejecución. Debido a la disponibilidad diferida de las implementaciones C99 conformes, C11 hace que algunas características sean opcionales, para que sea más fácil cumplir con el estándar de lenguaje central.

El borrador final, N1570, [4] se publicó en abril de 2011. La nueva norma aprobó su borrador final de revisión el 10 de octubre de 2011 y fue oficialmente ratificada por ISO y publicada como ISO / IEC 9899: 2011 el 8 de diciembre de 2011, con No hay comentarios que requieran la resolución de los organismos nacionales participantes.

Una macro estándar __STDC_VERSION__ se define con el valor 201112L para indicar que el soporte de C11 está disponible. GCC admite algunas características de C11 a partir de la versión 4.6, Clang a partir de la versión 3.1, e IBM XL C a partir de la versión 12.1.

3.4.2.2.8. Norma C18

C18 y C17 son nombres informales para ISO / IEC 9899: 2018, la norma más reciente para el lenguaje de programación C, publicada en junio de 2018. Reemplazó C11 (norma ISO / IEC 9899: 2011). El soporte fue programado para GCC 8 y LLVM Clang 6.0.

Historia de la C18

Después de que ANSI produjo el estándar oficial para el lenguaje de programación C en 1989, que se convirtió en un estándar internacional en 1990, la especificación del lenguaje C se mantuvo relativamente estática durante algún tiempo, mientras que C++ siguió evolucionando, en gran parte durante su propio esfuerzo de estandarización. La Enmienda Normativa 1 creó un nuevo estándar para C en 1995, pero solo para corregir algunos detalles del estándar de 1989 y para agregar un soporte más amplio para los conjuntos de caracteres internacionales. La norma se sometió a una revisión adicional a fines de la década de 1990, lo que llevó a la publicación de ISO / IEC 9899: 1999 en 1999, que se adoptó como norma ANSI en mayo de 2000. El lenguaje definido por esa versión de la norma se conoce comúnmente como " C99 ". El grupo de trabajo ISO / IEC JTC1 / SC22 / WG14 mantiene el estándar C internacional.

Diseño de la C18
C99 es, en su mayor parte, compatible con versiones anteriores de C89, pero es más estricto en algunos aspectos.

En particular, una declaración que carece de un especificador de tipo ya no se ha asumido implícitamente. El comité de estándares de C decidió que era más valioso para los compiladores diagnosticar una omisión inadvertida del especificador de tipo que procesar de forma silenciosa el código heredado que se basaba en int implícito. En la práctica, es probable que los compiladores muestren una advertencia, luego asuman int y continúen traduciendo el programa.

C99 introdujo varias características nuevas, muchas de las cuales ya se habían implementado como extensiones en varios compiladores:

- funciones en línea (*inline*)
- Declaraciones y código entremezclados: la declaración de variables ya no está restringida al alcance del archivo o al inicio de una declaración compuesta (bloque), lo que facilita el formulario de asignación única estática
- varios tipos de datos nuevos, incluidos *long long int*, tipos de enteros extendidos opcionales, un tipo de datos booleano explícito y un tipo complejo para representar números complejos
- *arrays* de longitud variable (aunque posteriormente relegados en C11 a una característica condicional que las implementaciones no tienen que admitir)
- miembros de la matriz flexible
- soporte para comentarios de una línea que comienzan con //, como en BCPL, C ++ y Java
- Nuevas funciones de biblioteca, como snprintf
- nuevos encabezados, como <stdbool.h>, <complex.h>, <tgmath.h>, y <inttypes.h>
- Funciones de tipo genérico de matemáticas (macro), en <tgmath.h>, que seleccionan una función de biblioteca matemática basada en argumentos flotantes, dobles o largos, etc.
- soporte mejorado para el punto flotante IEEE
- inicializadores designados
- literales compuestos (por ejemplo, es posible construir estructuras en llamadas a funciones: función ((estructura x) {1, 2})) [5]
- soporte para macros variadic (macros con un número variable de argumentos)
- La calificación restringida permite una optimización de código más agresiva, eliminando las ventajas de acceso a la matriz en tiempo de compilación que anteriormente tenía FORTRAN sobre ANSI C
- nombres de caracteres universales, que permiten a las variables de usuario contener otros caracteres que no sean el conjunto de caracteres estándar
- palabra clave estática en índices de matriz en declaraciones de parámetros.

Las partes del estándar C99 se incluyen en la versión actual del estándar C++, incluidos los tipos de enteros, encabezados y funciones de biblioteca. Las matrices de longitud variable no están entre estas partes incluidas porque la biblioteca de plantillas estándar de C++ ya incluye una funcionalidad similar.

3.4.2.3. Norma ISO/IEC 12207 – Modelos de Ciclos de Vida del Software.

Estándar para los procesos de ciclo de vida del software de la organización. Este estándar se concibió para aquellos interesados en adquisición de software, así como desarrolladores y proveedores. El estándar indica una serie de procesos desde la recopilación de requisitos hasta la culminación del software.

El estándar comprende 17 procesos lo cuales son agrupados en tres categorías:

- Principales
- De apoyo
- De organización

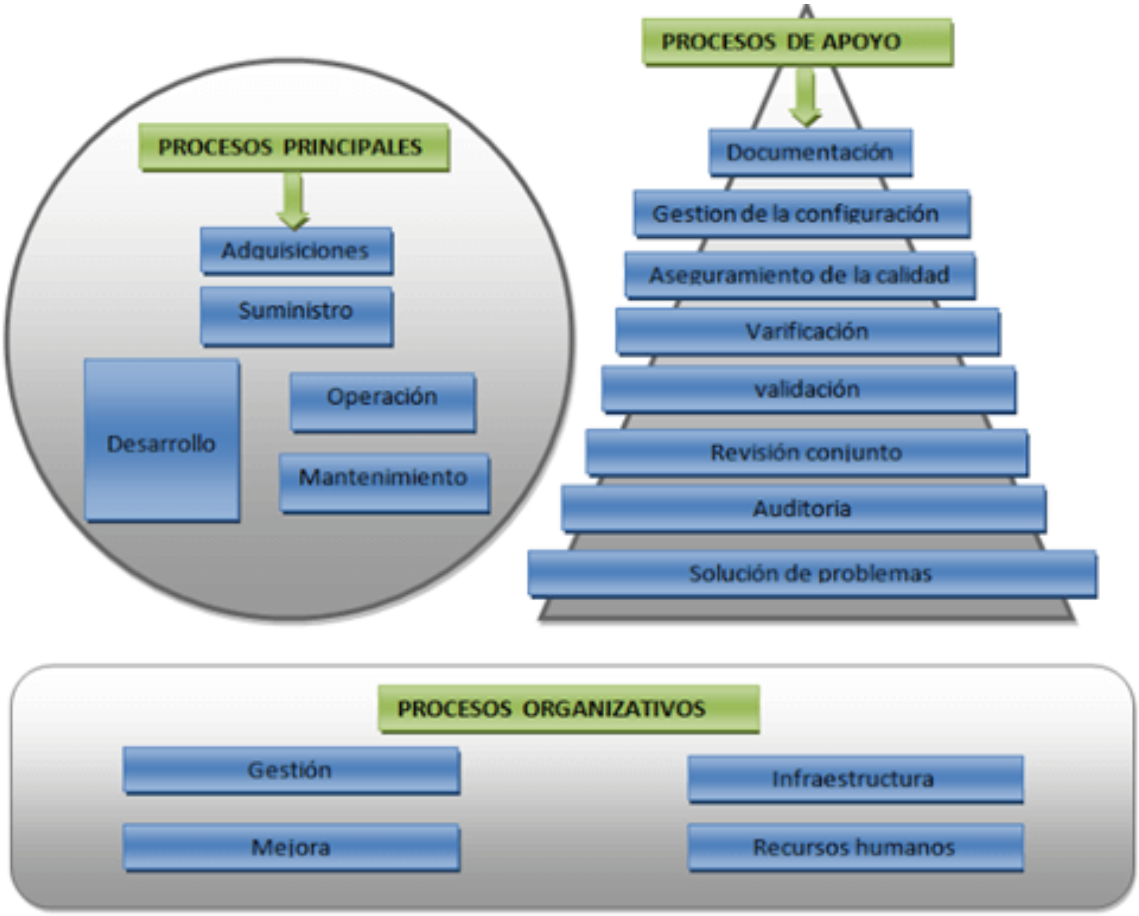


Figura 2 ISO/IEC 12207 Diagrama de procesos de software

Este estándar agrupa las actividades que se pueden llevar a cabo durante el ciclo de vida del software en cinco procesos principales, ocho procesos de apoyo y cuatro procesos organizativos.

3.4.2.4. Norma ISO/IEC 14598 – Base metodológica para evaluar productos de SW.

El estándar ISO/IEC 14598 es actualmente usado como base metodológica para la evaluación del producto software. En sus diferentes etapas, establece un marco de trabajo para evaluar la calidad de los productos de software proporcionando, además, métricas y requisitos para los procesos de evaluación de los mismos.

La norma define las principales características del proceso de evaluación

- Repetitividad.
- Reproducibilidad.
- Imparcialidad.
- Objetividad.

Para estas características se describen las medidas concretas que participan:

- Análisis de los requisitos de evaluación.
- Evaluación de las especificaciones.
- Evaluación del diseño y definición del plan de evaluación.
- Ejecución del plan de evaluación.
- Evaluación de la conclusión.

El estándar ISO/IEC 14598 define el proceso para evaluar un producto de software, el mismo consta de seis partes:

- **ISO/IEC 14598-1 Visión General**: provee una visión general de las otras cinco partes y explica la relación entre la evaluación del producto software y el modelo de calidad definido en la ISO/IEC 9126.
- **ISO/IEC 14598-2 Planeamiento y Gestión:** contiene requisitos y guías para las funciones de soporte tales como la planificación y gestión de la evaluación del producto del software.
- **ISO/IEC 14598-3 Proceso para desarrolladores:** provee los requisitos y guías para la evaluación del producto software cuando la evaluación es llevada a cabo en paralelo al desarrollo por parte del desarrollador.
- **ISO/IEC 14598-4 Proceso para compradores:** provee los requisitos y guías para que la evaluación del producto software sea llevada a cabo en función a los compradores que planean adquirir o reutilizar un producto de software existente o pre-desarrollado.
- **ISO/IEC 14598-5 Proceso para avaladores:** provee los requisitos y guías para la evaluación del producto software cuando la evaluación es llevada a cabo por evaluadores independientes.
- **ISO/IEC 14598-6 Documentación de Módulos:** provee las guías para la documentación del módulo de evaluación.

3.4.2.5. Norma ISO/IEC 15939 – Evaluación de las mediciones de calidad

ISO 15939 tiene un modelo de información que ayuda a determinar que se debe especificar durante la planificación, desempeño y evaluación de la medición. Para su aplicación, cuenta con los siguientes pasos: Recopilar los datos, Preparación de los datos y Análisis de los datos.

SQuaRE está formada por las divisiones siguientes:

- ISO/IEC 2500n. División de gestión de calidad. Los estándares que forman esta división definen todos los modelos comunes, términos y referencias a los que se alude en las demás divisiones de SQuaRE.
- ISO/IEC 2501n. División del modelo de calidad. El estándar que conforma esta división presenta un modelo de calidad detallado, incluyendo características para la calidad interna, externa y en uso.

- ISO/IEC 2502n. División de mediciones de calidad. Los estándares pertenecientes a esta división incluyen un modelo de referencia de calidad del producto software, definiciones matemáticas de las métricas de calidad y una guía práctica para su aplicación.

- ISO/IEC 2503n. División de requisitos de calidad. Los estándares que forman parte de esta división ayudan a especificar los requisitos de calidad. Estos requisitos pueden ser usados en el proceso de especificación de requisitos de calidad para un producto software que va a ser desarrollado ó como entrada para un proceso de evaluación. El proceso de definición de requisitos se guía por el establecido en la norma ISO/IEC 15288 (ISO, 2003).

ISO/IEC 2504n. División de evaluación de la calidad. Estos estándares proporcionan requisitos, recomendaciones y guías para la evaluación de un producto software, tanto si la llevan a cabo evaluadores, como clientes o desarrolladores.

ISO/IEC 25050–25099. Estándares de extensión SQuaRE. Incluyen requisitos para la calidad de productos de software “Off-The-Self” y para el formato común de la industria (CIF) para informes de usabilidad.

Práctica

Una "práctica" es una costumbre o un conjunto de ideas sobre implementación para mantener la calidad del código fuente. Cada práctica refleja el concepto básico de la regla de codificación individual. Estas prácticas se desglosan en esquemas y detalles.

Regla

Una "regla" es un acuerdo específico que debe seguirse y constituye una parte de la convención de codificación.

Esta guía presenta estas reglas como información de referencia. En esta guía, una "regla" también se usa a veces como un término colectivo que representa un grupo de reglas relevantes.

3.4.2.6. Norma ISO/IEC TR 18037:2008 – Especifica extensiones del lenguaje de programación C para procesadores integrados.

C formalmente conocido como lenguaje de programación C contiene extensiones que soportar procesadores integrados, en un informe técnico aprobado por ISO para su publicación, especifica una serie de extensiones del lenguaje de programación C. El informe técnico fue publicado como ISO / IEC TR 18037.

El informe técnico cubre tres cláusulas separadas:

- Aritmética de punto fijo,
- Espacios de direcciones con nombre y clases de almacenamiento de registros con nombre,
- Direccionamiento básico de hardware de E / S.

Debido a que este es un informe técnico, las implementaciones son libres de elegir y es posible elegir las funciones que se desean admitir. El informe técnico fomenta implementaciones que seleccionen una de las cláusulas para respaldar completamente la cláusula completa.

3.4.2.6.1. Historia

La primera edición del informe técnico se publicó en 2004 como ISO / IEC TR 18037: 2004. La segunda edición del informe técnico se publicó en 2008 como ISO / IEC TR 18037: 2008.

3.4.2.6.2. Encabezados

El reporte técnico agregó 2 nuevos encabezados: <stdfix.h> y <iohw.h>.

3.4.2.6.3. Aritmética de punto fijo

El informe técnico introdujo valores de datos de punto fijo, ya sea como valores de datos fraccionarios o como parte integral y partes fraccionarias. Los datos de punto fijo se designan con las nuevas palabras clave _Fract y _Accum. El especificador _Sat se puede usar para especificar un tipo de punto fijo de saturación. Una versión conveniente de estas palabras clave se define en <stdfix.h>. El encabezado también incluye un conjunto de nuevas funciones de soporte de operaciones aritméticas de punto fijo.

3.4.2.6.4. Clases de almacenamiento con nombre y registro

El informe técnico agregó la capacidad de acceder a los registros del procesador que no son direccionables en ninguno de los espacios de direcciones de la máquina. Dicha capacidad a menudo se requiere en procesadores integrados que realizan operadores especiales. El informe técnico agregó la capacidad de asignar un nombre al registro después de la palabra clave de registro.

3.4.2.6.5. Direccionamiento básico de hardware de E / S

El informe técnico agregó soporte para acceder a la funcionalidad de I/O de hardware del dispositivo integrado a través del nuevo encabezado <iohw.h>. El nombre de los designadores de registro de I/O debe ser proporcionado por una implementación en un encabezado de su elección. Además, la palabra clave _Access se agregó para asociar la dirección, el espacio de memoria y los límites de acceso a una declaración en C.

3.4.2.7. Norma ISO/IEC TR 19769:2004 – Especifica Extensiones para admitir nuevos tipos de datos de caracteres

El lenguaje C ha evolucionado a lo largo de las últimas décadas, se han introducido varias páginas de códigos y bibliotecas multibyte, y se ha introducido el soporte extendido de conjuntos de caracteres; sin embargo, el soporte para tipos de datos de caracteres extendidos en el lenguaje C todavía es limitado.

Hoy en día, la introducción y el éxito del estándar Unicode / ISO10646 y de su implementación en lenguajes informáticos modernos crean demandas cada vez mayores en el lenguaje C para dar un mejor soporte a Unicode / ISO10646. Este documento aborda la introducción de nuevos tipos de datos de caracteres extendidos en el lenguaje C para admitir futuros formularios de codificación de caracteres, incluido Unicode / ISO10646.

El estándar Unicode soporta 3 formas de codificación:

- UTF-8
- UTF-16
- UTF-32

Cada forma de codificación tiene ventajas y desventajas, por lo que la elección de la forma de codificación debe dejarse a la aplicación. Actualmente, algunas aplicaciones de C implementan UTF-8 usando el tipo char, UTF-16 usando short sin signo o wchar_t, y UTF-32 usando long sin signo o wchar_t. La situación actual, sin embargo, enfrenta los siguientes problemas principales:

- El tamaño de wchar_t es la implementación definida. Mientras que wchar_t ofrece una forma de portabilidad de plataforma para aplicaciones C, Unicode ofrece la posibilidad de escribir aplicaciones independientes de la plataforma utilizando un formato de datos independiente de la plataforma.

- No hay un literal de cadena para tipos de enteros basados en 16 o 32 bits, pero las formas de codificación Unicode requieren literales de cadena.

Es sensato dar a todos los formularios de codificación Unicode compatibilidad con el tipo de datos adecuado. UTF-8 se considera normalmente como la codificación multibyte preferida, para secuencias de uno o más elementos de tipo char. Este documento sugiere la implementación de tipos de datos de caracteres de 16 y 32 bits: char16_t y char32_t. Los nuevos tipos de datos garantizan la portabilidad del programa a través de anchos de caracteres claramente definidos. La codificación de los nuevos tipos de datos debe ser lo más genérica posible para admitir no solo Unicode sino también otras codificaciones de caracteres.

En general, es deseable que las aplicaciones de C procesen cadenas completas a la vez en lugar de procesar caracteres individuales de forma aislada. Este documento no especifica el detalle de las funciones de la biblioteca para los nuevos tipos de datos, excepto un conjunto de funciones de conversión de caracteres.

3.4.2.7.1. UTF8

UTF-8 es una codificación de caracteres de ancho variable capaz de codificar todos los puntos de código válidos de 1,112,064 en Unicode usando de uno a cuatro bytes de 8 bits.

La codificación está definida por el estándar Unicode y fue diseñada originalmente por Ken *Thompson* y *Rob Pike*. El nombre se deriva del formato de transformación Unicode (o juego de caracteres codificado universal) de 8 bits.

Fue diseñado para la compatibilidad con versiones anteriores con ASCII. Los puntos de código con valores numéricos más bajos, que tienden a ocurrir con más frecuencia, se codifican utilizando menos bytes. Los primeros 128 caracteres de Unicode, que se corresponden uno a uno con ASCII, se codifican utilizando un solo octeto con el mismo valor binario que ASCII, de modo que el texto ASCII válido también es Unicode codificado en UTF-8 válido. Dado que los bytes ASCII no se producen cuando se codifican puntos de código no ASCII en UTF-8, es seguro usar UTF-8 dentro de la mayoría de los lenguajes de programación y documentos que interpretan ciertos caracteres ASCII de una manera especial, como "/" en los nombres de archivo, " \ "en secuencias de escape, y"% "en printf.

