

Buenas Prácticas de C

Autor: Joaquín Azcárate

Este documento no pretende enseñar a programar, simplemente es una recopilación de lo que a mi, personalmente, me parecen buenas prácticas de desarrollo.

Las "ovbias"^[1]

Temo necesario tener un apartado para enlistar las típicas cuestiones en programación. Son prácticas globalmente aceptadas, y de las cuales no tengo más que enlistarlas. Confío que son suficientemente autoexplicativas, o que lo escuchaste en otro momento:

- Variables, funciones, o cualquier cosa que se nombre, que sea significativos; basta de `aux` , `aux2` , `var` ...
- Buscar en google por un snippet esta bueno, siempre y cuando entiendas lo que copy-pasteás.
- Seguir una nomenclatura a lo largo de los proyectos y el grupo.
- Comentar código y poner buenos textos en commits.
- Tratar de evitar funciones con muchos parámetros.
- Evitar el uso de `goto` s.
- Armar funciones que hagan una sola tarea (y que el nombre de cuenta de cual es esa tarea).

El mítico "código espagueti"

Empiezo por una de las mayores críticas que tengo sobre la programación de los TPs que he visto, el mítico: *código espagueti*. Si cursaste / estás cursando Paradigmas de Programación, probablemente escuchaste hablar sobre este extraño fenómeno.

Con algo de suerte puedes poner una mano en tu corazón y jurar que nunca incursionamos en prácticas del ocultismo espaguetital, pero C + un TP complejo + desconocimiento = la receta para terminar con un monstruo del estilo:

```

1  int main() {
2      while (vivo()) {
3          while (edad < 25) {
4              if (edad < 19) {
5                  yo->destino = ESCUELA;
6                  while (yo->lugar != ESCUELA)
7                      mover(yo, ESCUELA);
8                  estudiar(ESCUELA);
9              } else {
10                 yo->destino = FACULTAD;
11                 while (yo->lugar != FACULTAD)
12                     mover(yo, FACULTAD);
13                 estudiar(ESCUELA);
14             }
15         }
16         if (dia == 365) {
17             edad++;
18         }
19         if (dia % 7 == 6 || dia % 7 == 7) {
20             if (condicionesClimaticas == LINDO) {
21                 if (hora > 12 && hora < 16 && suenio()) {
22                     dormir();
23                 } else {
24                     salir();
25                 }
26             }
27             if (hora > 23) {
28                 dormir();
29             }
30         }
31         if (hora > 20) {
32             dormir();
33         }
34     }
35 }

```

c

Ademas de inmantenible e indebuggeable, poco se entiende. ¡Recordemos que estás trabajando con otras 4 personas!

Síntomas de código espagueti: la cantidad de sentencias de `if` y `while`. Pero a no desesperar, estar leyendo esto es el primer paso de la rehabilitacion. Sería un mentiroso diciendo que mi código, cuando curse la materia, no se parecía a "eso"... incluso creo que tenía menos llamadas a funciones.

Función-a

Despues de machacar la idea de que usar funciones es bueno, puede que ocurra el proceso contrario, donde tenemos lo que yo llamo "espagueti funcional", en donde no es muy claro que tiene que hacer una función, simplemente está porque -Me dijeron que ponga algunas funciones-.

Me niego a que alguien haga algo porque "le dijeron", sin muchas mayores razones.

Síntomas de este tipo de programación de funciones "forzadas", son que para cada función le tenes que pasar varios parámetros, o confían en muchas variables globales, y no se les puede poner un nombre lindo. Particularmente esta última razón, aunque bastante "tonta" es de las más útiles.

Esto nos trae a la discusión de cuándo parar con llamadas a funciones, y está a criterio de cada uno. Lo que sí es universal (por lo menos para mí) es que la repetición de código es de las cosas menos deseables.

Si tiene una función de una línea, que se repite con el mismo propósito, entonces esa función esta perfecta.

Si tiene una función de una línea que solo se invoca una vez, podría admitir desecharla, pero la verdad es que pensar cuándo poner una llamada a una función y cuándo no me lleva mas trabajo que hacer la llamada y potencialmente perder 2 microsegundos en la ejecución... ¡siempre y cuando el propósito no sea de optimizar!

