Universidad Tecnológica Nacional

Facultad Regional Buenos Aires

Departamento de Electrónica

Asignatura: Informática I

Buenas prácticas de programación en lenguaje C.

Autor: Salanitri Sergio, Ayudante TP

Introducción

En la práctica cotidiana existen diferentes formas de hacer las cosas, todas pueden tener el mismo resultado, pero la gran diferencia es como se ejecutan. Esta diferencia influye en la calidad del producto realizado.

La práctica de la programación no es una excepción y por lo tanto es muy conveniente seguir buenas prácticas de desarrollo para tener un código sea más fácil de mantener, corregir, expandir por otros programadores y hasta por nosotros mismos pasado cierto tiempo.
Quién de nosotros no ha visto un código en donde la primera pregunta es cómo pudieron llegar a hacer algo así y que funcionase y lo más increíble aún es que lo hayan mantenido y mejorado en funcionalidad a pesar de estar construido sin comentarios, con funciones de decena de líneas (o peor aún sin funciones), con nombres de variables cripticas etc. Evidentemente fue con mucho esfuerzo y poca eficiencia.

El siguiente documento describe las buenas prácticas de programación a nivel general de lenguajes de programación orientados a procedimientos y en particular para el lenguaje de programación C.

Clasificación

Estas buenas prácticas las clasificaremos en diferentes grupos.

Nomenclaturas: Se refiere a las reglas aplicables a la denominación de nombres de variables, constantes, funciones, estructuras etc. Estas reglas tiene como objetivo definir un lenguaje común para facilitar la lectura y mantenimiento de un código. Son reglas generales de muchos lenguajes de programación, que son aplicables perfectamente al lenguaje C.

Sintaxis: Se refiere a reglas de la sintaxis en si del lenguaje de programación, y recomendaciones para hacer mas legible el código generado.

Manejo de errores y excepciones:

Este tema se podría incluir dentro del grupo de sintaxis (mejores prácticas para manejar los errores) o patrones (patrones de manejo de errores), pero como se considera un tema muy importante en el desarrollo de software de calidad se lo agrega como item aparte.
Se define como un error en un programa a un defecto (bug) que provoca que el programa no funcione correctamente en todas las funcionalidades para el cuall fue diseñado. Un error puede provocar que un cálculo devuelva un dato incorrecto o que el sistema se "cuelgue" en ante ciertos parámetros de entrada. Durante la etapa de pruebas se debe detectar la mayor cantidad de errores posibles, para luego ser corregidos antes de publicar el programa para su utilización productiva. También existen errores provocados por eventos externos a nuestro programa, como por ejemplo la no existencia de un archivo, permisos inadecuados para

ejecutar determinada acción etc. Ante estos errores, que deben ser considerados en el diseño, el sistema debe responder con un código de error para que el programa que invoque la función pueda actuar en consecuencia.

En cambio una excepción en un problema inesperado que no tiene un manejo como el caso del error. En el entorno Linux se pueden utilizar diferentes técnicas para manejar las excepciones, como por ejemplo intercepción de mensajes. Lamentablemente el lenguaje C no tiene un buen manejo de excepciones en forma nativa y se debe recurrir a soluciones externas como la mencionada.

Patrones: Los patrones son diseños y recomendaciones generales que solucionan problemas comunes de determinados contextos. Existen patrones de diseño aplicados a diferentes tipos de lenguajes de programación (orientados a procedimientos, objetos, eventos, web etc), para diferentes contextos de ejecución (multitareas, tiempo real, distribuidos, ejecución batch etc), para diseño en diferente nivel de abstracción de un sistema, para integración entre sistemas que corren en la misma o diferente plataforma etc.
En nuestro caso nos dedicaremos a los aplicables a lenguajes de programación orientada a procedimientos (como el C).

Nomenclaturas

Esta sección describiremos las buenas prácticas a la hora de definir elementos que utilizaremos dentro de nuestros códigos, estos comprendes varialbes, constantes, estructuras, enumeradores, funciones etc. El buen uso de estas prácticas permite tener un código más legible y por lo tanto más fácil de mantener, modificar y depurar tanto para alguien que lo tenga que mantener para mejoras o resolución de bug o para el mismo desarrollador que luego de varios meses ha olvidado detalles del código creado.