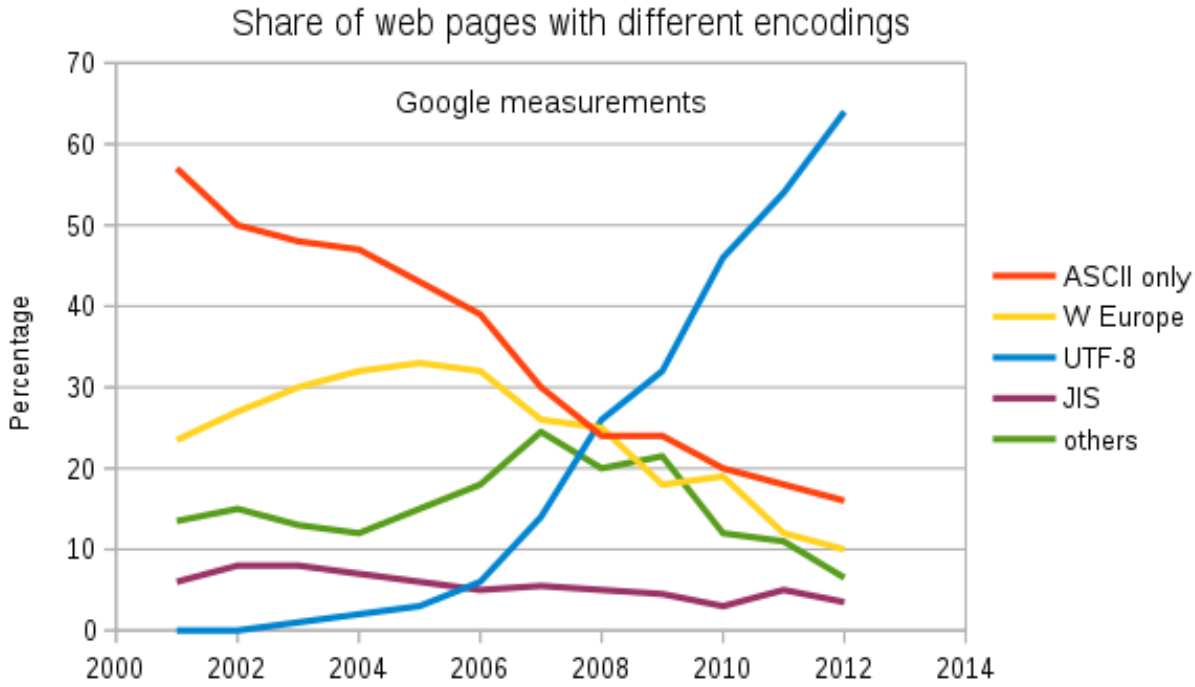


Figura 3 ISO/IEC TR 19769:2004 para uso de UTF

UTF-8 ha sido la codificación de caracteres dominante para la *World Wide Web* desde 2009, y en octubre de 2018 representa el 92.3% de todas las páginas web (algunas de las cuales son simplemente ASCII, ya que es un subconjunto de UTF-8) y el 95.4% de las 1.000 páginas web mejor calificadas. Las siguientes codificaciones multibyte más populares, Shift JIS y GB 2312, tienen un 0,4% y un 0,4% respectivamente. El *Internet Mail Consortium* (IMC) recomendó que todos los programas de correo electrónico puedan mostrar y crear correo usando UTF-8, y el W3C recomienda UTF-8 como la codificación predeterminada en XML y HTML.

https://www.fileformat.info/info/charset/UTF-8/list.htm

3.4.2.7.2. UTF16

UTF-16 (Formato de transformación de Unicode de 16 bits) es una codificación de caracteres capaz de codificar todos los 1,112,064 puntos de código válidos de Unicode. La codificación es de longitud variable, ya que los puntos de código se codifican con una o dos unidades de código de 16 bits.

https://www.fileformat.info/info/charset/UTF-16/list.htm

UTF-16 surgió de una codificación anterior de 16 bits de ancho fijo conocida como UCS-2 (para el conjunto de caracteres universal de 2 bytes) una vez que quedó claro que se necesitaban más de 216 puntos de código. [1]

UTF-16 se usa internamente en sistemas como Windows y Java y en JavaScript, y con frecuencia para texto sin formato y para archivos de datos de procesamiento de texto en Windows. Rara vez se utiliza para archivos en Unix / Linux o macOS. Nunca ganó popularidad en la web, donde UTF-8 es dominante: UTF-16 es usado por menos del 0.01% de las páginas web mismas. [2] WHATWG recomienda que, por razones de seguridad, las aplicaciones del navegador no utilicen UTF-16.

3.4.2.7.3. UTF32

UTF-32 significa Formato de transformación Unicode en 32 bits. Es un protocolo para codificar puntos de código Unicode que utiliza exactamente 32 bits por punto de código Unicode (pero un número de bits iniciales debe ser cero, ya que hay menos de 221 puntos de código Unicode). UTF-32 es una codificación de longitud fija, en contraste con todos los demás formatos de transformación Unicode, que son codificaciones de longitud variable. Cada valor de 32 bits en UTF-32 representa un punto de código Unicode y es exactamente igual al valor numérico de ese punto de código.

https://www.fileformat.info/info/charset/UTF-32/list.htm

La principal ventaja de UTF-32 es que los puntos de código Unicode están directamente indexados. Encontrar el punto de código Nth en una secuencia de puntos de código es una operación de tiempo constante. En contraste, un código de longitud variable requiere acceso secuencial para encontrar el punto Nth en una secuencia. Esto convierte a UTF-32 en un reemplazo simple en el código que usa números enteros que se incrementan en uno para examinar cada ubicación en una cadena, como se hizo comúnmente para ASCII.

La principal desventaja de UTF-32 es que es ineficiente para el espacio, ya que utiliza cuatro bytes por punto de código. Los caracteres más allá del BMP son relativamente raros en la mayoría de los textos y, por lo general, pueden ignorarse para calcular el tamaño. Esto hace que UTF-32 sea casi el doble del tamaño de UTF-16. Puede ser hasta cuatro veces el tamaño de UTF-8 dependiendo de cuántos de los caracteres están en el subconjunto ASCII.

3.4.2.8. Norma ISO/IEC 25000 (SquaRE) – Evaluación del software

ISO 25000:2005 (SQuaRE -Software Quality Requirements and Evaluation) es una nueva serie de normas que se basa en ISO 9126 y en ISO 14598 (Evaluación del software). Uno de los principales objetivos de la serie SQuaRE es la coordinación y armonización del contenido de ISO 9126 y de ISO 15939:2002 (Measurement Information Model).

3.4.2.9. Norma ISO / IEC 25010 – Calidad del Código Fuente

ISO / IEC 25010 y Código de calidad ISO / IEC 25010 define la calidad del producto de software dividiéndolo en ocho características (características de calidad): "confiabilidad", "mantenibilidad", "portabilidad", "eficiencia", "seguridad", "Funcionalidad", "usabilidad" y "compatibilidad".

Entre ellas, "funcionalidad", "usabilidad" y "compatibilidad" se consideran las tres características de calidad que deben abordarse en una etapa temprana, preferiblemente antes de pasar a las fases de diseño en el proceso de upstream.

Considerando que, "confiabilidad", "mantenibilidad", "portabilidad", y también se considera la "eficiencia", son características de calidad que tienen una relevancia cercana con el desarrollo de un código fuente de alta calidad y, por lo tanto, deben examinarse en profundidad durante la fase de codificación.

La "seguridad", que se ha definido como la subcaracterística de calidad de la "funcionalidad" en la norma anterior, ISO / IEC 9126-1, se considera básicamente como una característica de calidad que es relevante en la fase de diseño, pero la codificación es para evitar el apilamiento. el desbordamiento también puede afectar la seguridad.

Para obtener más información sobre las prácticas de codificación relacionadas con la seguridad, consulte el "Estándar de codificación segura CERT C". Basada en la amplia categorización anterior, esta guía ha adoptado las últimas cuatro características de calidad: "confiabilidad", "mantenibilidad", "portabilidad" y "eficiencia" - como el enfoque principal, y recopiló las prácticas de codificación que están relacionadas principalmente con cualquier de estos cuatro. La Tabla 1 muestra la relación entre las "características de calidad" definidas en ISO / IEC 25010 y la "calidad de código" propuesta en esta guía, junto con las "subcaracterísticas de calidad".

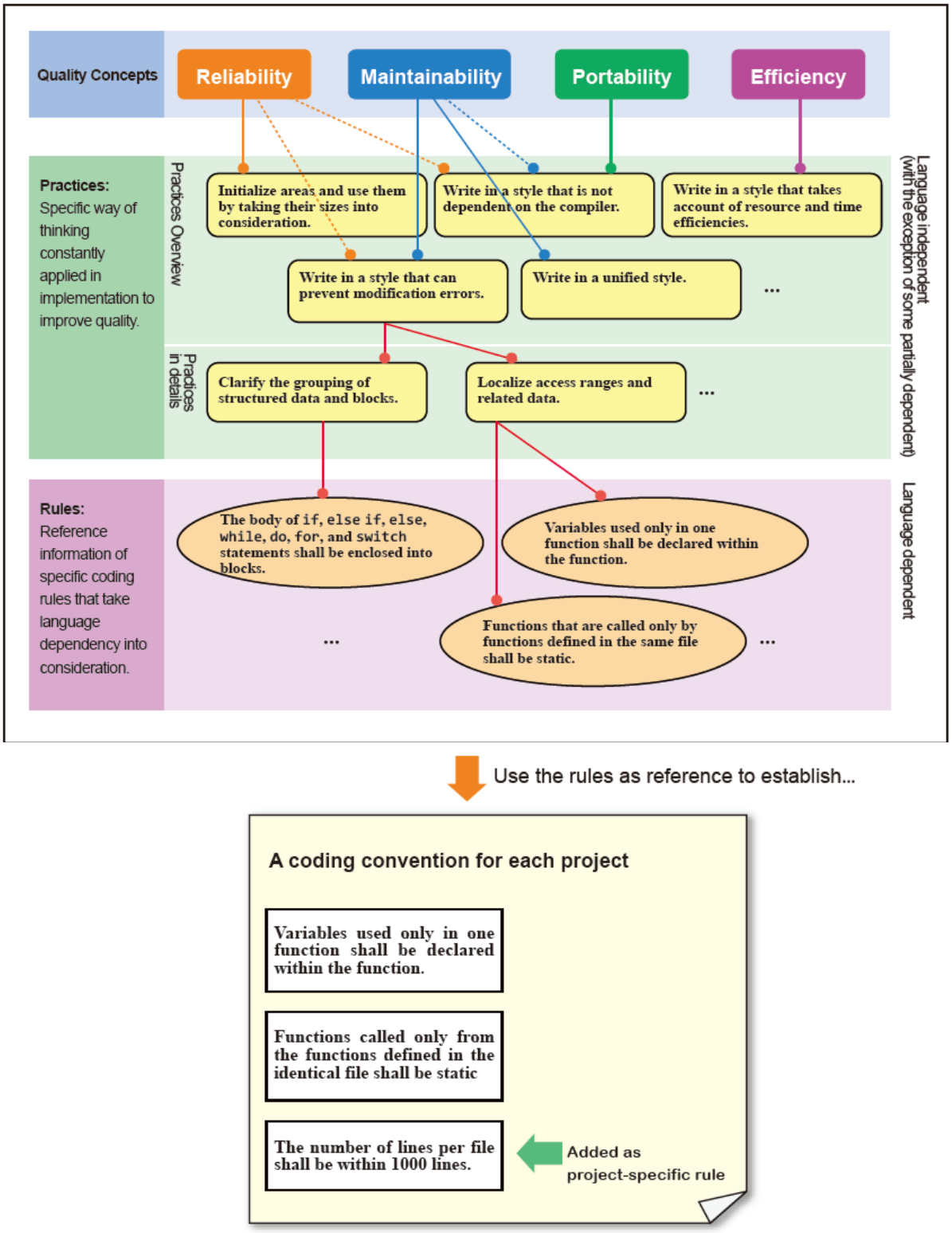


Figura 4 ISO / IEC 25010, diagrama de la Calidad del Código Fuente

3.4.2.9.1. Confiabilidad.

Una gran cantidad de software integrado se incorpora a los productos y se utiliza para respaldar nuestra vida diaria en diversas situaciones. En consecuencia, el nivel de fiabilidad exigido a una gran cantidad de software integrado es extremadamente alto. La confiabilidad del software requiere que el software sea capaz de no comportarse incorrectamente (no ocasionar fallas), no afecte la funcionalidad de todo el software y el sistema en caso de una falla de funcionamiento, restaure rápidamente su comportamiento normal después de una falla.

En el nivel del código fuente, el punto que se debe tener en cuenta con respecto a la confiabilidad del software es la necesidad de crear métodos para evitar la codificación que pueda causar tales fallas en la medida de lo posible.

Inicialice las áreas y utilícelas teniendo en cuenta sus tamaños.
Varias variables se utilizan en programas escritos en lenguaje C. Sin tener en cuenta las áreas que deben reservarse en la computadora y asegurarse de que estas áreas ya estén inicializadas en el momento en que se usen estas variables, pueden ocurrir fallas inesperadas.

Además, los punteros en lenguaje C deben usarse con cuidado al ser conscientes de las áreas a las que apuntan. Dado que el uso incorrecto de los punteros puede causar serios problemas a todo el sistema, es necesario tener especial cuidado al usarlos.
"Confiabilidad 1" consiste en las siguientes tres prácticas.

- Fiabilidad 1.1 Utilizar áreas después de inicializarlas.
- Fiabilidad 1.2 Describa las inicializaciones sin exceso o deficiencia.
- Fiabilidad 1.3 Preste atención a la gama del área señalada por un puntero

Utilice los datos teniendo en cuenta sus rangos, tamaños y representaciones internas.
Los datos utilizados en los programas varían en la forma en que se representan internamente y en el rango en el que se pueden operar, dependiendo de sus tipos. Cuando se utilizan estos diferentes tipos de datos para la operación, deben escribirse cuidadosamente prestando atención a varios aspectos, incluyendo precisión y tamaño. De lo contrario, pueden producirse fallos de funcionamiento inesperados cuando se procesan, como, por ejemplo, operaciones aritméticas. Por lo tanto, es necesario manejar los datos con cuidado, teniendo en cuenta sus rangos, tamaños y representaciones internas, entre otros.

Escriba de tal forma que asegure el comportamiento deseado.
Es esencial ser minucioso al describir cómo manejar todos los errores potenciales, teniendo en cuenta también los eventos inesperados que pueden ocurrir en casos que incluso se conciben como altamente improbables desde el punto de vista de las especificaciones del programa. Además, escribir el código de manera que no se base en las especificaciones de idioma, como la indicación explícita de la precedencia del operador también puede mejorar la seguridad. Para lograr una alta

confiabilidad, es deseable hacer todo lo posible para evitar la codificación que conduce a un mal funcionamiento y escribir de una manera que garantice el comportamiento deseado y la seguridad tanto como sea posible.

Los compiladores no garantizan el orden de ejecución (evaluación) de cada argumento real en funciones con múltiples parámetros. Los argumentos pueden ser ejecutados desde la derecha o desde la izquierda. Además, los compiladores no garantizan el orden de ejecución del lado izquierdo y del lado derecho de las operaciones binarias, como la operación +.

Por lo tanto, si el mismo objeto se actualiza y se hace referencia en una secuencia de argumentos o expresiones de operación binarias, el resultado de la ejecución no está garantizado.

Este problema, donde el resultado de la ejecución no está garantizado, se denomina problema de efectos secundarios. No escriba el código que causa tales problemas de efectos secundarios.

Esta regla, sin embargo, no prohíbe las descripciones, como las que se muestran a continuación que no causan el problema de los efectos secundarios.
x = x + 1;
x = f (x);

Características de la calidad (ISO/IEC 25010)		Sub-características de la calidad (ISO/IEC 25010)		Código de calidad
Confiabilidad	Grado en el que un sistema, producto o componente desarrolla una función especificada bajo ciertas condiciones por un tiempo específico	Madurez	Grado en que un sistema satisface las necesidades de confiabilidad en condiciones normales de funcionamiento.	Baja ocurrencia de bugs a través del uso continuo
		Disponibilidad	Grado en que un sistema, producto o componente. Es operacional y accesible cuando se requiere para utilizar.	
		Tolerancia al fallo	Grado en que un sistema, producto o componente. Funciona según lo previsto a pesar de la presencia de hardware o fallas de software.	Tolerancia a los bugs e violaciones de interfaces, etc
		Recuperación	Grado en que, en caso de interrupción. o un fallo, un producto o sistema puede recuperar el Datos directamente afectados y restablecer el deseado. estado del sistema	

De los siguientes ejemplos encuentre el error (Documento de referencia: *Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, JapanEmbedded System development Coding Reference guide [C Language Edition*], Ver 2.0):

1. Pg. 54

Compliant example

```
Compliant example of (1)
extern char *mes[3];
…
char *mes[] = {"abc", "def", NULL};

Compliant example of (2)
extern char *mes[];
…
char *mes[] = {"abc", "def", NULL};

Compliant example of (1) and (2)
extern int var1[MAX];
…
int var1[MAX];
```

Non-compliant example

```
Non-compliant example of (1)
extern char *mes[];
…
char *mes[] = {"abc", "def", NULL};

Non-compliant example of (1) and (2)
extern int var1[];
…
int var1[MAX];
```

2. Pg. 55

Compliant example

```
char var1[MAX];
for (i = 0; i < MAX && var1[i] != 0; i++) {
   /* Even if 0s are not set in the var1 array,
   there is no risk of accessing outside the
   array range */
    …
```

Non-compliant example

```
char var1[MAX];
for (i = 0; var1[i] != 0; i++) {  /* If 0s are
   not set in the var1 array, there is a risk
   of accessing outside the array range */
    …
```

3. Pg. 55

Compliant example

```
int a[ 5 ] = { [ 0 ] = 1 };
```

Non-compliant example

```
int b[ ] = { [ 0 ] = 1 };
```

4. Pg. 56

Compliant example

```
#define MAX   1024
void func(void)  {
 int  a[MAX]; /* Compliant Secured an array of
                 largest length */
```

Non-compliant example

```
void func(int  n) {
 int  a[n];       /* Non-compliant Variable
                     length array */
```

5. Pg. 57

Compliant example

```
if (y != 0)
    ans = x/y;
```

Non-compliant example

```
ans = x/y;
```

6. Pg. 57

Compliant example

```
if (p != NULL)
   *p = 1;
```

Non-compliant example

```
*p = 1;
```

7. Pg. 58

Compliant example

```
p = malloc(BUFFERSIZE);
if (p == NULL)
  /* Error handling */
else
  *p = '\0';
```

Non-compliant example

```
p = malloc(BUFFERSIZE);
*p = '\0';
```

8. Pg. 59

Compliant example

```
int func(int para) {
  if (!((MIN <= para) && (para <= MAX)))
    return range_error;
  /* Normal processing */
   ...
}
```

Non-compliant example

```
int func(int para) {
  /* Normal processing */
   ...
}
```

9. Pg. 60

Compliant example

–

Non-compliant example

```
unsigned int calc(unsigned int n)
{
  if (n <= 1) {
    return 1;
  }
   return n * calc(n-1);
}
```

10. Pg. 61

Compliant example

```
/* else clause of an if-else if statement where
the else condition does not normally occur */
if (var1 == 0) {
    …
} else if (0 < var1) {
    …
} else {
    /* Write an exception handling process */
    …
}
…
if (var1 == 0) {
    …
} else if (0 < var1) {
    …
} else {
    /* NOT REACHED */
}
```

