Recomendaciones de documentación para C++

- ✓ Para comentarios internos, que sólo deban ser leídos por los desarrolladores, utilice comentarios mediante:
  - // para una sola línea
  - /* para varias
  - líneas */
- ✓ Para comentarios externos, que sólo deban ser leídos por una herramienta de documentación, utilice comentarios mediante:
  - /// para una sola línea
  - ///< usado en ocasiones, para una sola línea
  - /** para varias
  - líneas */
- ✓ La documentación debe aparecer donde se declara un componente por primera vez. En general, esto significa que los bloques de documentación aparecerán en los archivos de encabezado (.h) en lugar de los archivos fuente (.cpp)
- ✓ La documentación debe aparecer antes de la declaración que describe, y con la misma sangría.
- ✓ Pueden agregarse etiquetas al estilo Javadoc, dentro de los comentarios detallados:
  - @see para documentar referencias (a clase, función, método o enlace)
  - @param para documentar un parametro de una función o método
  - @returns para documentar el valor de retorno de un metodo
  - @tparam para documentar un parametro de una plantilla
  - @author para documentar el nombre del desarrollador
  - @version para documentar la versión del código
  - @throws para documentar una de las excepciones producidas
  - @note o @warning para documentar información conceptual adicional
  - No debería usarse: @file, @return, @throw, @exception, @remark
  - Un ejemplo de los 2 ultimos criterios sería:

```
/**
 * Suma los números almacenados en un vector.
 *
 * @param valores Es el contenedor de los valores a ser sumados.
 * @return la suma de los `valores`, o 0.0 si `valores` está vacío.
 */
double sumar(std::vector<double> & const valores) {
    ...
}
```

Recomendaciones de codificación para C++

- ✓ Evite incluir código al estilo C en programas C++. Trate de adaptar a C++ cuando quiera reutilizar código en C.
- ✓ Cree una clase específica para cualquier comportamiento asociado a una estructura de datos, no cree una estructura para los datos y funciones separadas para operar en ella.
- ✓ Las estructuras de datos son apropiadas sólo para casos que necesitan una estructura de datos muy ligera y sin comportamiento.
- ✓ Evite jerarquías de herencia demasiado complejas, más de 3 niveles debería ser una señal de advertencia (excepto en Frameworks).
- ✓ Use la herencia para especializar el comportamiento de los mismos datos o similares, use plantillas para especializar los datos con el mismo comportamiento.
- ✓ Evite la herencia múltiple y utilícela sólo cuando se trate de consideraciones completamente distintas o inconexas (como el rol de un contenedor en la aplicación frente al tipo de persistencia de un contenedor).
- ✓ De ser necesario, aproveche la lista de parámetros en el diseño de funciones. Puede sobrecargar las funciones miembro, pero intente hacerlo sólo cuando sea necesario (funciones virtuales) o necesite variar la lista de parámetros.
- ✓ Mantenga las funciones cortas y con un único propósito.
- ✓ Ubique las referencias sa las librerías en el .cpp, dejando en el .h sólo las estrictamente necesarias.
- ✓ Declare cada variable de forma separada.
  - // Multiple definición de variables, desaconsejada
  - int contador, numeroFilas, numeroColumnas;
  - // Definición correcta de variables.
  - int contador;
  - int numeroFilas;
  - int numeroColumnas;
- ✓ Evite la declaración de constantes con #define e incorpore el uso de const
- ✓ Las variables y funciones globales deberían evitarse y, si se usan, siempre debe referirse a ellas mediante el operador ::
  - Por ejemplo, ::mainWindow.open(), ::applicationContext.getName(), ::hacerAlgo(1.0)
- ✓ Nombrado de variables:
  - La magnitud fundamental debe aparecer primero, con los modificadores concatenados después. Sea por ejemplo una cantidad de fecha/hora importante: fechaObservada es correcto, observadaFecha no; tiempoTotal es correcto, totalTiempo no.
  - Use nombres significativos que se ajusten a la magnitud a representar. Por ejemplo, si se está hablando de valores medios de color de una imagen obtenidos a partir del color del pixel: imagenColorPromedio es correcto, pixelColorPromedio no.
    - No incluya las unidades en el nombre de la magnitud. Por ejemplo, si se necesitara medir el volumen de un cuerpo en m^3 y en liltros: volumen es correcto, volumenM3 y volumenLitros son incorrectos.