Modularidad

Otro gran tema a abordar, esta linda palabra: *modularidad*, lo describe bastante bien.

La idea es muy simple, y aún más simple de lograr: *"La programación modular consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable"*.

C particularmente (como muchos otros lenguajes de programación) nos permite codificar en **más de un archivo**. Así como lees, no tiene porque estar todo en un archivo que se llame "main.c". Logramos esto con una directiva del preprocesador de C: `#include` .

En términos muy básicos, podés pensar que cuando apretás "compilar" lo primero que hace es: copiar textual todo el texto de el archivo a incluir, pegarlo donde esta la directiva, y después compilar eso en un archivo masivo.

Ahora bien, esto trae un par de problemas *interesantes*, como tener estos dos archivos:

```
corredor.c  sumar.c
1  #include "sumar.c"
2
3  void main() {
4      sumarTres(5);
5  }
```

De intentar compilar en forma automática esto, van a ver un error parecido a:

```
/sumar.c:3: multiple definition of `sumarTres'
./corredor.o: /sumar.c:3: first defined here
```

Lamentablemente tiene mucho sentido esto, si miramos a como intentamos compilar, podemos ver algo como:

```
gcc corredor.c sumar.c -o corredor
```

Entonces, básicamente estamos tratando de compilar dos archivos de esta forma (una vez pasado el preprocesador):

corredor.c sumar.c

```
1 int sumarTres(int numero) {  
2     return numero+3;  
3 }  
4  
5 void main() {  
6     sumarTres(5);  
7 }
```

c

Tiene mucho sentido que nos diga que esto no puede ser así, porque `sumarTres` está dos veces.

Ciertamente podríamos compilar solamente el `corredor.c` :

```
gcc corredor.c -o corredor
```

sh

Esto quiere decir que cada vez que editemos algún archivo que esté incluido por `corredor` (o por cualquier otro que esté incluido en alguno incluido), tiene que volver a compilar todo devuelta. Con el poder computacional que tenemos hoy en día, no parecería mucho problema, pero es una práctica pobre.

Cabezazos

A lo largo de los años se adoptó una forma de modularizar esto, con el agregado de un archivo como encabezado (aka: *header*), por lo que tendríamos estos **tres** archivos:

corredor.c sumar.c sumar.h

```
1 #include "sumar.h"  
2  
3 void main() {  
4     sumarTres(5);  
5 }
```

c

Ahora sí podemos compilar por separado cada `.c`, que genera dos objetos diferentes, y al final los "pega" todos juntos y nos da un ejecutable. Si alteramos el código de `sumar.c`, no necesitamos compilar todo devuelta, solo el objeto^[2] de `sumar.c` y volver a pegarlo.

Pero ahora tenemos un archivo con código, y un archivo diferente con la interfaz. Esto es agradable, pero no el final de la película. Si uno, sin querer, incluimos más de una vez `sumar.h`, la definición de nuestra función `sumarTres()` estaría definida más de una vez.

Este sería el error:

```
sumar.c: error: redefinition of 'sumarTres'
```

Suena raro incluir más de una vez un archivo, pero pasa más seguido de lo que se imagina. Para esto C nos trae otras directivas del preprocesador: `#define`, `#ifdef`, `#ifndef` y `#endif`.

Esto nos permite tener este código en nuestro archivo:

corredor.c sumar.c sumar.h


```
1  #include "sumar.h"
2
3  void main() {
4      sumarTres(5);
5  }
```

c

Analizemos un poco el código agregado de `sumar.h` :

- Preguntamos si no está definida `SUMAR_H_` . De no estar definida:
 - Definimos `SUMAR_H_` .
 - Agregamos el resto del código.

Si algún otro archivo intenta incluirlo, la primera línea le va a decir que ya fue definido, entonces no va a copiar y pegar el código. y esto es **bueno**, porque significa que ya fue copiado y pegado antes, no que no lo va a hacer ninguna vez.

Suele ser una buena práctica ponerle a la variable a definir, el nombre del archivo. A este *truco* se lo llama **guardas** .

Última cosa sobre headers, lo prometo!

