Introducción

El siguiente compendio de ideas para la programación en C no debe ser visto como un cuerpo de reglas rígidas. Se pretende que sea un conjunto de sugerencias para que todos podamos trabajar mejor y crear código más legible, eficiente y con menos errores.

Tipos y variables

- La notación húngara no es bienvenida en código nuevo. Contribuye a malas prácticas de programador, y no participa en la idea de que el programador debe manipular abstracciones para poder dedicarse mejor al problema que realmente le atañe.

- Cada conjunto de valores que merezca un tipo, deberá ser manejado con un tipo aparte, de esta manera se abstrae al programador del detalle de qué debe ser ese tipo. Tambien se reserva uno la posibilidad de alterar el tipo que se usa para esos valores.

- Se deben, en lo posible, usar los tipos nativos del lenguaje. Es decir que en C++ se usará bool y no alguna extraña macro BOOL o BOOLEAN.

- Las variables de tipo lógico o booleano deberán en lo posible tener nombres que se vean como predicados. Ejemplo: Si hay una variable estamos_fritos, podremos escribir: if(estamos_fritos) abort();
. Anti-ejemplo: En cambio, si la variable se llama
estado_fritura
, el código escrito no queda igual de evidente:
if(estado_fritura)
....

- Las funciones y variables globales deben tener nombres descriptivos, sin importar que sean largos. Ejemplo: cerrar_conexion().

- Las variables de corto alcance, por ejemplo un contador, deberían tener nombres cortos y simples.

- Las variables y funciones globales que pertenezcan a un módulo o biblioteca reusable deberán estar demarcadas, cuidando así el espacio de nombres. En C esto quiere decir que se usará un prefijo delante de los nombres. En C++ se deberán usar los namespaces del lenguaje.

- Se deben preferir los enums por sobre las macros del preprocesador. Proveen mayor chequeo de tipos y son vistas por los debuggers.

Documentación

- Cada función deberá estar documentada en el código: Qué hace. Cuáles son sus parámetros. Esto deberá hacerse utilizando la sintaxis que requiera el generador automático de documentación. Por ejemplo si se usa doxygen, esto se vería algo así:

```
codeDivStart()/** Divide dos enteros.
 *  Esta función divide dos enteros y no hace más que eso.
 *   @param numerador   lo que va a arriba
 *   @param denominador   lo que va abajo
 *   @return         el cociente
 */
int dividir(int numerador, int denominador)
{
```

- No se deberá documentar de más. La documentación debe hablar generalmente sobre el qué, y no sobre el cómo. Anti-ejemplo:

```
codeDivStart()/* Le asignamos a la variable a el valor 3 */
a = 3;

/* Invocamos a la función que corta el pasto */
cortar_pasto();
```

General

- El código no deberá provocar warnings al ser compilado.

- Todas las funciones deberán tener siempre sus prototipos.

- Se deberán incluir todos los .h necesarios de modo de nunca depender de que el compilador asuma implicitamente cómo está definida una función.

- Los compiladores deberán estar configurados (por ejemplo desde el Makefile correspondiente) para emitir una generosa cantidad de warnings.

- Toda función cuyo ámbito de aplicación se circunscriba al archivo en la que está siendo definida deberá ser declarada static. Esto evita colisiones de nombres con otras partes de la aplicación, además algunos compiladores pueden aprovechar esto y mejorar los warnings y/o generar código más eficiente.

- Se prefiere que una función static esté definida antes de su uso, para evitar la necesidad de prototipos que en este caso son redundantes.

- Una función no debe extenderse por más de una pantalla o dos. Se debe aislar unidades funcionales y codificarlas en funciones aparte aún si esas funciones solamente serán llamadas en un solo lugar. Esto contribuye en gran manera a la lectura posterior del código.

- Es preferible usar la devolución de la función para realmente devolver lo que la función calcula, en vez de devolver un código de error y poner el resultado en un parámetro pasado por referencia. Ejemplo: Una función dame precio se prefiere que sea precio = dame_precio() en vez de
st = dame_precio(&precio);
. Si se quiere devolver un código de error, mejor hacerlo como parámetro de salida:

```
codeDivStart()char *error = NULL;
  precio = dame_precio(articulo, &error);
  if(error)
  {
    fprintf(stderr, "error: %s
", error);
    free(error);
  }
```

Esto tiene la ventaja de facilitar el propagado de los errores hacia arriba. Como ejemplo, la función dame_precio podría ser así:

```
codeDivStart()double dame_precio(const char *articulo, char **error)
{
```

```
    return obtener_valor(articulo, "precio", error);
}
```

Si hubo error en obtener_valor, éste se pasa hacia arriba sin más problemas.

- Se deberá usar const en los casos que corresponda. Por ejemplo:

- En parámetros string que sean de entrada y no se toquen en la función.
- En variables string a las que se asignará un "literal" string.
- En las funciones miembro de una clase que no alteren al objeto.

- No se debe abusar de las macros. Si un string aparece una sola vez, y nunca va a cambiar probablemente un define solamente haga más compleja la lectura del código. Las macros con parámetros a menudo es más conveniente codificarlas como funciones inline.