Variables:

.
Se recomienda definir las variables con nombre que sean autodescriptivos, escritas en el mismo idioma (se sugiere usar el inglés) y nomeclatura camel (primera palabra del nombre toda en minúsculas y la primera letras de las siguientes palabras en mayúsculas). Además se debe tener un prefijo que indique el tipo de datos (i = int, c = char, p = pointer, f = float etc).

Ejemplos:

cInBuffer, iUserID, cLogFile,iInPort, cConfigurationFile, cImageFile etc.

Variables de iteración locales: Se recomienda utilizar nombres variables cortas tales como i,j,ii,x,y pues tiene un ámbito de uso muy limitado (dentro de un ciclo for, while o do while).

Lo que no se recomienda hacer:

- Abreviaturas cripticas, por ej iB (en vez de inBuffer), cFile (configurationFile), bArrH (por bufferArrayHuffman)

- Nombres en spanisglish, com por ejemplo, entradaBuffer, solicitudDeData, serialPuertoAddress, usbEntradaBuffer etc.

Constantes:

Las constantes siempre se escriben todo en MAYUSCULAS, las mismas pueden ser globales o locales a una determinada función. Se recomienda en uso de contantes definidas con la directiva **#DEFINE** para constantes globales y **const** para constantes locales.

Ejemplos:

```
#DEFINE PORT1 8080

void function1()
{
const int CONSTANTE = 100;
        …….
}
```

Lo que no se recomienda hacer:

```
#DEFINE Constante 100
#DEFINE constante = 200
void function1()
{
const int consTante = 100;
…….
}
```

Comentarios

Es común en los programadores noveles subestimar los comentarios, pero es altamente recomendable realizar comentarios por lo menos con la implementación de cada función y con los bloques de código que por su complejidad requieran aclaración adicional. Esto permite tener una descripción del código, los motivos de la realización de determinada operaciones y tener una documentación muy útil a la hora del mantenimiento del mismo tanto sea por otro programador como por el mismo creador del código original pasado varios meses de construido.

Sintaxis

Este apartado no está destinado a describir la sintaxis del C sino las buenas prácticas de cómo se debe escribir el código en pos de generar código más fácil de comprende y por ende más fácil de mantener.

Identación

La identación es una práctica altamente recomendable para mejorar la legibilidad del código. Al igual que las buenas prácticas de nomenclatura la violación de practicas de sintaxis no impide la compilación del programa pero hay que tener en cuanta que los mismos son creados y mantenidos por personas y aplicar estás practicas nos puede evitar bastantes dolores de cabeza cuando dentro de varios meses queramos modificar el código, peor aún si ese código tiene que ser modificado por otros miembro del equipo.

Ejemplos:

```
if ( a == b)
{
        b =  SomeWhere();
a++;
if(a = MAX)
        return a;
else
        return b;
}

int i = 0;
for( i=0 ; i < MAX ; i++)
{
        arrayA[i] = arrayB[i+1];
        arrayA[i+1] = MAX;

}
```

Lo que no se recomienda hacer:

```
if ( a == b)
{
b =  SomeWhere();
a++;
if(a = MAX)
return a;
else
return b;
}

if ( a == b)
```

```
{
b =  SomeWhere();
a++;
if(a = MAX)
return a;
else
return b;
}

int i = 0;
for( i=0 ; i < MAX ; i++)
{
arrayA[i] = arrayB[i+1];
arrayA[i+1] = MAX;
}
```

Uso de llaves {}

El C como en otros lenguajes las llaves {} indican principio y fin de un bloque de código o definición de tipo (función, if, while, struct, for etc)
Al continuación se describe algunas recomendaciones:

- Al iniciar el bloque de código abrir y cerrar las llaves y luego empezar a escribir dentro de ellas, esto evita un clásico error de sintaxis que se produce cuando la cantidad de llaves abiertas difiere de las cerradas. Muchos IDEs facilitan estas tareas de edición.
- Se prefiere poner la primera llave abierta debajo de la sentencia y no al lado, esto facilita la lectura.

Por ejemplo

```
int i = 0;
for( i=0 ; i < MAX ; i++)
{
        arrayA[i] = arrayB[i+1];
arrayA[i+1] = MAX;

}
```

Y no

```
int i = 0;
for( i=0 ; i < MAX ; i++){
        arrayA[i] = arrayB[i+1];
arrayA[i+1] = MAX;
}
```

En este caso es una regla sutil que depende de la costumbre de cada programador.

Manejo de errores

En este apartado nos dedicaremos a describir buenas prácticas de programación para el correcto manejode errores, el tema del manejo de excepciones queda fuera del alcance de este docuento.

Valores de retorno:

La mayoria de las APIS estándar dela lilbc devuelve un valor entero igual o mayor que 0 si la operación se ejecutó con éxito y < 0 si hubo error. Hay casos con el función open() que devuelve el descriptor del archivo en caso de exito (que es mayor que 0) y -1 si fallo. En todos los casos se debe leer la documentación correspondiente usando el comando "man".

Por lo tanto en una primera aproximación podríamos validar esa salida como en el siguiente ejemplo:

```
if((res = sampleFunction()) < 0)
{
   perror("Ha ocurrido un error en la función sampleFunction()");
   exit(-1);
}
```

Analizando este sencillo ejemplo tenemos que si bien se valido la salida de la funciona y se presento un mensaje, no se sabe cual fue el motivo del error. A tal efecto muchas bibliotecas de C cargan la variable **errno** con un valor entero diferente para cada tipo de error (ver man errno) y luego devuelve -1 al proceso que la invoca para indicar que hubo un error.
Por ejemplo para la función open() errno puede estar cargado entre otros los siguiente valores:

- EEXIST camino ya existe y se usaron O_CREAT y O_EXCL.

- EISDIR camino se refiere a un directorio y el acceso pedido implicaba escribir (esto es, O_WRONLY o O_RDWR estaban activos).

- EACCES El acceso pedido al fichero no es permitido, o uno de los directorios en camino no tiene permiso de búsqueda o paso (ejecución), o el fichero todavía no existe y el directorio padre no tiene permiso de escritura.

Entonces considerando el valor de **errno** podemos devolver más información sobre el error, por ejemplo supongamos que se quiere abrir un archivo al cual no tenemos acceso, la función open cargara errno en EACESS. Se recomienda construir funciones independientes para manejar cada tipo de error (error handles). En caso triviales puede resultar una suma de complejidad innecesaria, pero en la práctica

muchas veces es necesario registrar el error en un archivo de log, restablecer ciertos estados del programa conviene hacerlo en una funcion separada (aplicación en patrón de separación de conceptos). Para simplificar el ejemplo no se implementan estos handles que lo que hacen es cargar en la variable errorMess el mensaje correspondiente a ese tipo de error.

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int main(void)
{
	char* errorMess;  //Se carga con el mensaje de error especifico de cada tipo de error.

	 //Si el archivo no existe sera creado.
	if((res = open("archivo",O_CREAT)) == -1)
	{
		switch(errno)
		{
		   case EEXIST: eexistHandle(&errorMess); break; //"El archivo ya existe"
		   case EISDIR: eisdirHandle(&errorMess);break;// "Es un directorio"
		   case EACESS: eacessHandle(&errorMess);break;// "Acceso denegado"
		   default: defaultErrorHandle(); //Error desconocido.
		}

		return(-1);
	}

	//Hace las operación especificada para este programa.

	return 0;