Supongamos que ahora también queremos un tipo de dato, como que los números tengan también un `numeroAnterior` que guarde el número pre-suma. Tendríamos una estructura parecida a:

```
typedef struct {
    int numero;
    int numeroAnterior;
} t_numero;
```

c

El drama es donde ponerlo. las alternativas son:

1. `corredor.c` : Cuando intentamos compilar `suma.c` , que necesita saber como es la estructura, nos va a decir que no entiende que es un `t_numero` .
2. `sumar.c` : La función va a poder hacer lo que quiere, pero en `corredor.c` no vamos a poder crear un `t_numero` , porque no existe, y en el header no vamos a poder poner el correcto prototipo.
3. `sumar.h` : `sumar.c` no ve el código en su encabezado, porque no tiene ningún `#include` .

Las tres formas tienen problemas, pero hay una más fácil de arreglar que las otras:

corredor.c sumar.c sumar.h

```
1  #include "sumar.h"
2
3  void main() {
4      t_numero cinco;
5      cinco.numero = 5;
6      cinco.numeroAnterior = 0;
7      sumarTres(cinco);
8  }
```

c

Con algo de suerte ahora es un poco más evidente por qué ponemos las guardas, y cómo es que varios archivos pueden incluir a uno.

Tipación Inteligente

b² - 4ac (Discriminante)

Un pequeño paréntesis antes de hablar más seriamente de los tipos de datos, creo que merece esta idea de "discriminar"; en el concepto de "Separar, diferenciar una cosa de otra".

Muchas veces tenemos dos o más estructuras que son iguales en términos de datos, pero se manejan de forma muy diferente. Por ejemplo, si estamos programando un Sistema de Archivos, podríamos tener los **inodos** de archivos, de links y de directorios con los mismos atributos, pero con un tratamiento definitivamente diferente.

Ahora que tenemos nuestro código distribuido en varios archivos, de forma prolija y ordenada, podemos adentrarnos a formas de programar. Esta idea también aplica a el mandar mensajes con una cabecera describiendo qué tipo de mensaje es.

Una primera idea podría ser escribir 4 caracteres con el tipo de mensaje antes, y tener un código como:

```
1  if (!strcmp(buffer, "DATO", 4)) {  
2      procesarDato(buffer);  
3  } else if (!strcmp(buffer, "TIPO", 4)) {  
4      procesarTipo(buffer);  
5  } else if (!strcmp(buffer, "MSGE", 4)) {  
6      procesarMensaje(buffer);  
7  } else {  
8      error("No entendi el mensaje");  
9  }
```

Sería más feliz, probablemente, si C nos dejara hacer un switch con strings.

Como no nos deja, otra implementación podría ser (y es un **refactor** común, y bastante útil):

```
1  #define DATO 4  
2  #define TIPO 5  
3  #define MENSAJE 6  
4  
5  switch (buffer[0]) {  
6      case DATO:  
7          procesarDato(buffer);  
8          break;  
9      case TIPO:  
10         procesarTipo(buffer);  
11         break;  
12     case MENSAJE:  
13         procesarMensaje(buffer);  
14         break;  
15     default:  
16         error("No entendi el mensaje");  
17         break;  
18 }
```

Ciertamente existen menos ifs, y ya es una muy buen cambio; pero sigue teniendo dos "problemas":

1. Si queremos agregar un nuevo discriminador, tenemos que inventar números que sean diferentes a los anteriores.
2. Al debuggear, el pre-procesador nos borró la idea de `DATO` , `TIPO` , o `MENSAJE` , ahora solo aparecen numeros, entonces podemos ver que llegó un 4, pero no sabemos bien qué es, tenemos que consultar a los `#define` s.

Ciertamente no son problemas graves. La alternativa que les propongo es usar un tipo de dato que fue pensado para exactamente esto:

```
1  typedef enum {
2      DATO,
3      TIPO,
4      MENSAJE
5  } t_tipo;
6
7  t_tipo discriminador = buffer[0];
8  switch (discriminador) {
9      case DATO:
10         procesarDato(buffer);
11         break;
12     case TIPO:
13         procesarTipo(buffer);
14         break;
15     case MENSAJE:
16         procesarMensaje(buffer);
17         break;
18     default:
19         error("No entendi el mensaje");
20         break;
21 }
```

Si, el código del switch es idéntico, pero si alguien en alguna versión agrega un nuevo discriminador e inventa su número, y otra persona agrega otro discriminador, con el mismo numero: problemas. Más importante aún, si uno intenta ver en GDB (debugger) el valor de `discriminador` :

```
(gdb) print discriminador
$1 = DATO
```

La revancha del `#define`

"Ah, pero entonces, los `#define` s no sirven".

Nonononono, no pongan bytes en mis dedos. Esta directiva es sumamente útil, si lo que uno quiere hacer es un Copy-Paste automático. Incluso nos permite hacer algo como "copiar, pegar y remplazar" todo en un solo momento; a esto se le dice **macros** [↗](#), y no creo que este sea el documento para discutir el uso o abuso de los macros del preprocesador, si bien mantengo que son una herramienta sumamente poderosa (pero altamente peligrosa).

Mas que nada, aliento el uso de estas definiciones para cuestiones *hardcodeadas*.

Si se les presenta un caso en el que tiene que implementar una búsqueda recursiva, pero solamente tiene que evaluar 3 niveles, en una primera instancia podrían tener:

```
1  t_elemento buscarINodo(char* nombre, int nivel) {
2      if (nivel <= 3) {
3          t_nivel nivelABuscar = obtenerNivel(nivel);
4          t_elemento elemento = buscarEnNivel(nombre, nivelABuscar);
5          if (elemento != 0)
6              return elemento;
7          else
8              return buscarINodo(nombre, nivel + 1);
9      } else
10         return 0;
11 }
```

Probablemente la búsqueda siempre sea hasta el tercer nivel, y jamás cambie. Pero mejor pregunta que "¿Alguna vez cambiará?" puede ser "¿Qué tan difícil sería que cambie?", o mejor aún, en el contexto de este documento "¿Qué tan entendible es ver un 3?".

Esta idea de ver números en el medio del código sin entenderlos muy bien se la conoce como **"números mágicos"**.

Nos quedaría mas lindo leer:

```
1  #define MAX_NIVEL_A_BUSCAR 3
2
3  t_elemento buscarINodo(char* nombre, int nivel) {
4      if (nivel <= MAX_NIVEL_A_BUSCAR) {
5          t_nivel nivelABuscar = obtenerNivel(nivel);
6          t_elemento elemento = buscarEnNivel(nombre, nivelABuscar);
7          if (elemento != 0)
8              return elemento;
9          else
10             return buscarINodo(nombre, nivel + 1);
11     } else
12         return 0;
13 }
```

De necesitar cambiarlo, está en un lugar fuera de la función de búsqueda; incluso más de una función podría usar ese número, y de cambiarlo en un lugar, cambiaría en todos los lugares.

typedef es tu amigo

Una de los mayores problemas que he detectado es en el uso incorrecto de la memoria, y no me refiero a no hacer `free()`, sino más bien a intentar tratar todo como un `int` o un `char`.

"¿Al fin y al cabo es todo memoria, que importa el tipo?"

Es un buen argumento; tan bueno que me lo he dado muchas veces a mí mismo. Pero nada más lejano de la realidad.

Un hermoso ejemplo de esto son las **commons** que les damos. Tomemos como ejemplo el **encabezado de los bitarrays**. Todas las funciones manejan de una u otra forma un `t_bitarray`, que

básicamente es un puntero a `char` y un tamaño. ¿Por qué no usar esos dos valores y mandar punteros a `char` y tamaños?

Más allá de la comodidad de encapsular la información (que es tema para otra sección), de hacer esto, ¿qué te distingue una cadena, de un bitarray? ¿Se tratan de forma igual? La respuesta tendría que ser no, de tratarse de forma idéntica, no tendrían un nombre diferente!

Tal vez el ejemplo no queda muy claro, pensemos otro un poco más grueso: Supongan que tiene que emular manejo de memoria. Si tuvieron / tienen que hacer esto, probablemente estén tentados a decir:

```
"Es memoria, no se que hay adentro, entonces voy a definirlo como void* !"
```

Es una buena idea, pero rápidamente se dan cuenta que para acceder al byte número 700 tiene que hacer casteos de la forma: `((char *) memoria)[700]` ^[3], y después del 7mo casteo, se cansan y van a cambiar el `void*` por un `char*`. Todos lo hemos hecho.