Non-compliant example

```
/* if-else if statement without the else clause
*/
if (var1 == 0) {
    …
} else if (0 < var1) {
    …
}
```

11. Pg 62

Compliant example

```
/* Default clause in a switch statement where
   the default condition does not normally occur
*/
 switch(var1) {
 case 0:
     …
     break;
 case 1:
     …
     break;
 default:
     /* Write an exception handling process */
     …
     break;
 }
 …
 switch(var1) {
 case 0:
     …
     break;
 case 1:
     …
     break;
 default:
     /* NOT REACHED */
     break;
 }
```

Non-compliant example

```
/* Switch statement without the default clause
*/
switch(var1) {
case 0:
    …
    break;
case 1:
    …
    break;
}
```

12. Pg 63

Compliant example

```
void func() {
  int i;
  for (i = 0; i < 9; i += 2) {
     …
```

Non-compliant example

```
void func() {
  int i;
  for (i = 0; i != 9; i += 2) {
     …
```

13. Pg 64

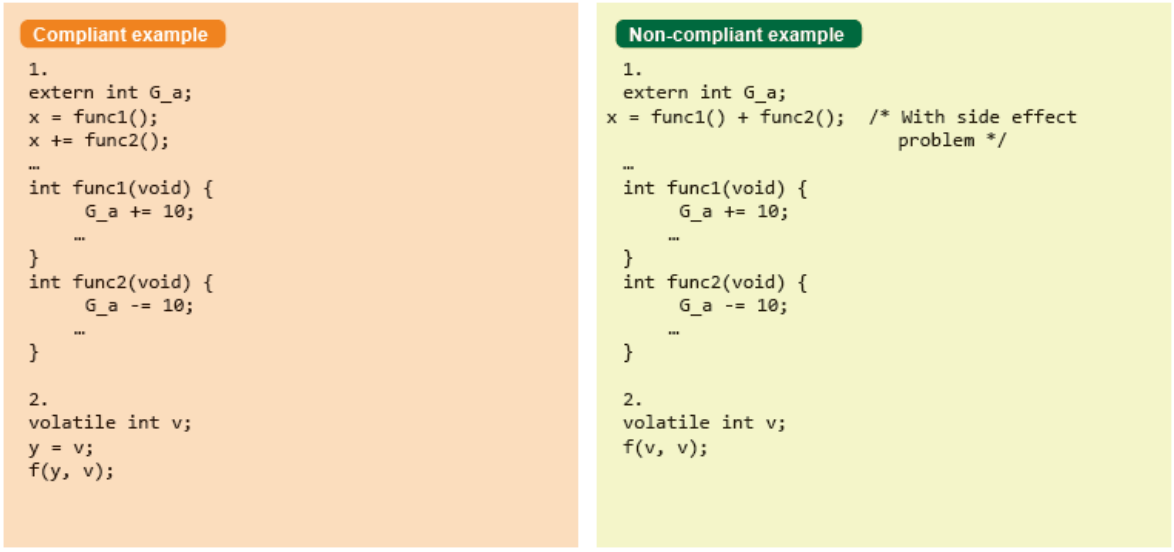
Compliant example

```
f (x, x);
x++;
- or -
f (x + 1, x);
x++;
```

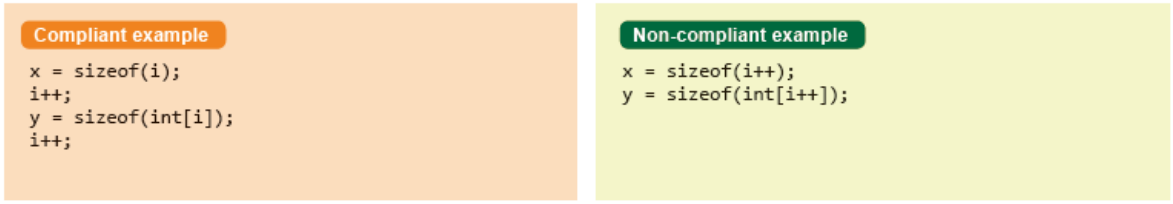
Non-compliant example

```
f (x, x++);
```

14. Pg. 65



15. Pg 66



3.4.2.9.2. Mantenibilidad.

Escritura de programas simple

Desde el punto de vista de la capacidad de mantenimiento del software, no hay mejor software que los creados a partir de programas escritos de forma simple.

El lenguaje C permite la estructuración del software, por ejemplo, al dividir el programa en archivos y funciones de origen separados. Programación estructurada que representa el programa.
La estructura a través de tres formas: secuencia, selección y repetición, es también una de las técnicas aplicables para escribir programas de software simples. Escribir descripciones de software simples a través del uso efectivo de la estructuración de software es altamente deseado Por otra parte, debe prestarse especial atención

Escribir en un estilo que no dependa del compilador

El uso de compiladores es inevitable cuando se programa en lenguaje C ++. Varios compiladores están disponibles en el mundo y cada uno tiene sus propias características. Si el código fuente está mal escrito, el código puede volverse dependiente de las características del compilador usado y puede causar resultados inesperados cuando se usa un compilador diferente.

Por esta razón, la programación debe realizarse con cuidado, teniendo en cuenta que el código debe escribirse en un estilo que no dependa de la implementación.

- **Portabilidad 1.1.** No utilice funcionalidades que sean características avanzadas o definidas por la implementación.
- **Portabilidad 1.2.** Use solo los caracteres y las secuencias de escape definidas en el lenguaje estándar.
- **Portabilidad 1.3.** Confirmar y documentar representaciones de tipos de datos, especificaciones de comportamiento de funcionalidades avanzadas e implementación de las partes dependientes.
- **Portabilidad 1.4.** Para la inclusión del archivo fuente, confirme la implementación dependiente partes y escribir en un estilo que no depende de la implementación.
- **Portabilidad 1.5** Escriba en un estilo que no dependa del entorno utilizado para la compilación.

Estilo unificado

También se puede dar a los estilos de escritura aplicados a la descripción, como el procesamiento de iteración, la asignación y las operaciones, ya que algunos pueden hacer que el programa sea difícil de mantener.

Recientemente, el desarrollo de programas bajo los esfuerzos compartidos por múltiples programadores se ha convertido en un enfoque ampliamente aceptado en proyectos de software. Si estos programadores aplican diferentes estilos de codificación para escribir su parte asignada del código fuente, los revisores u otros programadores pueden luego tener dificultades para verificar lo que ha escrito cada programador. Además, si la denominación de las variables, la información que debe describirse en un archivo y el orden para describir la información, entre otros, no son uniformes, pueden surgir malentendidos inesperados o errores de tales inconsistencias. Es el por qué es deseable escribir, tanto como sea posible, el código fuente de acuerdo con un estilo de codificación unificado en un solo proyecto o dentro de la organización.

Características de la calidad (ISO/IEC 25010)		Sub-características de la calidad (ISO/IEC 25010)		Código de calidad
Mantenibilidad	Grado de efectividad y eficiencia con el que un producto o sistema puede ser modificado por el mantenedores intencionados	Modularidad	Grado en que un sistema o programa informático se compone de componentes discretos tales que una cambio a un componente tiene un impacto mínimo en otros componentes	Grado para que la componentes están compuestos tal que un cambiar a uno componente de el código tiene impacto mínimo en otro componentes
		Reusabilidad	Grado en que un activo puede ser utilizado en más de un sistema, o en la construcción de otros activos	Grado para que un código se puede utilizar en otros programas
		Analizable	Grado de eficacia y eficiencia con cual es posible evaluar el impacto en una producto o sistema de un cambio previsto a uno o más de sus partes, o para diagnosticar un producto para deficiencias o causas de fallos, o para identificar partes a modificar	Facilidad de comprensión el código
		Modificable	Grado en que un producto o sistema puede ser efectivamente y eficientemente modificado sin Introducción de defectos o degradación del producto existente. calidad	Facilidad de modificando el código, y la lentitud de impacto de modificaciones
		Pruebas	Grado de efectividad y eficiencia con qué criterios de prueba pueden establecerse para una sistema, producto o componente y pruebas pueden ser realizado para determinar si esos criterios se han cumplido	Facilidad de pruebas y depurando el código modificado

De los siguientes ejemplos encuentre el error (Documento de referencia: *Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, JapanEmbedded System development Coding Reference guide [C Language Edition]*, Ver 2.0):

1. Pg. 69

Compliant example

```
void func(void) {

When necessary in call back function
int cbfunc1(int arg1, int arg2);
int cbfunc2(int arg1, int);
 /* In case the call back function types are
    fixed to be int(*)(int,int), the second
    argument is necessary even when it is not
    used. */
```

Non-compliant example

```
void func(int arg) {
       /* arg is unused */
```

2. Pg 69

Compliant example

```
…
#if  0   /* Ineffective due to ~ */
    a++;
#endif
…
```

Non-compliant example

```
…
/* a++; */
…
```

3. Pg 70

Compliant example

```
Compliant example of (1)
int i;
int j;
Compliant example of (2)
int i, j;
int k = 0;

int *p;
int i;
```

Non-compliant example

```
Non-compliant example of (1)
int i, j;
Non-compliant example of (2)
int i, j, k = 0;
  /* A variable with
     initialization and variables without
     initialization are mixed (Non-compliant) */

int *p, i;
 /* Variables of different types
    are mixed (Non-compliant) */
```

4. Pg 71

Compliant example

```
void func(long int);
…
float f;
long int l;
unsigned int ui;

f = f + 1.0F;  /* Explicitly state that it is a
                  float operation */
func(1L); /* Description of L should be an
             uppercase letter */
if (ui < 0x8000U) { /* Explicitly state that it
                       is an unsigned comparison */
    …
```

Non-compliant example

```
void func(long int);
…
float f;
long int l;
unsigned int ui;

f = f + 1.0;
func(1l); /* 1l (numeral "1" and alphabet letter
             "l" ) can get easily confused with 11
             (numeral "1" and numeral "1" ). */
if (ui < 0x8000) {
    …
```

5. Pg 71

Compliant example

```
char abc[] = "aaaaaaaa¥n"
             "bbbbbbbb¥n"
             "cccccccc¥n";
```

Non-compliant example

```
char abc[] = "aaaaaaaa¥n¥
              bbbbbbbb¥n¥
              cccccccc¥n";
```

6. Pg 72

Compliant example

```
if (i_var1 == 0) {
  i_var2 = 0;
} else {
  i_var2 = 1;
}
```

Non-compliant example

```
switch (i_var1 == 0) {
case 0:
  i_var2 = 1;
  break;
default:
  i_var2 = 0;
  break;
}
```

7. Pg 72

Compliant example

```
switch (x) {
case 1:
    {
      …
    }
    …
    break;
case 2:
    …
    break;
default:
    …
    break;
}
```

Non-compliant example

```
switch (x) { /* Compound statement of the
                switch statement body */
case 1:
    {  /* Nested compound statement */
case 2: /* Do not describe case label in
           nested compound statement */
        …
    }
    …
    break;
default:
    …
    break;
}
```

8. Pg 73

Compliant example

```
extern int global;

int func(void) {
    …
}
```

Non-compliant example

```
extern global;

func(void) {
    …
}
```

9. Pg 73

Compliant example

```
if ((x > 0) && (x < 10))
if ((x != 1) && (x != 4) && (x != 10))
if (flag_tb[i] && status)
if (!x || y)
```

Non-compliant example

```
if (x > 0 && x < 10)
if (x != 1 && x != 4 && x != 10)
```

10. Pg 74

Compliant example

```
a = (b << 1) + c;
        - or -
a = b << (1 + c);
```

Non-compliant example

```
a = b << 1 + c; /* There is a possibility that
                   the operator precedence is
                   misunderstood */
```

11. Pg 74

Compliant example

```
void  func(void);
void (*fp)(void) = &func;

if (func()) {
```

Non-compliant example

```
void  func(void);
void (*fp)(void) = func; /* Non-compliant:
                            There is no & */
if (func) {   /* Non-compliant: Address is
                 obtained rather than calling the
                 function. It might be mistakenly
                 written as afunction call without
                 arguments. */
```

12. Pg 75

Compliant example

```
int x = 5;

if (x != 0) {
 …
}
```

Non-compliant example

```
int x = 5;

if (x) {
 …
}
```

13. Pg 75

Compliant example

```
 /* Counter variable and work variable for
    replacement are different */
for (i = 0; i < MAX; i++) {
    data[i] = i;
}
if (min > max) {
    wk = max;
    max = min;
    min = wk;
}
```

Non-compliant example

```
 /* Counter variable and work variable for
    replacement are the same */
for (i = 0; i < MAX; i++) {
    data[i] = i;
}
if (min > max) {
    i = max;
    max = min;
    min = i;
}
```

14. Pg 76

Compliant example

```
compliant example of (2)
/* When the typ is INT, i_var is valid.
   When the type is CHAR, c_var[4] is valid. */
struct stag {
  int type;
  union utag {
    char c_var[4];
    int i_var;
  } u_var;
} s_var;
…
int i;
…
if (s_var.type == INT) {
    s_var.u_var.i_var = 1;
}
…
i = s_var.u_var.i_var;
```

Non-compliant example

```
Non-compliant example of (2)
/* When the typ is INT, i_var is valid.
   When the type is CHAR, c_var[4] is valid. */
struct stag {
  int type;
  union utag {
    char c_var[4];
    int i_var;
  } u_var;
} s_var;
…
int i;
…
if (s_var.type == INT) {
    s_var.u_var.c_var[0] = 0;
    s_var.u_var.c_var[1] = 0;
    s_var.u_var.c_var[2] = 0;
    s_var.u_var.c_var[3] = 1;
}
…
i = s_var.u_var.i_var;
```

15. Pg 77

Compliant example

```
int var1;
void func(int arg1) {
  int var2;
  var2 = arg1;
  {
    int var3;
    var3 = var2;
    …
  }
}
```

Non-compliant example

```
int var1;
void func(int arg1) {
  int var1;    /* The same name of a variable
                  outside the function is used
*/
  var1 = arg1;
  {
     int var1;  /* The same name of a variable
                   in the outer scope is used */
     …
     var1 = 0;  /* Intention of which var1 is
                   assigned is unclear */
     …
  }
}
```

16. Pg78

Compliant example

```
#include <string.h>
void *my_memcpy(void *arg1, const void *arg2,
size_t size) {
     …
}
```

Non-compliant example

```
#undef NULL
#define NULL  ((void *)0)

#include <string.h>
void *memcpy(void *arg1, const void *arg2, size_
t size) {
     …
}
```

17. Pg 79

Compliant example

```
_
```

Non-compliant example

```
int _Max1;   /* Reserved */
int __max2;  /* Reserved */
int _max3;   /* Reserved */

struct S {
   int _mem1; /* Not reserved, but shall not be
                 used */
};
```

18. Pg 80

Compliant example

```
#define   START 0x0410
#define   STOP  0x0401
```

Non-compliant example

```
#define  BIGIN  {
#define  END    }
#define  LOOP_STAT for(;;) {
#define  LOOP_END  }
```

19. Pg 81

Compliant example

```
s = "abc?(x)";
```

Non-compliant example

```
s = "abc??(x)";  /* Compilers that can process
                    trigraph sequences interpret
                    this as “abc[x)” */
```

20. Pg 82

Compliant example

```
for (;;) {
     /* Waiting for interruption */
}

#define NO_STATEMENT
i = COUNT;
while ((--i) > 0) {
    NO_STATEMENT;
}
```

Non-compliant example

```
for (;;) {
}

i = COUNT;
while ((--i) > 0);
```

21. Pg 83

Compliant example

```
#define MAXCNT 8
if (cnt == MAXCNT) {
  …
```

Non-compliant example

```
if (cnt == 8) {
  …
```

22. Pg 84

Compliant example

```
const volatile int read_only_mem; /* Read-only
                                     memory */
const int constant_data = 10; /* Read-only
                         data that does not
                         require memory
                         allocation */
/* Only reads the contents pointed by arg */
void func(const char *arg, int n) {
   int   i;
   for (i = 0; i < n; i++) {
      put(*arg++);
   }
}
```

Non-compliant example

```
int read_only_mem;   /* Read-only memory */
int constant_data = 10; /* Read-only data that
                    does not require memory
                    allocation */
/* Only reads the contents pointed by arg */

void func(char *arg, int n) {
   int   i;
   for (i = 0; i < n; i++) {
      put(*arg++);
   }
}
```

23. Pg 86

```
#if 0
 /* */
#endif

#if 0
…
#else
 int var;
#endif

#if 0
 /* I don't know */
#endif
```

```
#if 0
 /*
#endif

#if 0
…
#else1
 int var;
#endif

#if 0
 I don't know
#endif
```

24. Pg 88

Compliant example

```
int arr1[2][3] = {{0, 1, 2}, {3, 4, 5}};
int arr2[3] = {1, 1, 0};
```

Non-compliant example

```
int arr1[2][3] = {0, 1, 2, 3, 4, 5};
int arr2[3] = {1, 1};
```

25. Pg 89

Compliant example

```
void func1(void)
{
  static int x = 0;
  if (x != 0) {    /* Refer to the value in the
                   immediately preceding call */
      x++;
  }
    …
}
void func2(void)
{
  int  y = 0;    /* Initialize each time */
   …
}
```

Non-compliant example

```
int x = 0; /* x is accessed only from func1 */
int y = 0; /* y is accessed only from func2 */
void func1(void) {
  if ( x != 0 ) {  /* Refer to the value in the
                   immediately preceding call */
      x++;
  }
   …
}
void func2(void) {
  y = 0;      /* Initialize each time */
   …
}
```

26. Pg 93

Compliant example

```
end = 0;
for (i=0; loop iteration condition && !end; i++)
{
  Iterated processing 1;
  if (termination condition1 || termination
      condition2) {
    end = 1;
  } else {
    Iterated processing 2;
  }
}
-or-
for (i=0; loop iteration condition; i++) {
  Iterated processing 1;
  if (termination condition1 || termination
      condition2) {
    break;
  }
  Iterated processing 2;
}
```

Non-compliant example

```
for (i=0; loop iteration condition; i++) {
  Iterated processing 1;
  if (termination condition1) {
    break;
  }
  if (termination condition1) {
    break;
  }
  Iterated processing 2;
}
```

27. Pg 95

Compliant example

```
Compliant example of (1) and (2)
switch (week) {
case A:
  code = MON;
  break;
case B:
  code = TUE;
  break;
case C:
  code = WED;
  break;
default:
  code = ELSE;
  break;
}

Compliant example of (2)
dd = 0;
switch (status) {
case A:
  dd++;
  /* FALL THROUGH */
case B:
```

Non-compliant example

```
Non-compliant example of (1) and (2)
/* No matter what the value of week is, the
   code will be ELSE ==> bug */
switch (week) {
case A:
  code = MON;
case B:
  code = TUE;
case C:
  code = WED;
default:
  code = ELSE;
}

/* This This is a case where processing of case
   B is continued after dd++, but it is non-
   compliant not only to (1) but also to (2)
   because there is no comment. */
dd = 0;
switch (status) {
case A:
  dd++;
case B:
```

28. Pg 96

Compliant example

```
Compliant example of (1) and (2)
a = 1;
b = 1;

j = 10;
for (i = 0; i < 10; i++) {
   …
   j--;
}

Compliant example of (2)
for (i = 0, j = 10; i < 10; i++, j--) {
   …
}
```

Non-compliant example

```
Non-compliant example of (1) and (2)
a = 1, b = 1;

Non-compliant example of (1)
for (i = 0, j = 10; i < 10; i++, j--) {
   …
}
```

29. Pg 97

Compliant example

```
for (i = 0; i < MAX; i++) {
   …
  j++;
}
```

Non-compliant example

```
for (i = 0; i < MAX; i++, j++) {
  …
}
```

30. Pg 99

Compliant example

```
int **p;
typedef char **strptr_t;
strptr_t q;
```

Non-compliant example

```
int ***p;
typedef char **strptr_t;
strptr_t *q;
```

31. Pg 102

[Method 1] Write an operator at the end of the line.