- Evite los acrónimos. En caso que deba usarlo como nombre, escribalo siguiendo la regla general. Por ejemplo: htmlFuente es correcto, HTMLFuente no.
- Evite las abreviaturas. Por ejemplo, aperturaValvula es mejor que apValv
- Cuando deba asociar nombres de variables a campos de archivos o base de datos, verifique los nombre aprobados en elllos y use luego nombres iguales o similares para sus variables.
- Los nombres de los tipos definidos por usted deben iniciar en mayúsculas. Así como cada palabra o modificador que acompaña. Por ejemplo:
  - class Linea;
  - class CuentaPrincipal;
  - struct { ….. } Nodo;
  - Nodo miNodo;
  - typedef Vector<Piezas> PiezasVector;
- Use camelCase (por ejemplo: int anchoLinea) o snake_case (por ejemplo: int ancho_linea) de manera consistente en el conjunto de sus módulos. Si bien en C++ existen los 2 estilos y en Python se prefiere snake_camel, se espera que los programas nuevos sean consistentes.
- En caso que opte por contantes, use mayúsculas y guión bajo de separación, para definirlas. Por ejemplo: int const MAX_VOLUMEN = 23;
- Aunque siempre es mejor, si minimiza el uso de constantes, implementando su valor como un método. Para el caso anterior, sería : int getMaxVolumen( ) { return 23; }

✓ Nombrado de métodos:

- Los identificadores que representan métodos o funciones deberían describir naturalmente la acción del método (por lo general comenzar con un verbo afín) y deberían estar escritos en camelCase o snake_case, conforme al estilo adoptado. Los nombres de las funciones de los miembros privados deben, además, comenzar con un guión bajo.
- No coloque un espacio entre el nombre de la función y el paréntesis de apertura al declarar o invocar la función.
- Evite los acrónimos. Si debiera usarlo como nombre, escríbalo siguiendo la regla general. Por ejemplo: generarFuenteHtml() es correcto, generarFuenteHTML() no.

✓ https://docplayer.es/17931545-Documentacion-automatica-con-doxygen.html

✓ https://www.scenebeta.com/tutorial/documentando-el-codigo-con-doxygen

✓ https://www.youtube.com/watch?v=9JGCVseIq8w

✓ https://developer.lsst.io/cpp/api-docs.html

✓ https://developer.lsst.io/cpp/style.html

✓ https://www.youtube.com/watch?v=OLP2pWKLWLU

Espacio de Nombres

1. En tiempos remotos (antes de la estandarización y sin espacios de nombres):

   #include <iostream.h>

2. En tiempos más cercanos (después de la estandarización y aparición de espacios de nombres):

   #include <iostream>

   using namespace std;

3. En tiempos actuales, además de lo anterior se recomienda usar:

   #include <string>

   using std::string;

Clases INLINE vs. OFFLINE

1. Escribir una clase INLINE significa colocar la declaración y la definición de la clase en un mismo archivo .h, por ejemplo:

.h

```
class a
{
  private:
     int _valor;
  public:
    int getValor() {
      return _valor;
    }
    void setValor(int _valor)
    {
      _valor=valor;
    }
};
```

   El compilador reemplaza la definición de funciones inline en tiempo de compilación.

2. Escribir una clase OFFLINE, significa colocar la declaración o interfaz de la clase en el .h, y la definición o implementación en el .cpp, por ejemplo:

.h

```
class a
{
  private:
     int _valor;
  public:
    int getValor();
    void setValor(int);
};
```

.cpp

```
#include "prueba.h"

int prueba::getValor()
{
      return _valor;
 }

void prueba::setValor(int a)
{
      _a=a;
}
```

3. ¿Cuál se debe usar?

   OFFLINE, ya que cuando trabajamos inline el compilador inserta o incrusta una copia del cuerpo de cada método en cada lugar en el cual es llamado. Además, favorece el mantenimiento.

Guardas y librerías

1. Escribir SIEMPRE, primero las librerías estándar y luego los .h referentes a las clases nuevas creadas.
2. Por razones de eficiencia y orden se deben agregar guardas en cada archivo de cabecera.
3. Las guardas y librerías que coloquemos siempre deben llevar un orden especial

   Por ejemplo: