LexRis Logic Headers
Convenciones de Código

LexRis Logic

1 Presentación

Las Cabeceras de LexRis Logic han sido escritas de una forma que pueda ser legible por los usuarios que lo utilizaran o modificaran, estas convenciones están dirigidas al lenguaje C++ pero podría ser usado para otros lenguajes con características similares.

2 Reglas

2.1 Sobre la Cantidad de Caracteres

El límite de caracteres por linea son un total de 120 caracteres, si se supera los caracteres permitidos, en la siguiente línea aumentar una cantidad de indentación y seguir escribiendo hasta acabar la sentencia deseada.
Se puede usar más de una linea para terminar de escribir una sentencia, pudiendo variar la indentación, pero es necesario lograr que exista un orden en el código.

Ejemplo de Cantidad de Caracteres

Definir la siguiente función necesita de 161 caracteres pero esta se dividirá en dos lineas, una de 91 caracteres y otra de 70 caracteres acompañado de indentación, para mostrar un orden en el código y mejorar la lectura del mismo:

```
void example_function(int parameter_number_1,int parameter_number_2,int parameter_number_3,
                      int parameter_number_4,int parameter_number_5,int parameter_number_6);
```

2.2 Sobre la indentación

La cantidad de caracteres para la indentación son de 4 espacios, para poder diferenciar los ambientes.

Ejemplo de Indentación

```
void swap_int(int& number_1,int& number_2)
{
....int temporal=number_1;
....number_1=number_2;
....number_2=temporal;
}
```

2.3 Sobre los Ámbitos

Las llaves deben ocupar toda una linea, pueden ser obviados si el ámbito solo tiene una linea de código para casos del **if**, **else**. **while** y **for**; cada vez que se crea un nuevo ámbito, aumentar la indentación para diferenciarlos.

Ejemplo de Ambientes

```
int main()
{
....int test_var=0;
....while(test_var<10)
....{
........if(test_var%2)
............std::cout<<"a test line"<<std::endl;
........++test_var;
....}
....return 0;
}
```

En el caso de un **switch** y sus **case's**, es necesario crear llaves en cada **case** para diferenciar su ámbito y no olvidar la indentación.

Ejemplo de switch

```
int main()
{
....int test_var=0;
....swtich(test_var)
....{
........case 1:
........{
............std::cout<<"test option: 1"<<std::endl
............break
........}
........case 2:
........{
............std::cout<<"test option: 2"<<std::endl
............break
........}
........default:
............std::cout<<"test option: default"<<std::endl
....}
....return 0;
}
```

2.4 Sobre la Nomenclatura

La nomenclatura de variables, funciones, clases, estructuras, templates, definiciones de tipo, ...; deben describir su uso en el código.

2.4.1 Variables

En minúsculas y cada palabra separada por '_'.
Ejemplos:

- **int** iterator_x;
- **float** display_pos_x;
- **string** user_message;

2.4.2 Constantes y Macros

En mayúsculas y cada palabra separada por '_'.
Ejemplos:

- **LL_DEFAULT_OPTION**
- **MATH_PI**
- **AC3_CAR_AUTOCHASE_PATH**

2.4.3 Funciones

En minúsculas y cada palabra separada por '_'; usar prefijos para describir funcionalidades, como obtención de datos **(get)** o modificación de datos **(set)**; y los parámetros tienen las mismas reglas que las variables.
Ejemplos:

- **int** get_pos_x();
- **void** set_pos(**float** new_pos_x,**float** new_pos_y);
- **std::list<std::string>** split_string(**std::string** str,**char** split_char='\n')

2.4.4 Punteros a Funciones

En minúsculas y cada palabra separada por '_', y añadir el prefijo **Function_** para poder reconocerlos fácilmente.
Ejemplo:

- **int** (*Function_pointer_to_main)(**int**,**char****);

2.4.5 Definiciones de Tipo

En minúsculas y cada palabra separada por '_', añadir el prefijo **Type_** para poder reconocerlos fácilmente.
Ejemplos:

- **typedef pair<float,float>** Type_point;
- **typedef list<float>::iterator** Type_float_list_iterator;

2.4.6 Clases, Estructuras y Enumeraciones

Deben tener Mayúscula inicial por cada palabra usada, sin separadores, recomendado que el tamaño del nombre sea de 10 caracteres como máximo.
Ejemplos:

- **class** RTree;
- **struct** Point3D:**public** Point2D;
- **enum** KeyEvent;

2.4.7 Espacio de Nombres

Debe contener al inicio las siglas de la librería a la que pertenece en Mayúsculas, si se quiere separar por módulos se aumenta un '_' seguido del nombre del módulo, el nombre puede contener cualquier carácter permitido a excepción del guión bajo, recomendable que el largo sea máximo 15 caracteres.
Ejemplos:

- **namespace** LL;
- **namespace** LL_AL5;
- **namespace** LL_ENet;
- **namespace** AL_Display;

2.4.8 Iteradores

Tienen las mismas reglas que las variables, pero se abre una excepción, se puede usar de nombre "i", "j", "k", ...; mayormente usado en los ciclos contadores (Counting Loops) como el **for**.
Ejemplo:

- **for**(**unsigned int** i=0;i<**SIZE_VECTOR**;++i);

2.4.9 Variables del Template

En mayúsculas y cada palabra separada por '_', pero hay una excepción para las variables de tipo **typename** o **class**, donde puede usarse "T" como nombre.
Ejemplos:

- **template**<**typename** T>
- **template**<**unsigned int** SIZE,**int** DIMENSION>

2.5 Sobre los Valores Constantes

Para diferenciar los tipos de valores constantes, es necesario usar los que si demuestren su tipo de dato.

- **Valores Booleanos:** Usar **true** y **false**.
- **Valores Flotantes:** Usar siempre el punto flotante ('.')
  Ejemplo: 0.0, 1.0, 3.1416.
- **Punteros:** Usar **nullptr** que nos brinda C++11, para la representación de punteros nulos.

2.6 Sobre la Funcionalidad de los Operadores

Operadores Comunes:

- **operador** =
  Operador de Asignación.
- **operador** ==
  Operador Booleano de Igualdad.
- **operador** !=
  Operador Booleano de Desigualdad.
- **operador** !
  Operador de Negación.
- **operador** ()
  Operador para que un Objeto sea usado como Función. (Solo usado en Clases)
- **operador** []
  Operador de Acceso a una estructura por índices, de preferencia retornar por referencia "&" para la interacción con la estructura. (Solo usado en Clases)

Operador de Conversión Automática de Tipo:

- **operador** <**type**>
  Operador de Conversión Automática.

2.7 Sobre los Iteradores (iterator)

Si se crea un iterador para una estructura, este debe ser creado con el nombre "**iterator**".
La estructura debe contener dos funciones para manejar estos iteradores:

- **iterator** begin() - retorna el iterador al inicio de la estructura.
- **iterator** end() - retorna el iterador que indica fin de la estructura.

El iterador debe contener como mínimo las siguientes funciones y operadores:

- **iterator** () - un constructor vacío.
- **iterator** operator = () - copia de iteradores.
- **bool** operator != () - operador booleano de desigualdad entre iteradores.
- **iterator** operator ++ (**int**) - recorrer al siguiente elemento, retornando el anterior iterador.
- **iterator** operator ++ () - recorrer al siguiente elemento, retornando el nuevo iterador.
- **<type>** operator * () - acceso al valor actual del iterador.

2.8 Sobre los Niveles de Acceso, Métodos y Atributos de las Clases (class)

2.8.1 Orden de Declaración

En el nivel de acceso donde se declaren los métodos y atributos debe existir un orden con fines de ubicación, este orden se da más adelante seguido con su respectivas reglas y uso de prefijos.

1. **Constructores**
   Pueden ser Opcionales.
   Si es necesario un constructor vacío, las llaves pueden estar en la misma línea de declaración.

2. **Definiciones de Tipo**
   Si son privadas o protegidas se debe aumentar el prefijo **_T_** antes del prefijo **Type_**.

3. **Constantes**
   Deben tener el prefijo **CONSTANT_**.
   Si son privadas o protegidas se debe aumentar el prefijo **_C_** antes del prefijo **CONSTANT_**.

4. **Estructuras**
   Deben tener el prefijo **Structure_**.
   Si son privadas o protegidas se debe aumentar el prefijo **_S_** antes del prefijo **Structure_**.

5. **Clases**
   Deben tener el prefijo **Class_**.
   Si son privadas o protegidas se debe aumentar el prefijo **_C_** antes del prefijo **Class_**.

6. **Iteradores**
   Siempre deben ser públicos.

7. **Variables**
   Si son privadas o protegidas se debe aumentar el prefijo **_V_**.
   Recomendable evitar declarar variables públicas en clases.

8. **Punteros a Función**
   Si son privadas o protegidas se debe aumentar el prefijo **_P_** antes del prefijo **Function_**.

9. **Funciones**
   Si son privadas o protegidas se debe aumentar el prefijo **_F_**.

10. **Operadores**
    Primero declararse los operadores comunes.
    Finalmente declararse los operadores de conversión automática de tipo.

11. **Destructores**
    Pueden ser Opcionales.
    Si son destructores virtuales pueden declararse en una sola línea si no tiene contenido.

2.8.2 Niveles de Acceso

Los niveles de acceso pueden repetirse para establecer un orden de declaración, cuando se abre un nivel de acceso, la indentación debe ser aumentada.
Las estructuras solo deben ser usadas en clases que tengan puros métodos y atributos públicos, sino usar clases. Estos igualmente deben seguir el orden de declaración.

2.8.3 Ejemplo

Ejemplo Completo de una Clase

```
template<typename T>
class ExampleDataStructure
{
....private:
........typedef pair<int,T> _T_Type_data;
........const unsigned int _C_CONSTANT_MAX_SIZE=100;
........struct _S_Structure_Controller;
........class _C_Class_ListNode;
........unsigned int _V_size=0;
........list<_C_Class_node> _V_data_list;
........bool (*_P_Function_to_verify_data)(T)=nullptr;
........void _F_insert_node(T data_value);
....public:
........ExampleDataStructure(){}
........ExampleDataStructure(bool (*Function_to_verify_data)(T)){}
........class iterator;
........iterator begin();
........iterator end();
........unsigned int size();
........T operator [] (unsigned int index);
........operator bool ();
........virtual ~ExampleDataStructure(){}
}
```

3 Administración de Archivos

Cada **namespace** tiene una librería al que pertenece, por cada librería crear un directorio con su nombre, si estas librerías tienen módulos, estos deben tener sus propio subdirectorio, también se pueden crear más subdirectorios si estos desean ser separados como cabeceras especiales.
Evitar la dependencia entre módulos y en especial que estos pertenezcan a librerías diferentes.
El nombre de las cabeceras deben ser relacionado con el contenido, por ejemplo si se tiene solo una clase, puede llamarse como el nombre de la clase.
En cada cabecera se escribe una MACRO para evitar la doble definición de su contenido, se debe escribir de la siguiente forma:

```
#ifndef INCLUDED_<HEADER_MACRO>_H
#define INCLUDED_<HEADER_MACRO>_H

// El Código va Aquí.

#endif // INCLUDED_<HEADER_MACRO>_H
```

El valor de <HEADER_MACRO> puede ser:

- <LIB_ACRONYM>_<MODULE_NAME>
- <LIB_ACRONYM>_<MODULE_NAME>_<FILE_NAME>

4 Notas

Toda esta documentación es dada para que se pueda conocer sobre las convenciones de código usadas en las Cabeceras de LexRis Logic.

5 License