Example :

```
x = var1 + var2 + var3 + var4 +
    var5 + var6 + var7 + var8 + var9;
if (var1 == var2 &&
    var3 == var4)
```

[Method 2] Write an operator at the beginning of the continuation line.

Example :

```
x = var1 + var2 + var3 + var4
    + var5 + var6 + var7 + var8 + var9;
if (var1 == var2
    && var3 == var4)
```

32. Pg 103. Indentation

```
(1) Example of K&R style
void func(int arg1)
{ /* Write the { of a function on a
     new line */
   /* Indent is 1 tab */
    if (arg1) {
        …
    }
    …
}
```

```
(2) Example of BSD style
void
func(int arg1)
{ /* Write the { of a function on a
     newline */
    if (arg1)
    {
       …
    }
    …
}
```

```
(3) Example of GNU style
void
func(int arg1)
{ /* Write the { of a function on a
     new line at column 0 */
  if (arg1)
    {
       …
    }
    …
}
```

33. Pg 111

Compliant example

```
--- my_inc.h ---
extern int   x;
int   func(int);

--------------
#include "my_inc.h"
int x;
int func(int in)
{
  …
```

Non-compliant example

```
/* Declaration of variable x and function func
   are missing */
int x;
int func(int in)
{
   …
```

34. Pg 111

Compliant example

```
int  x;  /* Definition of one external variable
            shall be only once */
```

Non-compliant example

```
int  x;
int  x;  /* Definition of an external variable
            in multiple locations does not cause
            a compile error */
```

35. Pg 112

Compliant example

```
--- file1.h ---
extern int  x;    /* Variable declaration */
int  func(void);  /* Function declaration */

--- file1.c ---
#include   "file1.h"
int  x;           /* Variable definition */
int  func(void)   /* Function definition */
{
  …
}
```

Non-compliant example

```
--- file1.h ---
int  x;  /* External variable definition */
static int  func(void) /* Function definition */
{
   …
}
```

36. Pg 113

Compliant example

```
--- myheader.h ---
#ifndef MYHEADER_H
#define MYHEADER_H
    Contents of the header file
#endif     /* MYHEADER_H */
```

Non-compliant example

```
--- myheader.h ---
void func(void);
/* end of file */
```

37. Pg 113

Compliant example

```
Compliant example of (1)
int func1(int, int);

int func1(int x, int y)
{
   /* Process the function */
}

Compliant example of (2)
int func1(int x, int y);
int func2(float x, int y);

int func1(int x, int y)
{
   /* Process the function */
}

int func2(float x, int y)
{
   /* Process the function */
}
```

Non-compliant example

```
Non-compliant example of (1) and (2)
int func1(int x, int y);
int func2(float x, int y);

int func1(int y, int x)  /* The parameter
                            name differs from
                            the name in the
                            prototype
                            declaration */
{
   /* Process the function */
}

typedef  int INT;
int func2(float x, INT y) /* The type of y is
                             not literally the
                             same as in the
                             prototype
                             declaration */
{
   /* Process the function */
}
```

38. Pg 114

Compliant example

```
struct TAG  {
 int mem1;
 int mem2;
};
struct TAG x;
```

Non-compliant example

```
struct TAG  {
 int mem1;
 int mem2;
} x;
```

39. Pg 114

Compliant example

```
Compliant example of (1)
struct tag  data[] = {
  { 1, 2, 3 },
  { 4, 5, 6 },
  { 7, 8, 9 }    /* There is no comma after the
                    last element */
};
Compliant example of (2)
struct tag  data[] = {
  { 1, 2, 3 },
  { 4, 5, 6 },
  { 7, 8, 9 },    /* There is a comma after the
                     last element */
};
```

Non-compliant example

```
Non-compliant example of (1) and (2)
struct tag  x = { 1, 2, };
/* Not clear whether there are only two members
   or there are three or more */
```

40. Pg 115

Compliant example

```
Compliant example of (1)
char *p;
int dat[10];

p = 0;
dat[0] = 0;

Compliant example of (2)
char *p;
int dat[10];

p = NULL;
dat[0] = 0;
```

Non-compliant example

```
Non-compliant example of (1)
char *p;
int dat[10];

p = NULL;
dat[0] = NULL;

Non-compliant example of (2)
char *p;
int dat[10];

p = 0;
dat[0] = NULL;
```

41. Pg 116

Compliant example

```
#define M_SAMPLE(a, b) ((a)+(b))
```

Non-compliant example

```
#define M_SAMPLE(a, b) a+b
```

42. Pg 116

Compliant example

```
#ifdef  AAA
/* Process when AAA is defined */
  …
#else      /* not AAA */
/* Process when AAA is not defined */
  …
#endif     /* end AAA */
```

Non-compliant example

```
#ifdef  AAA
/* Process when AAA is defined */
  …
#else
/* Process when AAA is not defined */
  …
#endif
```

43. Pg 117

Compliant example

```
#if  defined(AAA)
…
#endif
- or -
#if defined AAA
…
#endif
```

Non-compliant example

```
#if  AAA
  …
#endif
- or -
#defined DD(x) defined(x)
#if  DD(AAA)
  …
#endif
```

44. Pg 117

Compliant example

```
#define   AAA    0
#define   BBB    1
#define   CCC    2
struct stag {
  int    mem1;
  char  *mem2;
};
```

Non-compliant example

```
/* Members with restriction on the assignable
   values exist */
struct stag {
  int mem1;  /* The following values are
                assignable: */
#define  AAA   0
#define  BBB   1
#define  CCC   2
  char *mem2;
};
```

45. Pg 119

Compliant example

```
#define TRUE  1
#define FALSE 0
#if TRUE
  …
#if defined(AAA)
  …
#if VERSION == 2
  …
#if 0  /* Invalidated due to ~ */
 …
```

Non-compliant example

```
#define ABC 2
#if ABC
  …
```

46. Pg 123

Compliant example

```
Compliant example of (2)
#define AAA(a, b)  a#b
#define BBB(x, y)  x##y
```

Non-compliant example

```
Non-compliant example of (1) and (2)
#define  XXX(a, b, c)     a#b##c
```

47. Pg 124

Compliant example

```
int func(int arg1, int arg2)
{
  return arg1 + arg2;
}
```

Non-compliant example

```
#define func(arg1, arg2) (arg1 + arg2)
```

3.4.2.9.3. Portabilidad y Eficiencia

Portabilidad

Uno de los aspectos distintivos del software integrado es que existen diversas opciones en la plataforma utilizada para la operación del software.

Esto también significa que hay muchas combinaciones posibles de opciones de MPU y opciones de SO para seleccionar las plataformas de hardware y software. A medida que aumenta la cantidad de funcionalidades realizadas por el software integrado, también aumentan las oportunidades de transferir el software existente a otras plataformas mediante la modificación o remodelación para que sea compatible con múltiples plataformas.

Debido a esta tendencia, la portabilidad del software se está convirtiendo en un elemento extremadamente importante también en el nivel del código fuente. En particular, escribir en un estilo que depende de la implementación es uno de los errores más comunes que se cometen regularmente.

- Portabilidad 1: escriba en un estilo que no depende del compilador.
- Portabilidad 2: localice el código que tiene un problema con la portabilidad.

Escriba en un estilo que no dependa del compilador.

El uso de compiladores es inevitable cuando se programa en lenguaje C ++. Varios compiladores están disponibles en el mundo y cada uno tiene sus propias características. Si el código fuente está mal escrito, el código puede depender de las características del compilador utilizado y puede generar resultados inesperados cuando se usa un compilador diferente.

Por esta razón, la programación debe realizarse con cuidado, con la precaución de que el código debe escribirse en un estilo que no dependa de la implementación.

- **Portabilidad 1.1** No utilice funcionalidades que sean características avanzadas o definidas por la implementación.
- **Portabilidad 1.2** Use solo los caracteres y las secuencias de escape definidas en el lenguaje estándar.
- **Portabilidad 1.3** Confirmar y documentar representaciones de tipos de datos, comportamentales, Especificaciones de funcionalidades avanzadas y de implementación de las partes dependientes.
- **Portabilidad 1.4** Para la inclusión del archivo fuente, confirme las partes dependientes de la implementación y escriba en un estilo que no dependa de la implementación.
- **Portabilidad 1.5** Escriba en un estilo que no dependa del entorno utilizado para la compilación.

Localiza el código que tiene problemas con la portabilidad.

El principio es no escribir código fuente dependiente de la implementación tanto como sea posible.

Pero en algunos casos, esto puede ser inevitable. Un ejemplo típico es cuando se llama a un programa en lenguaje ensamblador desde lenguaje C. En tal caso, se recomienda localizar las partes del código que dependen de la implementación tanto como sea posible.

- **Portabilidad 2.1** Localice el código que tiene un problema con la portabilidad.

Eficiencia.
El software embebido es característico por estar integrado en un producto y operar junto con hardware para cumplir sus propósitos en el mundo real. La creciente demanda de una mayor reducción del costo del producto ha impuesto varias restricciones, no solo en MPU o memoria, sino también en software.

Además, los requisitos, tales como, en la propiedad en tiempo real han colocado restricciones de tiempo más estrictas que deben cumplirse.

Por lo tanto, el software integrado debe codificarse con especial atención a la eficiencia de los recursos, como el uso eficiente de la memoria y la eficiencia del tiempo que tiene en cuenta el rendimiento del tiempo.

- **Eficiencia1:** Escriba en un estilo que tenga en cuenta la eficiencia de recursos y tiempo.
  Dependiendo de cómo se escriba el código fuente, el tamaño del objeto puede aumentar y la velocidad de ejecución puede disminuir. Si hay restricciones en el tamaño de la memoria y el tiempo de procesamiento, el código debe escribirse cuidadosamente con consideraciones adicionales dadas a estas restricciones.

• Eficiencia 1.1 Escriba en un estilo que tenga en cuenta la eficiencia de recursos y tiempo.

Características de la calidad (ISO/IEC 25010)		Sub-características de la calidad (ISO/IEC 25010)		Código de calidad
Portabilidad	Grado en el que un sistema, producto o componente desarrolla una función especificada bajo ciertas condiciones por un tiempo específico	Adaptabilidad	Grado en que un producto o sistema puede adaptarse de manera efectiva y eficiente para hardware, software u otros entornos operativos o de uso diferentes o en evolución	Facilidad de adaptación a diferentes entornos * Incluida la conformidad con los estándares
		Instalabilidad	Grado de efectividad y eficiencia con el que un producto o sistema se puede instalar y / o desinstalar con éxito en un entorno específico	
		Reemplazabilidad	Grado en que un producto puede ser reemplazado por otro producto de software específico para el mismo propósito en el mismo entorno	
Eficiencia en performance	Grado en el que un sistema, producto o componente desarrolla una función especificada bajo ciertas condiciones por un tiempo específico	Tiempo del comportamiento	Grado en que los tiempos de respuesta y procesamiento y las tasas de rendimiento de un producto o sistema, al realizar sus funciones, cumplen con los requisitos	Eficiencia con respecto al tiempo de procesamiento.
		Uso de recursos	Grado en que las cantidades y los tipos de recursos utilizados por un producto o sistema al realizar sus funciones cumplen con los requisitos	Eficiencia con respecto a recursos
		Capacidad.	Grado en que los límites máximos de un producto o parámetro del sistema cumplen con los requisitos	

Portabilidad

De los siguientes ejemplos encuentre el error (Documento de referencia: *Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, JapanEmbedded System development Coding Reference guide [C Language Edition]*, Ver 2.0):

1. Pg 131

Compliant example

```
char c = '\t';   /* compliant */
```

Non-compliant example

```
char c = '\e'; /* Non-compliant: The escape is
                  sequence not defined in the
                  language standard. It is not
                  portable */
```

2. Pg 132

Compliant example

```
char c = 'a';    /* Used to store characters */
int8_t  i8 = -1; /* To use it as 8-bit data,
                    use a type defined, for
                    example, with typedef */
```

Non-compliant example

```
char c = -1;
if (c > 0) { … }
/* Non-compliant: char can be signed or unsigned
   depending on the compiler, and this difference
   affects the result of the comparison. */
```

3. Pg 133

Compliant example

```
/* If int is 16bits and long is 32bits */
enum largenum {
  LARGE = INT_MAX
};
```

Non-compliant example

```
/* If int is 16bits and long is 32bits */
enum largenum {
  LARGE = INT_MAX+1
};
```

4. Pg 133

Compliant example

```
Compliant example of (2)
struct S {
   unsigned int bit1:1;
   unsigned int bit2:1;
};
extern struct S * p;   /* Compliant if p points
                          to a date that is, for
                          example, just a set of
                          flags and bit1 can be
                         any bit in that data */

p->bit1 = 1;
```

Non-compliant example

```
Non-compliant example of (2)
struct S {
   unsigned int bit1:1;
   unsigned int bit2:1;
};
extern struct S * p;
/* If the bit positions are meaningful, for
   example, when p points at IO ports; in
   other words, if there is a meaning for bit1
   to point at either the lowest bit or the
   highest bit of the data, the program is non-
   portable */

p->bit1 = 1;   /* As to which bit of the data,
                  p will point at, is
                  implementation-dependent */
```

5. Pg 134

Compliant example

```
#include <stdio.h>
#include  "myheader.h"
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h"
#endif
#include INCFILE
```

Non-compliant example

```
#include  stdio.h
   /* Neither <> nor “ ” is placed */
#include  "myheader.h"  1
   /* 1 is specified at the end */
```

6. Pg 135

Compliant example

```
#include <stdio.h>
#include "myheader.h"
```

Non-compliant example

```
#include "stdio.h"
#include <myheader.h>
```

7. Pg 135

Compliant example

```
#include "inc/my_header.h"  /* compliant */
```

Non-compliant example

```
#include "inc\my_header.h"  /* Non-compliant */
```

8. Pg 136

Compliant example

```
#include "h1.h"
```

Non-compliant example

```
#include "/project1/module1/h1.h"
```

9. Pg 138

Compliant example

```
#define SET_PORT1  asm("   st.b 1, port1")
void f() {
  …
  SET_PORT1;
  …
}
```

Non-compliant example

```
void f() {
  …
  asm("  st.b 1,port1");
  …
}
/* asm and other processes are mixed */
```

10. Pg 138

Compliant example

```
/* interrupt is defined as a keyword extended by
   a specific compiler. */
#define INTERRUPT interrupt
INTERRUPT  void int_handler (void) {
  …
}
```

Non-compliant example

```
/* interrupt is defined as a keyword extended
   by a specific compiler. It is used without
   being defined by a macro */
interrupt  void int_handler(void) {
  …
}
```

11. Pg 139

Compliant example

```
Compliant example of (1) and (2)
uint32_t flag32;   /* Use unit32_t if 32bits is
                                    assumed */

Compliant example for (2)
int i;
for (i = 0; i < 10; i++) {  … }
/* i is used as an index. It can be 8bits,
   16bits or 32bits and a basic type in the
   language specification can be used for i */
```

Non-compliant example

```
Non-compliant example of (1) and (2)
unsigned int flag32;  /* used by assuming int as
                                    32bits */
```

Eficiencia

De los siguientes ejemplos encuentre el error (Documento de referencia: *Software Reliability Enhancement Center, Technology Headquarters, Information-technology Promotion Agency, Japan, Embedded System Development Coding Reference guide [C Language Edition],* Ver 2.0):

1. Pg 143.

Compliant example

```
extern void func1(int,int); /* func1: called
                               only once */
#define func2(arg1, arg2)   /*  func2: called
                               many times */

func1(arg1, arg2);
for  (i = 0; i < 10000; i++) {
  func2(arg1, arg2);  /* Speed performance is
                         critical for this
                         process. */
}
```

Non-compliant example

```
#define func1(arg1, arg2)   /* func1: called only
                               once */
extern void func2(int, int); /* func2: called
                                many times */

func1(arg1, arg2);
for (i = 0; i < 10000; i++) {
  func2(arg1, arg2);  /* Speed performance is
                         critical for this
                         process. */
}
```

2. Pg 143.

Compliant example

```
var1 = func();
for (i = 0; (i + var1) < MAX; i++) {
  …
}
```

Non-compliant example

```
/* Function func returns the same result */
for (i = 0; (i + func()) < MAX; i++) {
  …
}
```

3. Pg 144

Compliant example

```
typedef struct stag {
  int mem1;
  int mem2;
    …
}  STAG;
int func (const STAG *p) {
  return p->mem1 + p->mem2;
}
```

Non-compliant example

```
typedef struct stag {
  int mem1;
  int mem2;
    …
}  STAG;
int func (STAG x) {
  return x.mem1 + x.mem2;
}
```

3.4.2.9.4. Seguridad.

La seguridad es un requisito no funcional que tiene una repercusión extraordinaria en la calidad de los productos software.

De hecho, la seguridad informática ha sido un campo que ha crecido enormemente desde los años 70, dando lugar a una gran cantidad de técnicas, modelos, protocolos, etc., que han venido acompañados también de una actividad muy pronunciada por parte de las organizaciones internacionales de normalización y certificación.

Tanto es así, que se pueden encontrar numerosas organizaciones internacionales de estandarización que han producido una compleja estructura de estándares relativos a temáticas relacionadas con la seguridad informática, que cambian y se actualizan con mucha frecuencia.

Existen numerosas definiciones de seguridad. Lo habitual es que todas ellas definan la seguridad en términos de otros conceptos relacionados. Por ejemplo, la "protección de información procesada por un computador frente a consultas no autorizadas, modificaciones inapropiadas o la falta de disponibilidad de un servicio en un momento dado". Otra definición clásica es: la seguridad como un sub-factor de la calidad del software, y la define como "la capacidad de los productos de software para proteger los datos y la información para que las personas o sistemas no autorizados no puedan leerla o modificarla y para que el acceso no sea rechazado al personal autorizado". En ambas definiciones están presentes los conceptos de confidencialidad, integridad y disponibilidad. Sin embargo, hay otras definiciones algo más recientes que consideran, además, otras propiedades importantes, como son la autenticación, el no repudio y la autorización y control de acceso. Aunque la seguridad se puede interpretar como un aspecto estrictamente técnico, hay autores que piensan que es mucho más que eso, teniendo por el contrario una dimensión estratégica, resultando uno de los criterios más importantes en el gobierno de las TIC.

Sin embargo, aunque también se han desarrollado ampliamente en las últimas décadas las técnicas y metodologías propias de la ingeniería del software, éstas no han considerado la seguridad como un aspecto importante del desarrollo, dejando que la construcción metodológica del software se centre fundamentalmente en los requisitos funcionales, y algunos otros requisitos no funcionales, y relegando los requisitos de seguridad a un momento tardío en el proceso de desarrollo de software. Algunas propuestas interesantes que tratan la seguridad, aunque de manera parcial y sin ofrecer un claro seguimiento de esos aspectos de seguridad a lo largo del proceso de desarrollo incluyen.