}
```

Patrones

Los patrones de software son buenas prácticas que solucionan problemas recurrentes durante el diseño y desarrollo de software.
A continuación se describen algunos conceptos importantes para poder analizar los patrones.

Acoplamiento:

Mide el grado de dependencia existente entre dos módulos software, tales como clases, funciones o bibliotecas. Del mismo documento de Ward Cunninghm mencionado se puede definir como “Dos módulos A y B, se dicen que están acoplados cuando al realizar un cambio en A si o si se debe realizar un cambio en B y viceversa”
Por lo tanto para que un diseño sea adecuado es deseable que los módulos del sistema tengan bajo acoplamiento.
Bajo acoplamiento no significa que un módulo está completamente aislado del otro, la comunicación se hace utilizando interfaces bien definidas y estables.

APIS:

Del inglés Application interfaces, son interfaces de programación que permiten acceder a ciertas servicios y funcionalidades con el fin de permitir al usuarios agregar nuevas funcionalidades a sistemas existentes o sistema propios creadas en bases a esas interfaces. En el caso particular de las apis de productos estos pueden ser tan genéricas como para permitir el total control del sistema desde ellas o restrictiva que solo permite acceder a ciertas funcionalidades. Es muy común que el proveedor del producto software indique en sus manuales que para poder agregar nuevas funcionalidades se deba utilizar en forma mandataria sus interfaces se aplicación. Es buena práctica del líder de producto recomendar que los desarrolladores no agreguen nuevas funcionalidades evitando el uso de las apis provistas (ej, mediante el acceso a los datos del sistema directamente). Las ventajas del uso de las apis son las abstracción del desarrolladores de la implementación interna para determinadas funcionalidades, para el lider asegurarse compatibilidad hacia nuevas versiones del producto y para el líder de proyecto mitigar los riesgos que implica construir funcionalidades personalizadas pues en caso de problemas se dispone del soporte técnico del producto.

Cohesión:

Indica el grado de relación de las características y responsabilidades de un determinado módulo software. Si un módulo tiene un valor de cohesión alto favorece el mantenimiento y la reusabilidad porque tienen menos dependencias.
Además un módulo con alta cohesión es más simple de probar, pues se debe probar solo las funcionalidades propias del mismo.
Como describe Ward Cunninghm (http://c2.com) básicamente “Si se tienen 2 módulos A y B, son cohesivos cuando un cambio de A no tienen repercusión en B mientras que ambos módulos pueden agregar valor al sistema”

Separación de conceptos (SoC):

Definido por Dijkstra en 1974 es su artículo “On the Role of Scientific Thought” define que los conceptos están asociados con los módulos software y en lo posible no deben tener funcionalidad duplicada. Dijkstra sugiere dedicarle atención a un concepto a la vez, pero sin olvidar los demás del sistema.

Modularidad:

Indica que se debe separar en módulos para cada funcionalidad significativa del sistema, este principio está asociado con SoC y ocultamiento de información.
Solo los miembros públicos de la interface son visibles al exterior del módulos, en programación basada en procedimientos serían los parámetros de entrada y salida a los procedimientos y los procedimientos expuestos en las bibliotecas para ser usadas desde otras aplicaciones.
En el caso de programación orientada este principio es más fácil de cumplir porque las clases encapsulan funcionalidad y propiedades dejando exponer solo lo necesarios.

Ocultamiento de información:

Este principio de diseño indica que al aplicar los principio de SoC y modularidad se pueden exponer interfaces definidas y estables de forma tal que al consumidor no le interesa la implementación interna. Mientras se mantenga la interface se puede cambiar la implementación interna sin afectar a los consumidores.

Para el caso de programación en C, se tiene que considerar que esun lenguaje orientado a procedimientos y por lo tanto no se tiene las ventajas de encapsulamiento que tiene el C++, pero de todas formas se pueden aplicar estas definiciones.

Consideremos el caso de tener una biblioteca de funciones que será generada para ser reutilizada por otros programas que pueden o no estar hechos en C.
También aplica a funciones dentro o fuera de una bilioteca.

Si está biblioteca tiene muchas dependencias con otras propias o no estaríamos en el caso de que tenga un alto acoplamiento pues el cambio de cualquiera de las dependencias afecta a nuestra biblioteca.
En cambio si depende de pocas bibliotecas externas y usa mucho más las APIS nativas del kernel seria el caso de bajo acoplamiento.

En cambio, una biblioteca tiene alta cohesión cuando todos sus funciones internas tiene una funcionalidad asociada, por ejemplo una biblioteca que se generó para facilitar el manejo de archivos podría tener funciones tales como OpenFile(), ReadFile(), DeleteFile(), ChangeFileAttribute(), WriteFile(), SearchInFile() etc.
Si en además tiene funciones tales como SetUserAttributes(), ReadSerialPort() (aunque el puerto serie en Linux se maneja como archivo), SetDisplay() etc estaríamos en el caso de una biblioteca con baja cohesión pues estas últimas funciones no tiene relación con el manejo de archivos.

Por lo tanto una buena práctica de diseño de bibliotecas es que tengan alta cohesión y bajo acoplamiento.
Ambos nos son mutuamente excluyentes pues una biblioteca puede tener alta cohesión pero bajo acomplimiento y viceversa.

La separación de concepto como lo indica la definición de más arriba implica que entre varios módulos software (en nuestro caso ponemos como ejemplo las librerías) no deben tener funcionalidad compartida, por ejemplo la librería A tiene la función ReadFile() y la B otra que se llama ReadTextFile() que sería una caso particular de la primera. Lo mejor sería poner ambas en una sola biblioteca o bien unificar funcionalidad en una solo función.

La modularidad y el ocultamiento de información son atributos que estás relacionado y por lo tanto importante a la hora de diseñar bibliotecas siguiendo buenas prácticas.