Lo que quiero hacerlos reflexionar es que tal vez no sea estrictamente un `char*`, y no tenga el mismo tratamiento que un `char*`, es más bien un: `t_memoria`:

```
typedef struct {  
    char *dato;  
} t_memoria;
```

c

Pensemos en el costo computacional de esta una línea: 0.

Cuando se compila, esto pasa a ser un puntero (o sea, en definitiva, un número entero), no tiene más complicación que tal vez unos nanosegundos en el preprocesador.

¿Qué ganamos?

Supongamos que armamos todo un **TAD** ^[4] de memoria, con `crearMemoria`, `destruirMemoria`, `asignarMemoria`, `moverMemoria`, lo que fuera. Por alguna razón loca, nos equivocamos y tratamos de hacer algo como:

```
int tamanoMemoria = crearMemoria();
```

c

El compilador nos va a avisar prontamente que `crearMemoria` tendría que devolver un `t_memoria`, y vos lo estas poniendo en un `int`.

Esto funciona a la perfección, compila y ejecuta, pero no es lo que queríamos; y debuggearlo nos va a llevar más que esos nanosegundos de poner esa una línea de `typedef`.

WARNING

Los warnings del compilador de C son importantes (~~más este que estás viendo~~). Hay que prestarles atención, ya que *suelen* indicar un mal manejo de tipos.

Si no los convencí con eso, imaginen que ahora les parece sensato que quieran sincronizar la memoria; que solo se pueda acceder de a uno por vez.

La solución fácil es ir a las funciones que usan la memoria y ponerles un semáforo global. Pero, ¿y si tenemos varios segmentos de memoria que manejamos?

Teniendo el `typedef` es tan trivial como ir a la especificación del `t_memoria` y cambiarlo por algo como:

```
typedef struct {  
    char* dato;  
    sem_t* semaforo;  
} t_memoria;
```

c


Cambiar las funciones que usen la memoria para que usen el semáforo de forma acorde y **nada más**. No hace falta tocar el código principal, solo cambiar el TAD de memoria. ¿Maravilloso no? Esta increíble hazaña no podría ser sin el grandioso concepto de:

Encapsularidad^[4]

Wikipedia lo pone muy simplemente:

En programación modular, y más específicamente en programación orientada a objetos, se denomina encapsulamiento al ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas para ese objeto.

Ciertamente no tenemos objetos en C, pero podemos acercarnos a algo parecido.

La idea es, como vimos en el ejemplo de `t_memoria`, armar todo un TAD alrededor de un tipo de dato, entonces hacer cualquier cambio de la forma de manejarlo es sumamente fácil y afecta a la menor cantidad de componentes. De vuelta, las **commons**  son un muy buen ejemplo de esto.

Abort! Abort! ABORT!

Si llegas a un punto donde quieres abortar el programa por un error, en vez de poner `exit(-1)`, poner `abort()`.


Abort termina el programa, pero con una señal que GDB frena, como Segmentation Fault, por lo que uno tiene posibilidad de mirar el estado de todo el sistema antes de abortar, y tratar de entender por qué llegó hasta donde llegó.

Último comentario

¡Comenten su código!

Su "yo" del futuro se los va a agradecer. Si necesitan convencimiento, agarren algún código que ustedes mismos hayan escrito hace más de un mes (de Algoritmos y Estructuras de Datos, Paradigmas de Programación, lo que fuese). Sáquenle los comentarios (... como si pusieran) e intenten interpretar qué es lo que hace.

Si se van a llevar una cosa de todo este palabrerío que sea este: **¡Pongan comentarios útiles!**

-
1. Si, sé que esta mal escrito; es a propósito ↩
 2. Si compilamos con gcc sin ningún argumento en particular, todos los objetos y etapas intermedias se pierden, por lo que aunque tengamos separado en archivos, va a tener que recompilar todo. ↩
 3. El por qué del casteo esta explicado muy bonito en **otra guía**. ↩
 4. El término correcto es **encapsulamiento** . ↩