Características de la calidad (ISO/IEC 25010)		Sub-características de la calidad (ISO/IEC 25010)		Código de calidad
Seguridad	Grado en que un producto o sistema protege la información y datos para que las personas u otros productos o sistemas tengan el grado de acceso a los datos apropiados a su tipos de niveles y de autorización.	Confidencialidad	Grado en que un producto o sistema garantiza que los datos sean accesibles solo para aquellos autorizados para tener acceso	Grado de certeza que los datos son accesibles solo para aquellos autorizado para tener acceso
		Integridad	Grado en que un sistema, producto o componente impide el acceso no autorizado o la modificación de programas o datos informáticos.	Grado de prevención de acceso no autorizado o modificación de programas o datos informáticos
		No-repudio	Grado en que se puede demostrar que se han llevado a cabo acciones o eventos, de modo que los eventos o acciones no pueden ser repudiados posteriormente	
		Accountability	Grado en que las acciones de una entidad se pueden rastrear de manera única a la entidad	
		Autenticidad	Grado en el que se puede demostrar que la identidad de un sujeto o recurso es la reivindicada	

3.4.3. MISRA

"La calidad es nuestra mejor garantía de la fidelidad de los clientes, nuestra más fuerte defensa contra la competencia extranjera y el único camino para el crecimiento y los beneficios."
Jack Welch

3.4.3.1. Propósito.

En 1998, la Asociación de Confiabilidad de Software de la Industria del Motor del Reino Unido (*UK's Motor Industry Software Reliability Association*) estableció un conjunto de 127 reglas para el uso de C en sistemas de seguridad crítica.

Los motivos de la popularidad de C en el ámbito de emdebidos son claros: el acceso fácil al hardware, los bajos requisitos de memoria y el rendimiento eficiente en tiempo de ejecución son los más importantes entre ellos. De igual forma conocidos son los problemas con C: control de tiempo de ejecución altamente limitado, una sintaxis que es propensa a errores que son técnicamente legales, y una gran cantidad de áreas donde el estándar ISO C establece explícitamente que el comportamiento es de implementación definida o no definida.

En manos de un programador verdaderamente experimentado, la mayoría de las limitaciones de C se pueden evitar. Por supuesto, eso deja el problema de los programadores inexpertos. El problema es similar al dilema que enfrentan los padres cuando sus hijos aprenden a conducir. La solución que la mayoría de los padres adoptan es darle a su hijo un vehículo grande y lento con excelentes frenos; Pocos entregan las llaves de un coche deportivo de alto rendimiento.

ISO C es como un vehículo de alto rendimiento, con frenos muy cuestionables. A pesar de esto, las organizaciones no tienen otra opción que dejar que los principiantes se pierdan en el lenguaje. Los resultados son previsibles y están bien documentados.

La Asociación de Confiabilidad de Software de la Industria del Motor (MISRA) del Reino Unido se dio cuenta de que, en muchas áreas del diseño de un automóvil, la seguridad es de suma importancia. También reconocieron que C era el lenguaje de facto para codificar tales sistemas y que estos sistemas son, por lo tanto, vulnerables a las limitaciones de C. En lugar de exigir el uso de un lenguaje más seguro como Ada, la organización buscó formas de hacer que C fuera más seguro.

El resultado fue un conjunto de "Reglas para el uso del lenguaje C en software basado en vehículos" o "MISRA C", ya que se conocen de manera más informal. Las reglas, que se publicaron por primera vez en 1998, comprenden un documento de 70 páginas que describe un subconjunto de C que evita muchos de sus problemas conocidos. (Desafortunadamente, se tiene que comprar el documento de MISRA, el cual está disponible en www.misra.org.uk por aproximadamente $ 50).

El documento MISRA C contiene ocho capítulos y dos apéndices y obviamente fue escrito por programadores de embebidos con experiencia. Los capítulos del 1 al 6 contienen información importante sobre los fundamentos de MISRA C y cómo interpretar las reglas. Estos capítulos deben leerse antes de sumergirse en las reglas actuales, que se encuentran en el Capítulo 7.

3.4.3.2. Reglas

MISRA C es un conjunto de pautas de desarrollo de software para el lenguaje de programación C desarrollado por MISRA (*Motor Industry Software Reliability Association*). Sus objetivos son facilitar la seguridad del código, la seguridad, la portabilidad y la confiabilidad en el contexto de los sistemas integrados, específicamente aquellos sistemas programados en ISO C / C90 / C99

3.4.3.3. Historia

- Proyecto: 1997
- Primera edición: 1998 (normas, requeridas / sugerencias).
- Segunda edición: 2004 (normas, requeridas / sugerencias).
- Tercera edición: 2012 (directivas; reglas, decididas/ sin decidir)
- Cumplimiento MISRA: 2016

Para las dos primeras ediciones de MISRA-C (1998 y 2004) todas las Pautas fueron consideradas como Reglas. Con la publicación de MISRA C: 2012, se introdujo una nueva categoría de Directrices: la Directiva cuyo cumplimiento está más abierto a la interpretación o se relaciona con asuntos de procesos o procedimientos.

3.4.3.4. Adopción

Aunque originalmente estaba dirigido específicamente a la industria automotriz, MISRA C ha evolucionado como un modelo ampliamente aceptado para las mejores prácticas por parte de desarrolladores líderes en sectores como automotriz, aeroespacial, telecomunicaciones, dispositivos médicos, defensa, ferrocarriles y otros. Por ejemplo:

- Los estándares de codificación C ++ del proyecto *Joint Strike Fighter* se basan en MISRA-C: 1998.
- Los estándares de codificación del Laboratorio de Propulsión a Chorro de la NASA C se basan en MISRA-C: 2004.
- La parte 6 de ISO 26262-1: 2011 Seguridad funcional - Vehículos de carretera cita MISRA C 2004 y MISRA AC AGC como un subconjunto apropiado del lenguaje C.
- La especificación general de software de AUTOSAR 4.2 (SRS_BSW_00007) requiere que, si la implementación del Módulo BSW está escrita en lenguaje C, entonces deberá cumplir con el estándar MISRA C 2004.

- La especificación general de software AUTOSAR 4.3 (SRS_BSW_00007 cambiado) requiere que, si la implementación del Módulo BSW está escrita en lenguaje C, deberá cumplir con el estándar MISRA C 2012.

3.4.3.5. Clasificación y categorización de las guías.

Cuando se inicia un nuevo proyecto de software, se debe utilizar el último estándar MISRA. Los estándares anteriores todavía están disponibles para su uso con proyectos de software heredados que necesitan referirse a ellos.

3.4.3.5.1. Clasificación.

Cada Guía se clasifica como: Obligatoria (nueva para MISRA C: 2012), requerida o sugerida. Además, el documento de cumplimiento de MISRA permite que las pautas de asesoramiento se desaparezcan

- Los lineamientos obligatorios siempre se cumplirán.
- Las pautas requeridas se cumplirán, a menos que estén sujetas a una Desviación.
- Los lineamientos sugeridos se consideran una buena práctica, pero el cumplimiento es menos formal.

3.4.3.5.2. Categorización

Las reglas se pueden dividir lógicamente en varias categorías:

- Evitar posibles diferencias de compilación, por ejemplo, el tamaño de un entero C puede variar, pero un INT16 siempre es de 16 bits. (C99 estandarizado en int16_t.)
- Evitar el uso de funciones y construcciones que son propensas a fallar, por ejemplo, “malloc” puede fallar.
- Producir código mantenible y depurable, por ejemplo, convenciones de nombres y comentarios.
- Reglas de mejores prácticas.
- Límites de complejidad.

3.4.3.6. Alcance

MISRA C: 2012 clasifica por separado cada directriz como Unidad de traducción única o Sistema.

3.4.3.7. Decidability

MISRA C: 2012 clasifica las reglas (pero no las directivas) como *Decidable* o *Undecidable*.

3.4.3.8. Logrando el cumplimiento

3.4.3.8.1. Cumplimiento de MISRA: 2016

En abril de 2016, MISRA publicó *MISRA Compliance*: 2016, una guía mejorada para lograr el cumplimiento de MISRA C y MISRA C ++.

3.4.3.8.2. Conformidad

Para que una parte del software declare que cumple con las Pautas MISRA C, todas las reglas obligatorias se cumplirán y todas las reglas y directivas requeridas se cumplirán o estarán sujetas a una desviación formal. Las reglas de asesoría pueden no aplicarse sin una desviación formal, pero esto aún debe registrarse en la documentación del proyecto.

Nota: Para fines de cumplimiento, no hay distinción entre las reglas y las directivas.

3.4.3.8.3. Desviaciones

Muchas reglas de MISRA C se pueden caracterizar como guía porque, bajo ciertas condiciones, los ingenieros de software pueden desviarse de las reglas y seguir siendo considerados compatibles con la norma. Las desviaciones deben documentarse en el código o en un archivo. Adicionalmente; se debe proporcionar una prueba de que el ingeniero de software ha considerado la seguridad del sistema y que desviarse de la regla no tendrá un impacto negativo; los requisitos para las desviaciones también incluyen:

- La regla se desvió de.
- Justificación de la desviación.

3.4.3.9. MISRA C Documentos publicados

3.4.3.9.1. MISRA C: 1998

La primera edición de MISRA C, "Pautas para el uso del lenguaje C en software basado en vehículos", que se publicó en 1998 y se conoce oficialmente como MISRA-C: 1998.

MISRA-C: 1998 tiene 127 reglas, de las cuales 93 son requeridas y 34 son consultivas; Las reglas están numeradas en secuencia del 1 al 127.

3.4.3.9.2. MISRA C: 2004

En 2004, se produjo una segunda edición de "Pautas para el uso del lenguaje C en sistemas críticos", o MISRA-C: 2004, con muchos cambios sustanciales en las pautas, incluida una renumeración completa de las reglas.

MISRA-C: 2004 contiene 142 reglas, de las cuales 122 son "requeridas" y 20 son "consultivas"; se dividen en 21 categorías temáticas, desde "Entorno" a "Fallos en tiempo de ejecución".

3.4.3.9.3. MISRA C: 2012

En 2013, se anunció MISRA C: 2012. MISRA C: 2012 extiende el soporte a la versión C99 del lenguaje C (al tiempo que mantiene las pautas para C90), además de incluir una serie de mejoras que pueden reducir el costo y la complejidad del cumplimiento, al tiempo que ayuda al uso consistente y seguro de C en aspectos críticos. sistemas.

MISRA-C: 2012 contiene 143 reglas y 16 "directivas" (es decir, reglas cuyo cumplimiento está más abierto a la interpretación o se relaciona con asuntos de procesos o procedimientos); Cada uno de los cuales está clasificado como obligatorio, requerido o asesor. Se clasifican por separado como Unidad de traducción única o Sistema. Además, las reglas se clasifican como *Decidable* o *Undecidable*.

3.4.3.9.4. MISRA C: 2012 Enmienda 1

En abril de 2016, MISRA publicó (como descargas gratuitas) la Enmienda 1 a MISRA C: 2012 que agregó catorce nuevas pautas de seguridad, junto con el Addendum 2 a MISRA C: 2012, que describe la cobertura de MISRA C: 2012 Contra ISO / IEC TS 17961: 2013 - C reglas de codificación segura.

3.4.3.10. Herramientas

Si bien existen muchas herramientas de software que pretenden verificar el código de "conformidad con MISRA", no existe un proceso de certificación MISRA.

La mayoría de las pautas se pueden verificar utilizando herramientas que realizan análisis de código estático. Las pautas restantes requieren el uso de análisis de código dinámico.

Las herramientas que verifican el código de conformidad con MISRA incluyen:

- ***Astrée*** *by AbsInt*
- ***Axivion Bauhaus Suite*** *by Axivion GmbH. MISRA C:2004, C:2012, C:2012 Amendment 1, C++:2008, Compliance:2016.*
- ***CodeSonar*** *by GrammaTech*
- ***Coverity by Synopsys*** *- Static Analysis*
- ***Cppcheck*** *- Open Source Static Analysis tool for C/C++*
- ***ECLAIR*** *by BUGSENG*
- ***Goanna*** *by Red Lizard Software – A software analysis tool for C/C++.*
- ***IBM Rational Logiscope*** *(discontinued since 2012, see Kalimetrix)*
- ***Rational Test RealTime*** *by IBM - A cross-platform solution for component testing, static and runtime analysis*
- ***Kalimetrix Logiscope*** *(previously known as IBM Rational Logiscope or Telelogic Logiscope)*
- ***Klocwork*** *by Rogue Wave Software*

- ***LDRA Testbed*** *by Liverpool Data Research Associates*
- ***mutato****r by Farzad Sadeghi.*
- ***Parasoft C/C++test*** *by Parasoft*
- ***PC-Lint*** *by Gimpel Software. MISRA C:1998, C:2004, C:2012, C++:2008.*
- ***Polyspace*** *by MathWorks*
- ***QA-C*** *by Programming Research*
- ***RESORT*** *for C and C++ by Soft4Soft*
- ***SonarQube*** *by SonarSource (Open Source with some commercial plug-in components)*
- ***SQuORE*** *by Squoring Technologies*
- ***Telelogic Logiscope*** *(discontinued since 2008, see Kalimetrix Logiscope)*
- ***Understand*** *by SciTools*

C/C++ compilers that support MISRA conformance include:

- ***Green Hills Software***
- ***IAR Systems*** *- MISRA C:1998, C:2004, C:2012, C++:2008.*
- ***TASKING*** *- MISRA C:1998, C:2004, C:2012.*
- ***TI Compilers****.*

3.4.4. Herramienta CppCheck para revisión de estándares

"La cualidad de un líder se refleja en
las exigencias que pone a sí mismo",
Ray Kroc

Cppcheck es una herramienta de análisis para el código C / C ++. A diferencia de los compiladores C / C ++ y muchas otras herramientas de análisis, no detecta errores de sintaxis. En su lugar, Cppcheck detecta los tipos de errores que los compiladores normalmente no detectan. El objetivo no es falsos positivos.

3.4.4.1. Página de descarga.

Página principal de CppCheck está localizado en:

http://www.abxsoft.com/misra/misra-index.html

Instalador está localizado en:

http://cppcheck.sourceforge.net/

3.4.4.2. Código y plataformas soportados.

Puede verificar el código no estándar que incluya varias extensiones de compilador, código en ensamblador, etc. Cppcheck debería funcionar en cualquier plataforma que tenga suficiente CPU y memoria. Se debe entender que hay límites de Cppcheck. Y que este, pocas veces vez se equivoca acerca de los errores reportados. Sin embargo, hay muchos errores que no detecta.

Se encontrarán más errores en su software al probar su software cuidadosamente, que al usar Cppcheck ó al instrumentar el software, que utilizando Cppcheck. Pero Cppcheck puede detectar algunos de los errores que se pierden al probar e instrumentar el software.

3.4.4.3. Severidades

Las posibles severidades para los mensajes son:

- **error**: se utiliza cuando se encuentran errores.
- **warning**: sugerencias sobre programación defensiva para prevenir errores.
- **style**: problemas de estilo relacionados con la limpieza del código (funciones no utilizadas, código redundante, constantes, etc.).
- **Performance**. Sugerencias para hacer el código más rápido. Estas sugerencias se basan únicamente en el conocimiento común. No es seguro que se obtendrá una diferencia mensurable en la velocidad al corregir estos mensajes.
- **Portabilidad**: advertencias de portabilidad. Portabilidad de 64 bits. El código podría funcionar diferente en diferentes compiladores. etc.
- **Información**: Problemas de configuración. La recomendación es habilitarlos solo durante la configuración.

3.4.4.4. Comprobación multiproceso

La opción -j se utiliza para especificar el número de subprocesos que desea utilizar. Por ejemplo, para usar 4 hilos para verificar los archivos en una carpeta:

```
cppcheck -j 4 path
```

Se debe tener en cuenta que esto deshabilitará la verificación de funciones no utilizadas.

3.4.4.5. Uso de la herramienta

1. Abra la aplicación
2. Seleccione del menpú principal Proyecto Nuevo

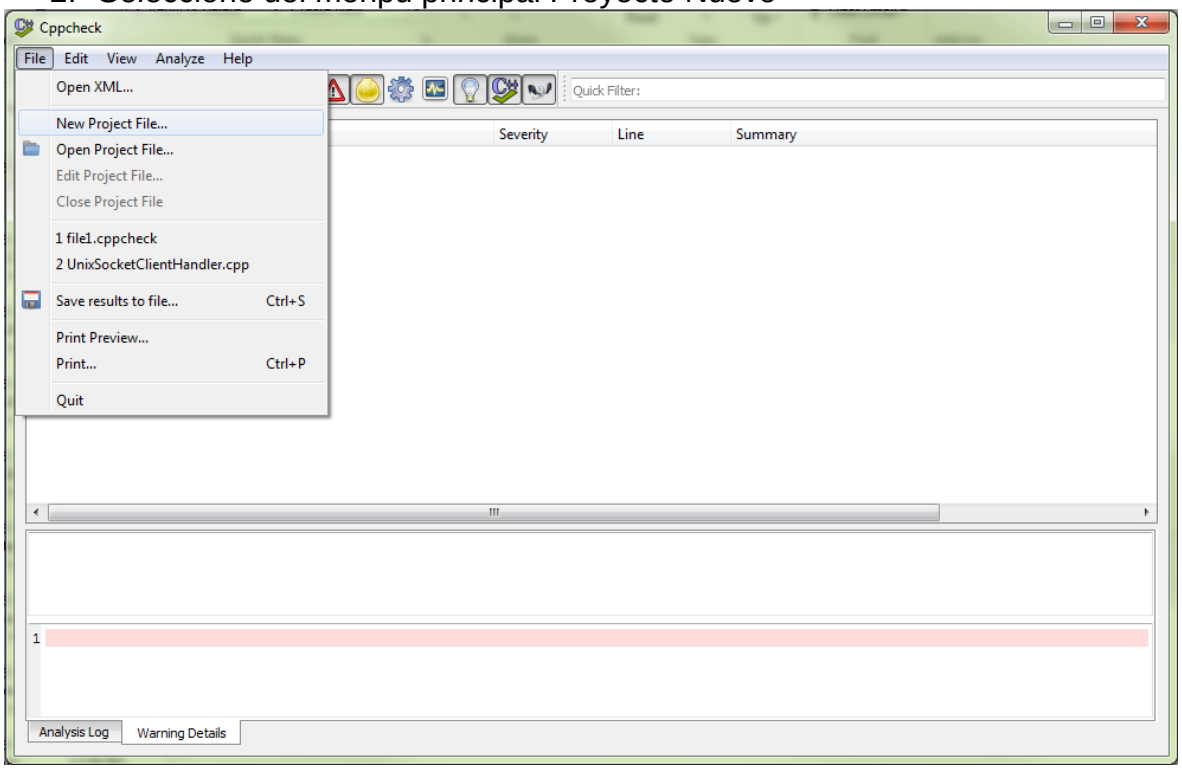


Figura 5 Herramienta cppcheck 1

3. Cree un carpeta para el proyecto.
4. Cree el proyecto en la carpeta seleccionada previamente.

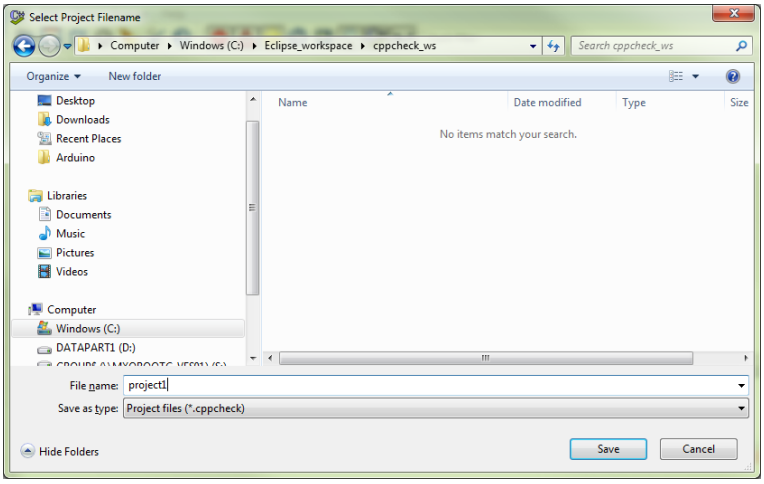


Figura 6 Herramienta cppcheck 2

5. Guarde el proyecto.

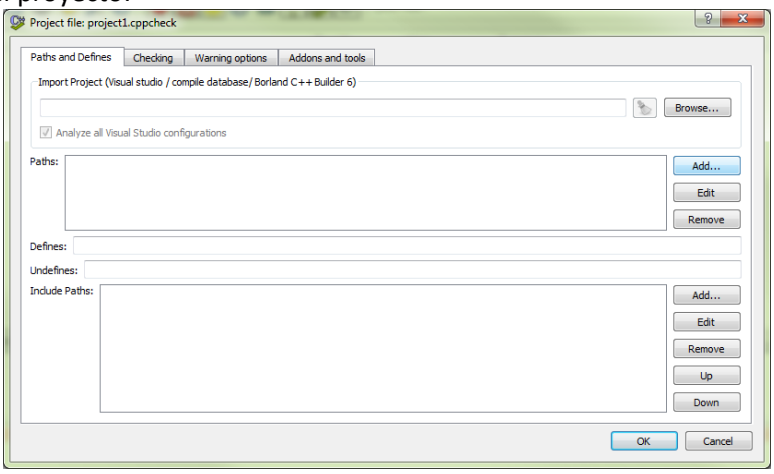


Figura 7 Herramienta cppcheck 3

6. Seleccione el path de la ruta del proyecto

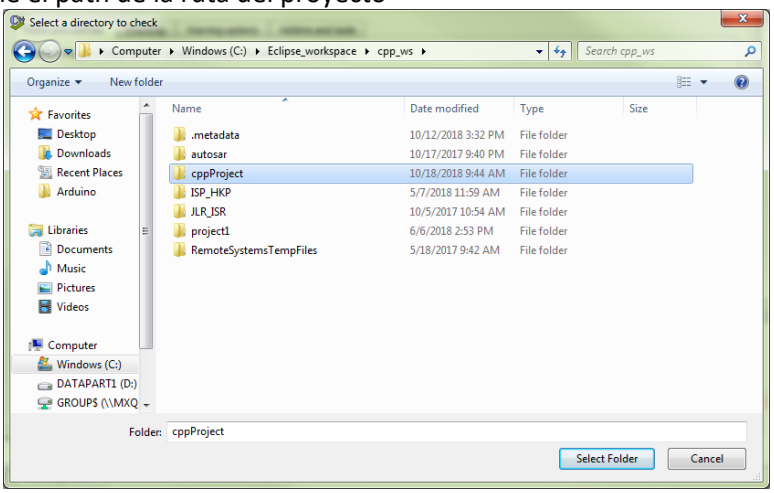


Figura 8 Herramienta cppcheck 4

7. Presione aceptar

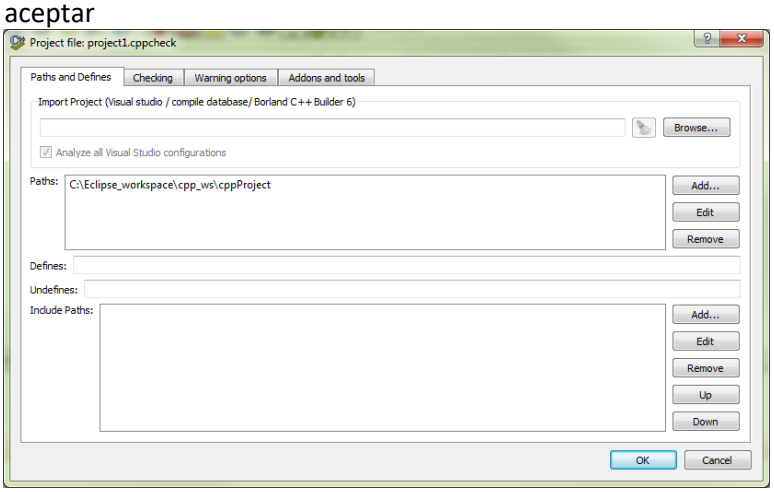


Figura 9 Herramienta cppcheck 5

8. Seleccione de la ventana siguiente la opción de Sí.

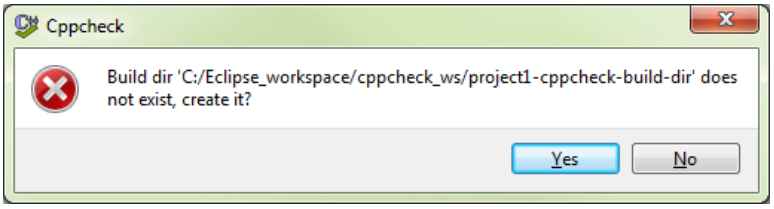


Figura 10 Herramienta cppcheck 6

9. El programa empezará a analizar el código de forma inmediata

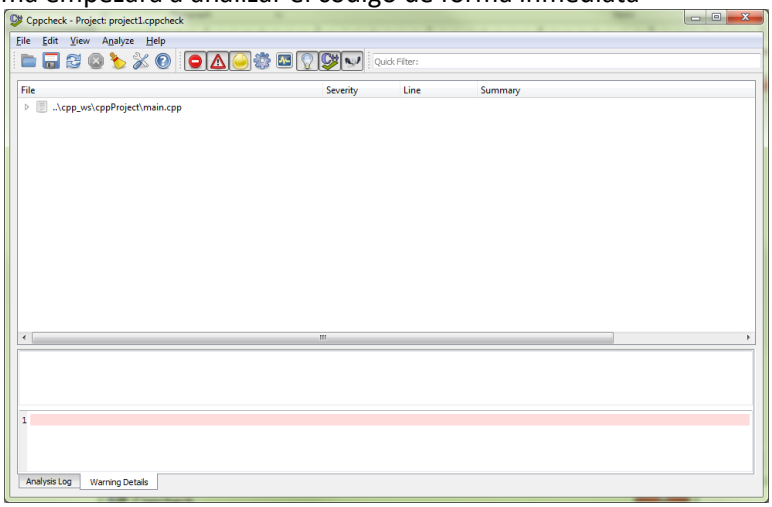


Figura 11 Herramienta cppcheck 7

10. Del siguiente menú de la barra de herramientas, seleccione las reglas que desea aplicar (en el proyecto serán las reglas C99 de C).

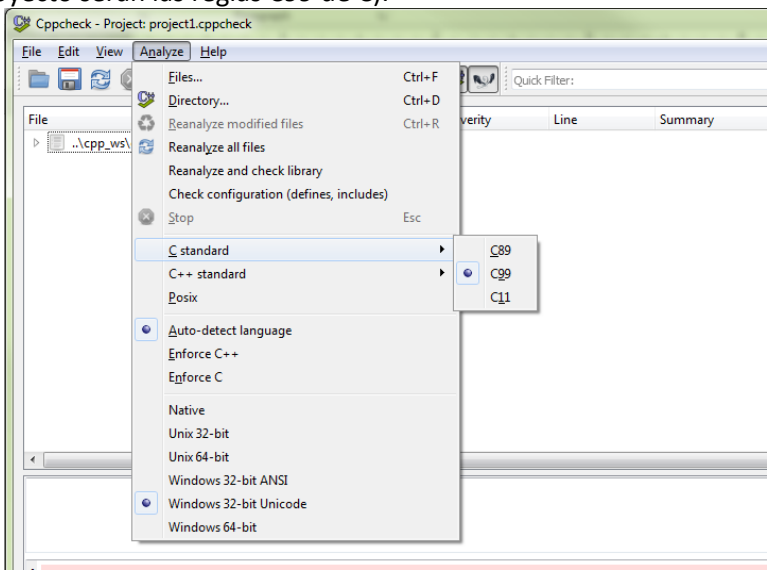


Figura 12 Herramienta cppcheck 8

11. En el caso de C++ usar la regla C++11

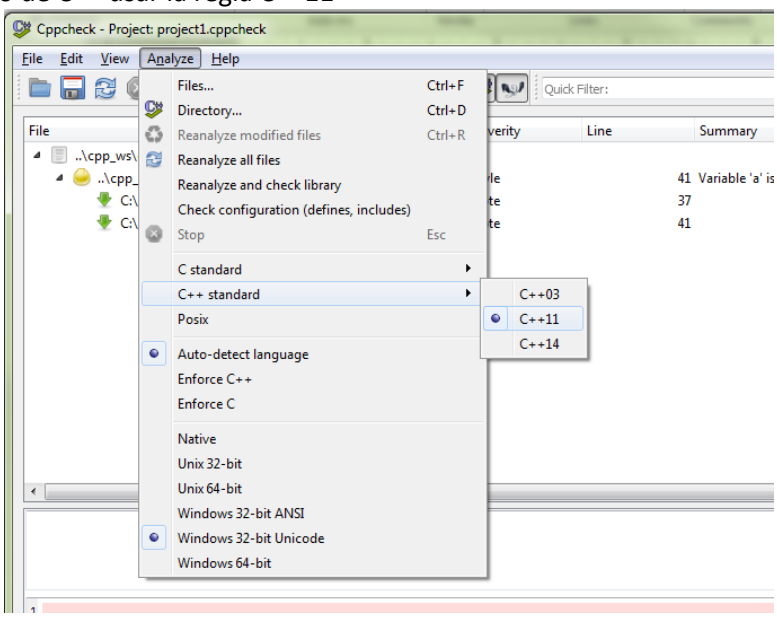


Figura 13 Herramienta cppcheck 9

12. Al cerrar debera refrescar la salida, al finalizar se mostrarán todas las reglas aplicables comose explica en la sección severidades.

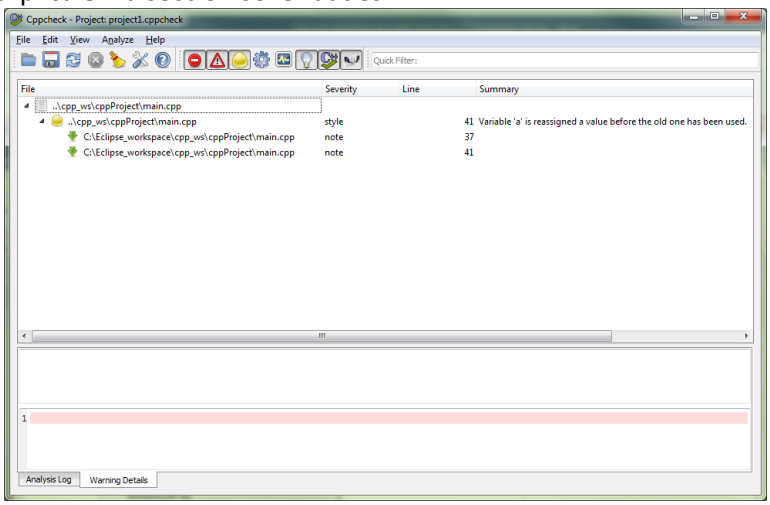


Figura 14 Herramienta cppcheck 10

3.4.4.6. Práctica individual

Realizar un programa en lenguaje C o C++ que:

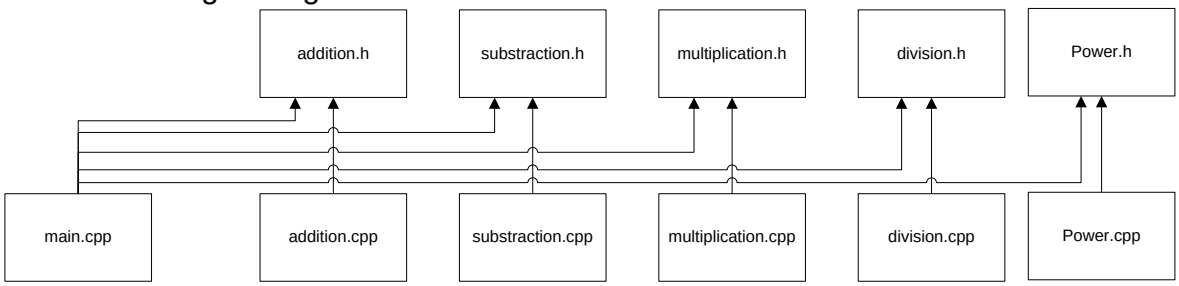
- Operaciones básicas: SUMAR, RESTAR, MULTIPLICAR Y DIVIDIR mediante consola.
- Deberá mostrar un menú como se muestra en al siguiente programa y las validaciones que se mostradas:

```
What kind of numbers do you want to use?
        [A] INTEGER
        [B] FLOAT
Please, depress any key to capture the value:
Option: a
The option selected is unknown, please retry the option:
Option: A
INTEGER is the option selected!
```

- Una vez que el tipo de valor sea seleccionado, se deberá mostrar la pantalla de operación a ejecutar. Como se muestra a continuación:

```
Program to calculate for integer numbers only:
        [A] ADDITION
        [B] SUBSTRACTION
        [C] MULTIPLICATION
        [D] DIVISION
        [D] POWER
Please, depress any key to capture the value:
Option: a
The option selected is unknown, please retry it:
Option: A
Addition is the option selected!
Insert the number:
Value: A
The value inserted is not valid, please, introduce valid data.
Value: 12.5
The value inserted is not valid, please, introduce valid data.
Value: 12
Insert the number:
Value: 3
The result for: 12 + 3 is 15
```

- Contenga la siguiente estructura:



- Las operaciones con valores flotantes soportan valores enteros.

ENTREGABLE

El código deberá ser entregado con:

1. El código deberá:
    a. Realizar lo que se describió anteriormente de forma TOTAL.
    b. Considerar las recomendaciones de la sección de Anexos de este documento.
2. Deberá ser revisado por medio de la aplicación CppCheck.
3. El reporte de la aplicación CppCheck deberá:
    a. Ser entregado en un documento de Word bajo la siguiente etiqueta: NOMBREAPELLIDOEMPRESA_CppCHeck.docx
    b. No contener “*errors*”, ni “*warnings*”.
    c. Ser subido a la plataforma Moodle.

3.4.5. Generación de documentación de código

"If your program isn't worth documenting,
it probably isn't worth running"
J. Nagler. 1995
Coding Style and Good Computing Practices

3.4.5.1. Introducción

Documentar el código de un programa es añadir suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores sepan qué hacer, sino que además los humanos entiendan qué están haciendo y por qué.

Porque entre lo que tiene que hacer un programa y cómo lo hace hay una distancia impresionante: todas las horas que el programador ha dedicado a pergeñar una solución y escribirla en el lenguaje que corresponda para que el ordenador la ejecute ciegamente.

Documentar un programa no es sólo un acto de buen hacer del programador por aquello de dejar la obra rematada. Es además una necesidad que sólo se aprecia en su debida magnitud cuando hay errores que reparar o hay que extender el programa con nuevas capacidades o adaptarlo a un nuevo escenario. Hay dos reglas que no se deben olvidar nunca:

1. todos los programas tienen errores y descubrirlos sólo es cuestión de tiempo y de que el programa tenga éxito y se utilice frecuentemente
2. todos los programas sufren modificaciones a lo largo de su vida, al menos todos aquellos que tienen éxito

Por una u otra razón, todo programa que tenga éxito será modificado en el futuro, bien por el programador original, bien por otro programador que le sustituya. Pensando en esta revisión de código es por lo que es importante que el programa se entienda: para poder repararlo y modificarlo.

¿Qué hay que documentar?

- Hay que añadir explicaciones a todo lo que no es evidente.
- No hay que repetir lo que se hace, sino explicar por qué se hace.

Y eso se traduce en:

- ¿de qué se encarga una clase o estructura? ¿un paquete?
- ¿qué hace un método o una función?
- ¿cuál es el uso esperado de un método o función?
- ¿para qué se usa un atributo o una variable?
- ¿cuál es el uso esperado de un atributo o variable?
- ¿qué algoritmo estamos usando? ¿de dónde lo hemos sacado?
- ¿qué limitaciones tiene el algoritmo? ¿... la implementación?
- ¿qué se debería mejorar ... si hubiera tiempo?

Tipos de comentarios
En Java y C disponemos de tres notaciones para introducir comentarios:

- **Comentarios de bloque**.
  Comienzan con los caracteres "/**", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/".

  Comienzan con los caracteres "/*", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/".

- **una línea**.
  Comienzan con los caracteres "/*" y terminan con la instrucción "*/".

Cada tipo de comentario se debe adaptar a un propósito:

- **Comentarios de bloque.**
  Para generar documentación externa.

  Para eliminar código. Ocurre a menudo que código obsoleto no queremos que desaparezca, sino mantenerlo "por si acaso". Para que no se ejecute, se comenta. (En inglés se suele denominar "comment out")

- **una línea**. Para documentar código que no necesitamos que aparezca en la documentación externa (que genere javadoc).

  Este tipo de comentarios se usará incluso cuando el comentario ocupe varias líneas, cada una de las cuales comenzará con "/* …*/"

¿Cuándo hay que poner un comentario?
Por obligación:

1. al principio de cada clase
2. al principio de cada método
3. ante cada variable de clase

Por conveniencia (una línea):

1. al principio de fragmento de código no evidente
2. a lo largo de los bucles

Y por si acaso (una línea):

1. siempre que hagamos algo raro
2. siempre que el código no sea evidente

Y una nota de cautela, cuando un programa se modifica, los comentarios deben modificarse al tiempo.

3.4.5.2. Doxygen

Doxygen es un generador de documentación para C++, C, Java, Objective-C, Python, IDL (versiones Corba y Microsoft), VHDL y en cierta medida para PHP, C# y D. Dado que es fácilmente adaptable, funciona en la mayoría de sistemas Unix así como en Windows y Mac OS X. La mayor parte del código de Doxygen está escrita por Dimitri van Heesch.

Doxygen es un acrónimo de dox (document) gen (generator), generador de documentación para código fuente.

Varios proyectos como KDE usan Doxygen para generar la documentación de su API. KDevelop incluye soporte para Doxygen.

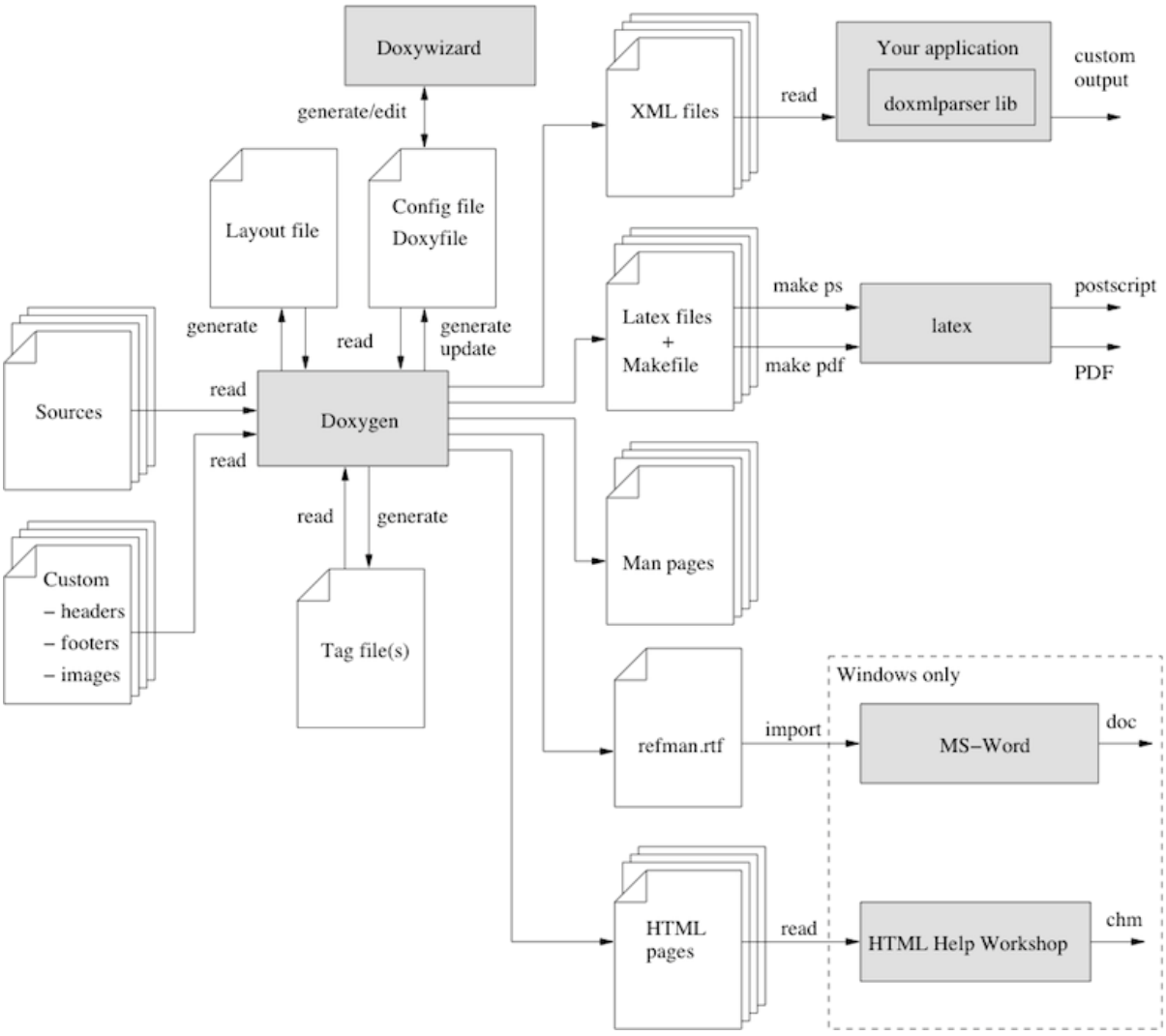


Figura 15 Doxygen

3.4.5.3. Descarga de Doxygen para documentar código

La siguiente liga corresponde al de descarga de la aplicación de Doxygen.

https://sourceforge.net/projects/doxygen/files/snapshots/

3.4.5.4. Documentación del código con Doxygen

3.4.5.4.1. Programación estructurada

Comentarios de bloque

Los comentarios decriben las estructuras de datos, algoritmos, etc. Deben estar en un comentario de bloque con el / en la columna 1, un * en la columna 2 antes de cada línea de texto, y cerrado */ en la columna 2-3 (Nota: que grep '^.*' atrapará todos los bloques de comentarios en el archivo.)

Por ejemplo:

```
/**
 *   write comment.
 *
 *
 */
```

Posición de los comentarios.

- Los bloques de comentarios dentro de una función.
  Debe ser tabeado a mismo tab en el que se encuentra el código que describe.
- Comentario fin de línea (End-of-line).
  Se empiezan aparte desde el enunciado usando tab. Si hay mas de un comentario, se deberán alinear todos al mismo tab establecido.

Cualquier archivo.

1. Comentario de encabezado.

```
/** @file main.cpp
 * @brief This file contain a code test for file.
 *
 * The main idea is to include most of the types defined for doxygen
 * to standard ANSI C document a file.
 *
 * @author Adbeel Alejandro Pérez (aaperez)
 * @bug No documented bugs.
 * @version 1.0.0.1 -- 2007-07-20
 * @version 1.0.0.2 -- 2007-08-02
 * - Fixes this
 * - Fixes that
 * - Fixes the other
 * @date  2018
 * @pre  Initalyzing the system.
 * @warning Improper use can crash your application
 * @copyright GNU Public License.
 */
```

Archivo de encabezado:

1. Includes

```
/**
 * @file stdio.h
 * @brief Standard library
 *
 * file used mostly by MS-DOS compilers to provide console input/output.[1]
 */
```

2. Declaraciones de funciones

```
/** @brief brief description here!
 *
 * Description here1!!!
 *
 * @param int var1
 * @param int var2
 * @return int return the value
 */
int funcion(int var1, int var2);
```

Source file:

1. Función principal

```
/** @brief main function.
 *
 * This is the entrypoint to any execution.
 *
 * @return Should return 0
 */
```

2. Includes

```
/* -- Includes -- */
/* libc includes. */
```

3. Structs

```
/** @struct foreignstruct
 * @brief This structure blah blah blah...
 * @var foreignstruct::a
 * Member 'a' contains...
 * @var foreignstruct::b
 * Member 'b' contains...
 */
struct MyStruct
{
 // ...
};

/**
 * \var MYSTRUCT
 * \brief You must call MyStruct::init() before using this variable.
 */
extern MyStruct MYSTRUCT;
```

4. Arrays

```
/*!
 * \brief Text literal containing the build number portion of the
 *    ESG Application Version.
 */
static const char build_version_text[] = "105";
```

5. Enums

```
/**
 * \enum compare_str_stat
 * \brief String Comparison Member Function status.
 *
 * \var no_match explanation 1
 * \var match explanation 2
 * \var partial_match explanation 3
 */
```

6. Defines

```
/*! A reference to an IID */
#ifdef __cplusplus
#define REFIID const IID &
#else
#define REFIID const IID *
#endif
```

7. Typedefs

```
/** @brief EMF+ manual 2.2.1.1, Microsoft name: EmfPlusBrush Object
*/
typedef struct {
 uint32_t Version; //!< EmfPlusGraphicsVersion object
 uint32_t Type;  //!< BrushType Enumeration
};
```

8. Variables globales

```
/*
 * state for kernel memory allocation.
 */
lmm_t malloc_lmm;

/*
 * Info about system gathered by the boot loader
 */
struct multiboot_info boot_info;
```

9. Variables locales

```
int var; /*!< Detailed description after the member */
```

10. Single line comments

```
/* for lprintf_kern() */
```

11. Block comments

```
/*
 * Tell the kernel memory allocator which memory it can't use.
 * It already knows not to touch kernel image.
 */
```

3.4.5.4.2. Programación orientada a objetos

Comentarios de clase

```
/**
 * @class ExampleClass
 *
 * @ingroup PackageName
 * (Note, this needs exactly one \defgroup somewhere)
 *
 * @brief Provide an example
 *
 * This class is meant as an example. It is not useful by itself
 * rather its usefulness is only a function of how much it helps
 * the reader. It is in a sense defined by the person who reads it
 * and otherwise does not exist in any real form.
 *
 * @note Attempts at zen rarely work.
 *
 * @author $Author: Adbeel Alejandro Pérez (aaperez) $
 * @bug No documented bugs.
 * @version 1.0.0.1 -- 2007-07-20
 * @version 1.0.0.2 -- 2007-08-02
 * - Fixes this
 * - Fixes that
 * - Fixes the other
 *
 * @date $Date: 2005/04/14 14:16:20 $
 *
 * Contact: bv@bnl.gov
 *
 * Created on: Wed Apr 13 18:39:37 2005
 *
 * $Id: doxygen-howto.html,v 1.5 2005/04/14 14:16:20 bv Exp $
 *
 */

#ifndef EXAMPLECLASS_H
#define EXAMPLECLASS_H

class ExampleClass
{

public:

 /// Create an ExampleClass
 ExampleClass();

 /// Create an ExampleClass with lot's of intial values
 ExampleClass(int a, float b);

 ~ExampleClass();

 /// This method does something
 void DoSomething();

 /** This is a method that does so
```

```
  * much that I must write an epic
  * novel just to describe how much
  * it truly does. */
 void DoNothing();

 /** \brief A useful method.
  * \param level an integer setting how useful to be
  * \return Output that is extra useful
  *
  * This method does unbelievably useful things.
  * And returns exceptionally useful results.
  * Use it everyday with good health.
  */
 void* VeryUsefulMethod(bool level);

private:

 const char* fQuestion; ///< the question
 int fAnswer;   ///< the answer

};        // end of class ExampleClass

#endif // EXAMPLECLASS_H
```

3.4.5.5. Uso de Doxygen.

1. Una vez instalado el programa de Doxygen.
2. Descargue el código a analizar en un repositorio local

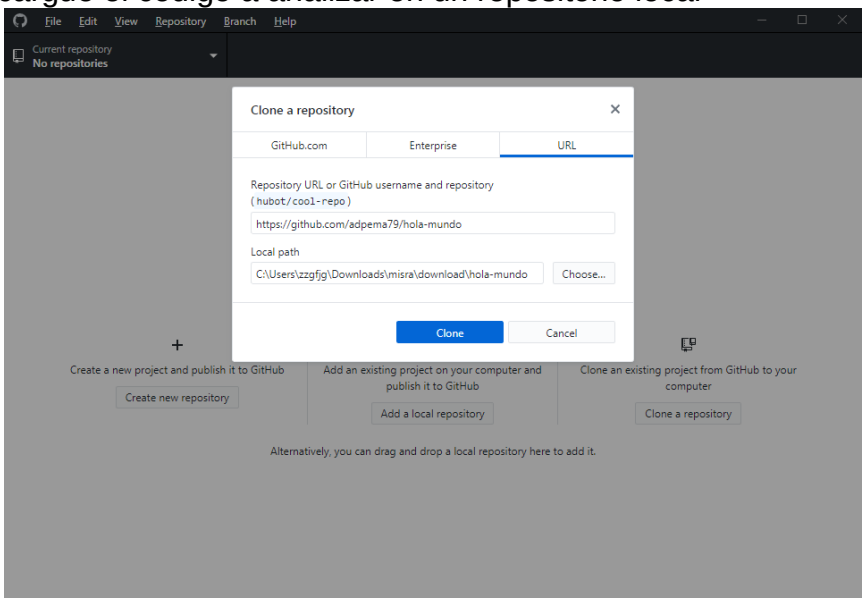


Figura 16 Herramienta Doxygen 1

3. Seleccione la aplicación de Doxywizard

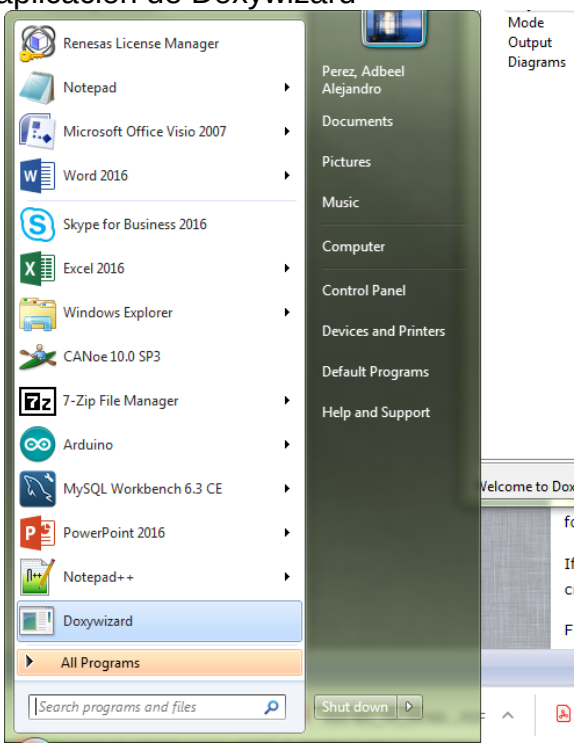


Figura 17 Herramienta Doxygen 2

4. Introduzca los datos de su Proyecto.

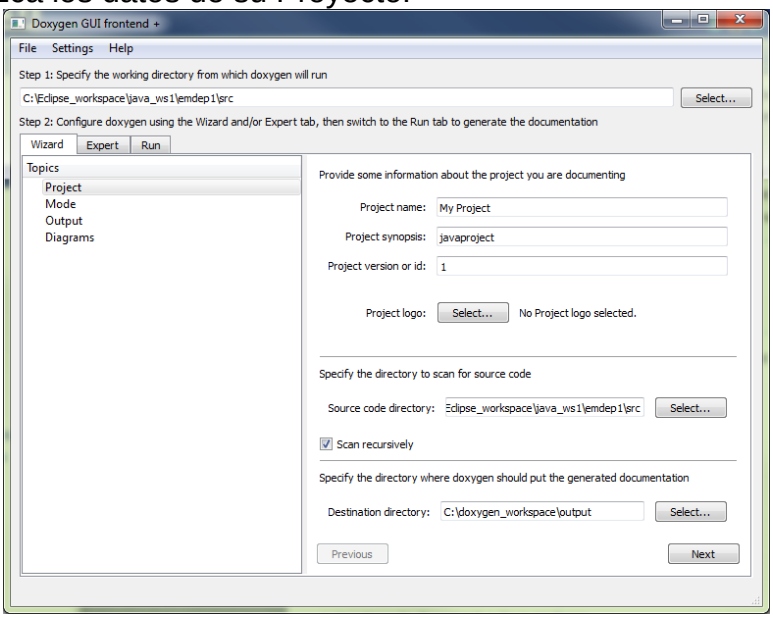


Figura 18 Herramienta Doxygen 3

5. Seleccione el lenguaje utilizado, ya sea C, Java ó C++.

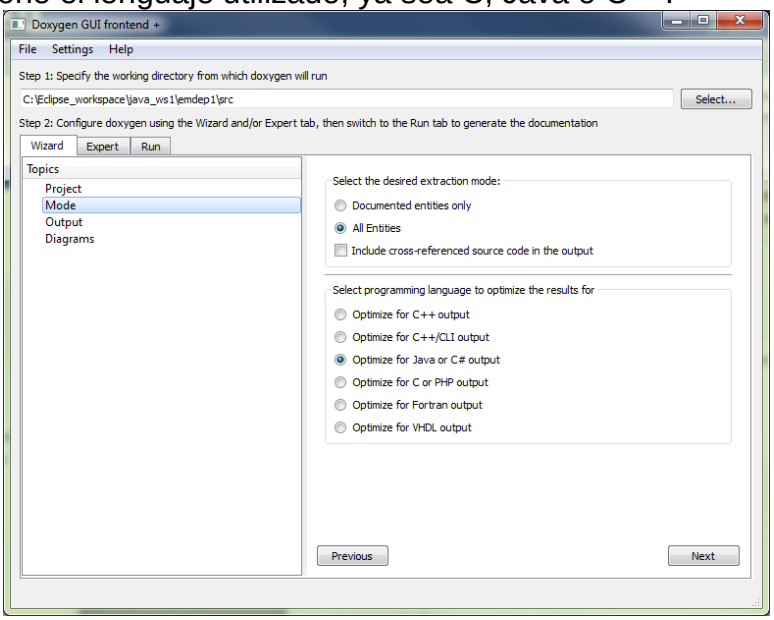


Figura 19 Herramienta Doxygen 4

6. Seleccione el modo de salida (esta opción deberá ser en html para fines del proyecto)

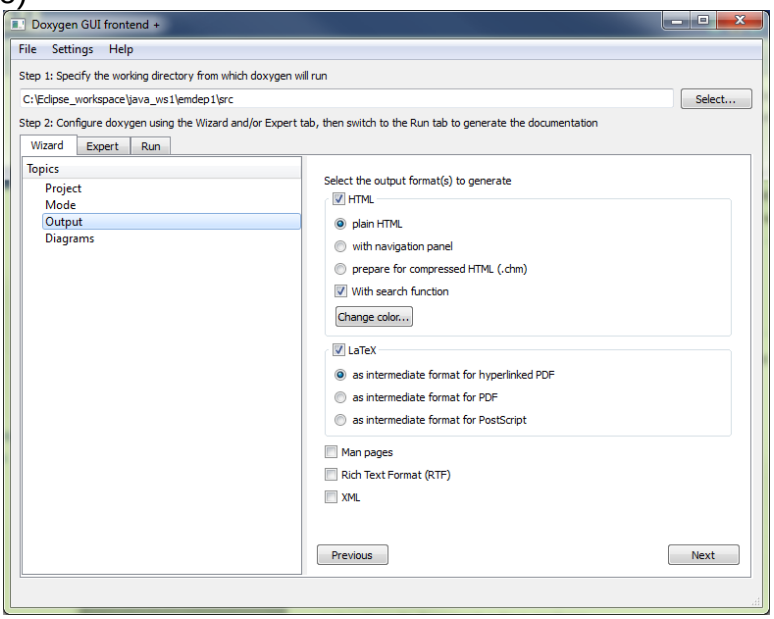


Figura 20 Herramienta Doxygen 5

7. Seleccione el modo de los diagramas como sigue:

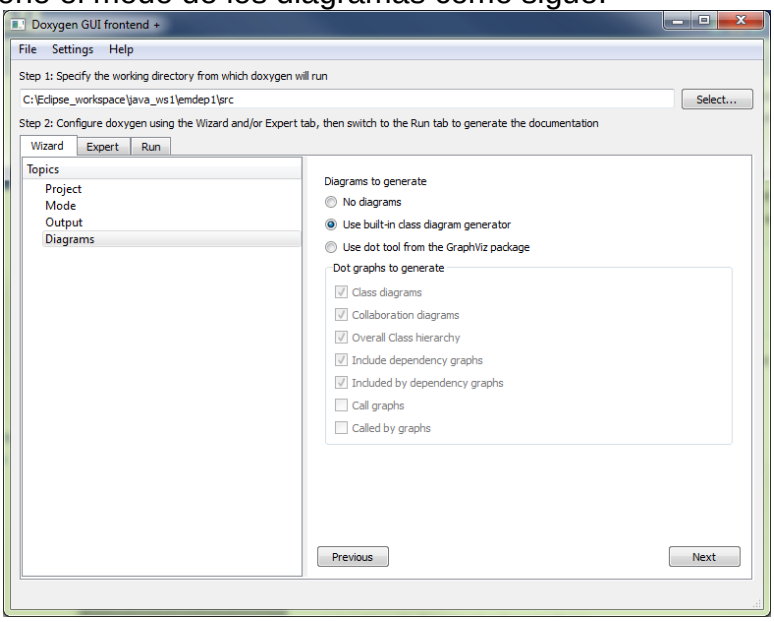


Figura 21 Herramienta Doxygen 6

3.4.5.6. Práctica individual

1. De la sección “3.4.4.6”, retome el ejercicio y documéntelo según las recomendaciones de Doxygen,
2. Deberá entregar la documentación del código en formato html en un archivo comprimido “zip” debidamente identificado NOMBREAPELLIDO_DOXYGEN.zip
3. Deberá ser subido a la plataforma Moodle.

III. ANEXO

1. Guía de documentación del proyecto.

1.1. Convenciones

1.1.1. Cuándo crear una convención para el proyecto

Cree la convención de codificación antes de pasar a la etapa de diseño del programa. Si bien una convención de codificación es un grupo de reglas a las que se hace referencia durante la codificación, algunas reglas, como la convención de nomenclatura aplicada a los nombres de funciones, están asociadas con el diseño del programa y, por lo tanto, deben decidirse antes de iniciar el diseño del programa.

1.1.2. Como crear una convención

Se recomienda que los proyectos que crean una nueva convención de codificación propia sigan el procedimiento descrito a continuación, paso a paso:

Paso 1. Decida la política para crear una convención de codificación.

Al crear una convención de codificación, lo primero que debe hacer es decidir sobre su política. Una política de creación define cómo se debe escribir el código para el proyecto, según, por ejemplo, las características del software desarrollado en el proyecto y los miembros del proyecto. Por ejemplo:

- ¿debería colocarse la prioridad en el código de seguridad y escritura que evite el uso de funciones que no son seguras, incluso si son convenientes de usar? ó,
- ¿debería escribirse el código de una manera que haga un uso cuidadoso de tales características inseguras pero convenientes?

Estas son algunas de las preguntas que deben abordarse en la política de creación.

Al decidir la política de creación, cada proyecto debe considerar qué características de calidad son particularmente importantes para el desarrollo de su software y examinar qué tipo de prácticas de codificación debe adoptar desde las siguientes perspectivas:

- Codificación que tenga en cuenta la seguridad ante fallos;
- Codificación que mejora la legibilidad del programa.
- Codificación que facilita la depuración, etc.

Paso-2. Elija las reglas basadas en la política de creación que se ha decidido.

El siguiente paso es elegir las reglas adecuadas del Cuadro de prácticas en la Parte 2, en función de la política de creación decidida en el paso-1. Si el proyecto decide sobre la política que prioriza la portabilidad, por ejemplo, se deben hacer esfuerzos para incluir muchas reglas que aborden los problemas de portabilidad en su convención de codificación.

En la “Parte 2 - Prácticas de codificación para software integrado” de esta guía, algunas reglas están marcadas con “○” o “●” como una guía para

facilitar el proceso de selección. Una regla marcada con "○" indica que se considera tan importante por la característica de calidad particular que aborda, si esta regla no se adopta como parte de la convención de codificación, ese aspecto de la calidad puede verse seriamente afectado. Mientras, "●" indica que es una regla que ya es tan conocida entre aquellos que tienen mucho conocimiento sobre las especificaciones del lenguaje C que puede no estar necesariamente incluida en la convención de codificación.

La forma más sencilla de crear una convención de codificación sería, por lo tanto, elegir solo las reglas indicadas con "○", lo que daría como resultado un conjunto de reglas ampliamente aplicadas.

Paso 3. Defina las partes de las reglas dependientes del proyecto.

Las reglas se tratan como uno de los siguientes tres tipos:

1) Reglas que se pueden usar como parte de la convención de codificación sin hacer ningún cambio (en el campo "Especificación de la regla", estas reglas no están marcadas).
2) Reglas que deben elegirse entre varias alternativas, de acuerdo con las características del proyecto (en el campo "Especificación de la regla", estas reglas están marcadas como "Elegir").
3) Reglas que deben prescribirse más específicamente en un documento (en el campo "Especificación de la regla", estas reglas están marcadas como "Definir" o "Documento".)

Las reglas tratadas como tipo 2) o tipo 3) no pueden incluirse en la convención de codificación tal como son. Para que las reglas tratadas como tipo 2) se deben adoptar como parte de la convención de codificación recién creada, primero se deben elegir entre las múltiples alternativas presentadas.

Para adoptar las reglas tratadas como tipo 3) como parte de la convención de codificación, deben estar mejor ajustadas para que puedan abordar las necesidades específicas de cada proyecto. Al hacerlo, la explicación complementaria que se proporciona a cada práctica descrita en esta guía debería servir como una referencia útil sobre la definición de reglas.

Paso 4. Determine el procedimiento para establecer excepciones a las reglas si es necesario.

Las características de calidad que deben enfocarse en el momento de la codificación pueden diferir, dependiendo de la característica que el proyecto pretende realizar a través de la implementación. (Por ejemplo, "En este proyecto, la eficiencia debe priorizarse sobre la capacidad de mantenimiento ...").

Puede haber casos en los que escribir un código que sea totalmente compatible con una determinada regla incluida en la convención de

codificación cause dificultades para lograr el objetivo específico del proyecto. Para lidiar con estos casos, es necesario tener un procedimiento para permitir excepciones parciales a esta regla W Los puntos importantes que se cubrirán en este procedimiento son los siguientes:

- Describa qué problemas pueden ocurrir escribiendo un código que cumpla con la regla;
- Haga que los expertos revisen los problemas y las posibles soluciones;
- Registrar el resultado de la revisión.

Asegúrese de no permitir excepciones con demasiada facilidad. La sustancia de la regla se perderá cuando haya demasiadas excepciones.

El siguiente es un ejemplo del procedimiento para permitir excepciones.

Paso adicional, sólo en caso necesario.
Después de seguir estos pasos en orden, agregue cualquier otra regla según sea necesario.

1.1.3. Reglas sobre convenciones de nombres.

La legibilidad de los archivos del proyecto se ve muy afectada por los nombres o etiquetas. Hay varios métodos para nombrar, pero los puntos importantes deben ser consistentes y hacer que los nombres sean fáciles de entender.

Para nombrar, se definirán los siguientes ítems:

- Pautas para los nombres en general.
- Cómo nombrar archivos (incluyendo carpetas y directorios)
- Cómo nombrar globalmente y localmente
- Cómo nombrar macros, etc.

A continuación, se presentan algunas pautas y reglas de nomenclatura introducidas en las convenciones de codificación existentes y en la literatura relacionada. Son útiles como referencia cuando se crea una nueva convención de nomenclatura específica del proyecto. Si no se define explícitamente ninguna convención de nomenclatura en el proyecto existente, la recomendación
sería crear una convención de nomenclatura más cercana al código fuente actual.

General

- Todas las etiquetas deben ser en inglés.
- Los nombres con guiones bajos iniciales y finales están reservados para propósitos del sistema y no deben usarse.
- Es mejor evitar los nombres que difieren solo en una letra: como foo y Foo.
- Usa guiones bajos para separar palabras en un nombre.
- Mantenga las minúsculas;
- Use nombres descriptivos para globales, nombres cortos para locales.

- Dar cosas relacionadas y nombres relacionados que muestren su relación y resaltar su diferencia.
- Los identificadores que nos e diferencian por:
    - The presence/absence of the underscore character
    - The interchange of the letter ‘O’, with the digit ‘0’ or the letter ‘D’
    - The interchange of the letter ‘I’, with the letter ‘l’ (el) or the digit ‘1’
    - The interchange of the letter ‘S’ with the digit ‘5’
    - The interchange of the letter ‘Z’ with the digit ‘2’
    - The interchange of the letter ‘n’ with the letter ‘h’.
- Cómo separar un nombre: un nombre que consta de varias palabras debe estar separado o delimitado usando una letra mayúscula para la primera letra de cada palabra. Determine qué estilo adoptar.
- Notación húngara: Hay una notación llamada notación húngara que indica explícitamente el tipo de variable.

Constantes

- #define constantes debe estar en todas las mayúsculas.
- Las constantes de enumeración deben tener el prefijo ENUM_ y todos los caracteres en mayúscula.

Estructuras arreglos o tipos especiales

- Reserve mayúsculas para macros y constantes de enumeración, y para los prefijos de nombres que siguen una convención uniforme.

Variables locales

- En variables, no elija nombres cortos: en su lugar, busque nombres que proporcionen información útil sobre el significado de la variable o función.

Variables globales

- Nombres globales deben tener un prefijo común que identifique el módulo al que pertenecen sys_xxxx, cont_xxxx.
- En variables, no elija nombres cortos: en su lugar, busque nombres que proporcionen información útil sobre el significado de la variable o función.

Funciones

- Los nombres de funciones deben basarse en verbos activos, tal vez seguidos de sustantivos.

Archivos

- Un nombre de archivo debe ser en inglés y tener ocho caracteres o menos (excluyendo la extensión), comenzando con una carácter o letra alfabética y seguido de caracteres alfanuméricos.
- Deben evitarse los nombres de archivo que sean iguales a los nombres de archivo de encabezado de biblioteca.
- Cómo nombrar archivos: asigne un nombre con un prefijo, por ejemplo, que exprese el subsistema.

1.2. Estructura del archivo

Los elementos que se usan comúnmente en un programa se describen en los archivos de encabezado para evitar el riesgo de errores de modificación cuando se encuentran dispersos en diferentes lugares.

Los archivos de encabezado deben contener definiciones de macro, declaraciones de etiquetas para estructuras, uniones y tipos de enumeración, declaraciones typedef, declaraciones de variables externas y declaraciones de prototipos de funciones que se usan comúnmente en varios archivos de origen.

1.2.1. Archivos de encabezado.

La forma en que debe ser estructurado el archivo de encabezado es el siguiente:

1. File header comment
2. Inclusion of system headers
3. Inclusion of user defined headers
4. #define macros
5. #define function macros
6. typedef definitions (type definitions for basic types such as int or char)
7. enum tag definitions (together with typedef)
8. struct/union tag definitions (together with typedef)
9. extern variable declarations
10. Function prototype declarations
11. Inline function

1.2.2. Archivos fuente

En el archivo fuente, deben describirse las definiciones de variables y funciones, las definiciones o declaraciones de macros, etiquetas y tipos (tipos typedef) que se usan solo en el archivo de origen individual. En la siguiente enumeración se establece la forma en que el proyecto debe ser ordenado:

1. File header comment
2. Inclusion of system headers
3. Inclusion of user-defined headers
4. #define macros used only in this file
5. #define function macros used only in this file
6. typedef definitions used only in this file
7. enum tag definitions used only in this file
8. struct/union tag definitions used only in this file
9. static variable declarations shared in this file
10. static function declarations
11. Variable definitions
12. Function definitions

* Con respecto a (2) y (3), tenga cuidado de no incluir elementos innecesarios.
* Evite describir (4) a (8) tanto como sea posible.

1.3. Validación de los estándares de codificación

Deberá usar la herramienta CppCheck como método de análisis de los estándares bajo el apartado C99 para C ó C03 para C++.

Durante la evaluación del código por la herramienta CppCheck:

- No deberá mostrar **errores** en ninguno de los archivos, en su defecto este tipo de advertencias, deberán ser autorizado por el asesor del proyecto.
- No deberá mostrar **warnings**, y en su defecto el equipo deberá justificar las desviaciones.
- No deberá mostrar problemas de **performance**, en caso de que estas se presenten, deberán atenderse.
- Se permite mostrar notas de **información**, **estilo y portabilidad** ya que esto no afecta el buen funcionamiento el código.

1.4. Documentación de código

Se deberá utilizar la herramienta Doxygen como método de documentación del código en formato HTML, el cual deberá estar controlado y actualizado conforme se realicen actualizaciones.

La documentación de código generada **NO deberá ser manejado como parte del documento de diseño y deberá ser presentado como resultado de la implementación y no como justificación del diseño**.

Se deberán utilizar todos los atributos expuestos en el presente documento y más…, pero no menos.

IV. REFERENCIAS

- **Abxsoft** , *Misra-C, Abraxas CodeCheck Solution*, abxsoft.com, http://www.abxsoft.com/misra/misra-index.html, publicado: NA, consultado: octubre 2018
- **Arciniega**, Fernando, *Normas y Estándares de calidad para el desarrollo de Software*, fernandoarciniega.com, https://fernandoarciniega.com/normas-y-estandares-de-calidad-para-el-desarrollo-de-software/, consultado 2018
- **Calidad ISO 9001**, *¿Qué es calidad?*, iso9001calidad.com, http://iso9001calidad.com/que-es-calidad-13.html, publicado: 2013, consultado: octubre 2018
- **clc-wiki**, *the C Standard*, clc-wiki, https://clc-wiki.net/wiki/The_C_Standard#Obtaining_the_Standard, publicado: 14 November 2013, consultado: octubre 2018.
- **clc-wiki**, *C18*, clc-wiki, https://en.wikipedia.org/wiki/C18_(C_standard_revision), publicado: 14 November 2013, consultado: octubre 2018.
- **clc-wiki**, *C11*, clc-wiki, https://en.wikipedia.org/wiki/C11_(C_standard_revision) , publicado: 14 November 2013, consultado: octubre 2018.
- **clc-wiki**, *C99*, clc-wiki, https://en.wikipedia.org/wiki/C99, publicado: 14 Noviembre 2013, consultado: octubre 2018.
- **Computer Science Department**, *15-410 Coding Style and Doxygen Documentation*, Carnegie Mellon, Computer Science Department, https://www.cs.cmu.edu/~410/doc/doxygen.html, publicado: 31 de agosto 2018, consultado: octubre 2018
- **Cppcheck**, *Cppcheck is a static analysis tool*, Cppcheck, http://cppcheck.sourceforge.net/, publicado:14 octubre 2018, consultado: octubre 2018
- **Doxygen**, *Documenting the code*, Doxygen, http://www.doxygen.org/ publicado: julio 2018, consultado: octubre 2018
- **Doxygen**, *Documenting the code*, Doxygen, http://www.doxygen.org/manual/starting.html, publicado: julio 2018, consultado: octubre 2018
- **Doxygen**, *Documenting the code*, Doxygen, https://www.stack.nl/~dimitri/doxygen/manual/docblocks.html, publicado: julio 2018, consultado: octubre 2018
- **Doxygen**, *Documenting the code*, Doxygen http://www.stack.nl/~dimitri/doxygen/download.html#srcbin, publicado: julio 2018, consultado: octubre 2018
- **Eduardo89**, *Estándares de calidad aplicadas al software*, slideShare, https://es.slideshare.net/eduardo89/estndares-de-calidad-aplicadas-al-software, publicado: 1 de diciembre 2009, consultado: octubre 2018
- **factorelectrico**, *¿QUE ES LA IEC?*, factorelectrico.blogspot.com, http://factorelectrico.blogspot.com/2014/01/que-es-la-iec.html, publicado: 30 de Enero 2014, consultado: octubre 2018

- **Fileformat**, *utf-8*, Fileformat, https://www.fileformat.info/info/charset/UTF-8/list.htm, publicado: NA, consultado: octubre 2018
- **Fileformat**, *utf-8*, Fileformat, https://www.fileformat.info/info/charset/UTF-16/list.htm, publicado:NA, consultado: octubre 2018
- **Fileformat**, *utf-8*, Fileformat, https://www.fileformat.info/info/charset/UTF-32/list.htm, publicado:NA, consultado: octubre 2018
- **GeekforGeeks**, Ide de compilación en C (BETA), GeekforGeeks, https://ide.geeksforgeeks.org/index.php, publicado: NA, consultado: octubre 2018.|
- **International Organization for Standardization**, *ISO/IEC 9899:2018*, International Organization for Standardization, https://www.iso.org/standard/74528.html, publicado: 2018, consultado: NA, octubre 2018.
- **Jenkins**, *Jenkins*, Jenkins, https://jenkins.io/download/thank-you-downloading-windows-installer/, publicado: NA, consultado: octubre 2018
- **Karron10**, *Normas y estándares en proyectos de TI*, Normas y estándares, wordpress.com, https://karron10.wordpress.com/2013/04/14/normas-y-estandares-en-proyectos-de-ti-2/, publicado: 14 de abril 2013, consultado: octubre 2018
- **Mañas**, José A., *Documentación de código*, Universidad Politécnica de Madrid, https://web.dit.upm.es/~pepe/doc/adsw/base/doc/doc.htm, publicado: 8 de Mayo 2003, consultado: octubre 2018
- **Nigel** Jones, *Introduction to MISRA C*, Embedded, Cracking the code to system development, https://www.embedded.com/electronics-blogs/beginner-s-corner/4023981/Introduction-to-MISRA-C, publicado: julio 1, 2002, consultado: octubre 2018
- **Stackoverflow**, *Use of #pragma in C*, Stackoverflow, https://stackoverflow.com/questions/232785/use-of-pragma-in-c, publicado: octubre 2008, consultado: octubre 2018.
- **SourceForge**, program2 download, SourceForge, https://sourceforge.net/projects/doxygen/files/snapshots/, publicado: 2018, consultado: octubre 2018
- **Software Reliability Enhancement Center IPA/SEC**, Technology Headquarters, *Embedded System development Coding Reference guide*, Information-technology Promotion Agency, Japan, https://www.ipa.go.jp/files/000040508.pdf, publicado: julio 2014, consultado: octubre 2018
- **Synopsys**, *Coverity*, Synopsys, Inc., https://scan.coverity.com/projects/new, publicado: NA, consultado: octubre 2018
- **Pardo**, Cesar Mauricio, Estándares Y Modelos De Calidad Del Software, http://evaluaciondesoftware2013.blogspot.mx/, publicado: 17 de marzo 2013, consultado: octubre 2018
- **Pérez**, Julián, Definición.de, https://definicion.de/estandar/, publicado: 2017, consultado: octubre 2018
- **Sánchez**, Luis E., **Fernández-Medina**, Eduardo, *Modelo de calidad para la seguridad*, University of Castilla-La Mancha, www.researchgate.net,

https://www.researchgate.net/publication/232252368, publicado: septiembre 2010, consultado: octubre 2018
- **Wikipedia**, *ANSI-C*, Wikipedia, https://en.wikipedia.org/wiki/ANSI_C, publicado: 16 de octubre 2018, consultado: octubre 2018
- **Wikipedia**, *Doxygen*, Wikipedia, https://es.wikipedia.org/wiki/Doxygen, publicado: 30 de Diciembre 2017, consultado: octubre 2018
- **Wikipedia**, *utf-8*, Wikipedia, https://en.wikipedia.org/wiki/UTF-8, publicado: 17 de octubre 2018, consultado: octubre 2018
- **Wikipedia**, *utf-16*, Wikipedia, https://en.wikipedia.org/wiki/UTF-16, publicado:21 de octubre 2018, consultado: octubre 2018
- **Wikipedia**, *utf-32*, Wikipedia, https://en.wikipedia.org/wiki/UTF-32, publicado:, consultado: octubre 2018
- **Wiki**, embedded, Wiki, https://en.wikichip.org/wiki/c/embedded, publicado: 4 de Enero 2015, consultado: octubre 2018

V. CONTROL DE CAMBIOS

Rev.	Descripción	Autor	Fecha
1b	Se ampliaron temas y modificaron generales, se agregaron secciones en anexos	Adbeel Alejandro Pérez	20181026
1	Inicial	Adbeel Alejandro Pérez	20181024