

This page last changed on 23 jul 2009 by unai.estebanez.

- 1 [Prácticas recomendables:](#)
  - 1.1 [Protección ante errores evitables mediante costumbre](#)
    - 1.1.1 [Usar Prefijos](#)
    - 1.1.2 [Inicialización y uso de variables](#)
    - 1.1.3 [Indentación](#)
    - 1.1.4 [Estilo robusto](#)
  - 1.2 [Uso del Stack](#)
  - 1.3 [Portabilidad](#)
  - 1.4 [Optimización](#)
    - 1.4.1 [Acceso a estructuras anidadas en bucles:](#)
  - 1.5 [Aplicaciones multithread](#)
    - 1.5.1 [Runtime de C traicionera](#)
    - 1.5.2 [Reentrantia y seguridad ante hilos \(reentrance and thread safety\)](#)
      - 1.5.2.1 [Reentrantia](#)
      - 1.5.2.2 [Thread safety](#)
      - 1.5.2.3 [Escribir funciones listas para multi-threading](#)
  - 1.6 [Herramientas](#)
    - 1.6.1 [Lex & Yacc](#)
  - 1.7 [Sobre el uso de memoria dinámica](#)
  - 1.8 [Notación a utilizar para la programación](#)

## Prácticas recomendables:

Llevo años programando y en muchas ocasiones vuelvo a tropezar con la misma piedra una y otra vez. En este documento trato de recoger mi experiencia personal para evitar caer más veces en los mismos errores.

### Protección ante errores evitables mediante costumbre

#### Usar Prefijos

- Poner prefijo a las funciones de distintas librerías. Es común que se llamen funciones igual y luego pases horas depurando hasta darte cuenta de que no está entrando por donde debe.
- Poner prefijo a los `#defines` `PREFIJO_NOMBRE_DEFINE` para evitar que coincidan los `defines` de varios `.h`
- Poner prefijo en general a todo lo susceptible de "pisarse" con otros nombres.

#### Inicialización y uso de variables

- Siempre inicializar las variables con un valor inocuo. Es muy común dejar variables sin inicializar que tendrán un valor indeterminado (en función de lo que hubiese en la memoria en ese momento dado). Ese valor puede producir errores inesperados por asumir que valdrán cero por ejemplo. A menos que haya un buen motivo, inicializar siempre las variables.
- Hay que tener cuidado con los tamaños de las variables y arrays a la hora de hacer copias. El operador "sizeof" puede darnos sorpresas. Personalmente recomiendo las siguientes prácticas

```
void mifuncion(void)
{
    int vector[5];
    int vector_auxiliar[5];
    memcpy( vector_auxiliar,vector,sizeof(vector)); // Esto funciona pero no lo recomiendo
    memcpy( vector_auxiliar,vector,sizeof(int)*5); // Forma recomendada
}
// Veamos un ejemplo de porque no recomiendo directamente el uso de sizeof sobre el nombre de variable.
f(char a[10])
{
    int i = sizeof(a);
```

```
printf("%d\n", i);
}

// sin embargo esto imprime 4, no 10. Esto es debido a que en C no se pasan
// como copia los arrays, aunque tu lo pongas como "a[10]" esto se traduce a un "char*" así que su tamaño es el
// de un puntero a char, es decir, 4.
```

- Evitar buffers overruns, esto se puede evitar en el manejo de cadenas utilizando las funciones de manejo de cadenas que admiten como parámetro un límite en el número de caracteres a copiar, por ejemplo, strncpy, snprintf, strncmp, etc.

## Indentación

- Identar usando el estilo K&R ([http://en.wikipedia.org/wiki/Indent\\_style](http://en.wikipedia.org/wiki/Indent_style)). Entre otras cosas nos evitará errores como el siguiente:

```
// Este código fallará, se quedará por siempre en el while
while(i > 0);
{
    // haz cosas...
    i--;
}
//Es más difícil caer en el error con este estilo.
while(i > 0){ // Es más difícil que te equivoques con el punto y coma debido a que pones el paréntesis en la misma línea
    i--;
}
```

## Estilo robusto

- La gente siempre se sorprende cuando ve en mi código la siguiente comparación (evitar errores con el operador ==):

```
// El cero a la izquierda????
if( 0 == variable){ // Este es mi modo de protegerme de errores como el que se muestra debajo.
// Haz cosas
}
// Bajo mi punto de vista evita errores como el siguiente
if( variable = 0 ){ // Esto siempre hace que entres en el if.
// Haz cosas
}
```

- Ayudate a ti mismo, usa los códigos de retorno y úsalos bien, es muy habitual desestimar los códigos de retorno de las funciones. Estos códigos están por algo, es probable que no puedas seguir con la aplicación y debas "abortar" el programa de forma ordenada. En cuanto a lo de "úsalos bien", esto tiene la siguiente explicación, es habitual encontrar cosas como esta:

```
if( VU_NO_ERROR == vu_get_var(variables[i].name,variables[i].owner_type,variables[i].owner_code,&var) ) {
    /* código del if*/
}
/* No puedo saber que valor me devolvió la función, solo se que falló */
```

esto se puede cambiar por:

```
int result;
```

```
result = vu_get_var(variables[i].name,variables[i].owner_type,variables[i].owner_code,&var);
if( VU_NO_ERROR == result ) {
/* código del if */
} else {
/* Puedo mostrar el error, o realizar cualquier procesado de su valor */
printf("Error:%d\n",result);
}
```

Sin duda esta última opción facilita la depuración.

## Uso del Stack

Es habitual que el programador no habituado a los embebidos utilice el stack de forma intensiva (yo mismo, a pesar de saber que tiene problemas, suelo tener la mala costumbre de abusar de él). Esto no es recomendable puesto que da lugar a problemas bastante complejos de depurar. Suele manifestarse con extraños casques o incluso con corrupción de otras variables en funciones padre.

Hay que familiarizarse con el hardware en concreto que vamos a usar, saber de sus limitaciones de stack y adaptarse a ellas. Es posible en ocasiones cambiar el tamaño del stack, pero hay que estar bien seguro de que esa opción funciona.

Ejemplo de uso del stack:

```
int mi_funcion(int param1)
{
int variable_en_el_stack[1000];
// código...
return 0;
}
```

Aquí hemos podido ver una reserva de  $1000 \times 4$  bytes en el stack, casi 4K, en algunos sistemas esto no será admisible. Podemos hacer varias cosas para evitar este problema:

- Usar menos variables o de menos tamaño
- Usar esas variables como "static" de forma que no se usa el stack. (son reservadas en el segmento de datos)
- Usar variables globales
- Usar memoria dinámica

Estas soluciones tienen, claro está, ventajas e inconvenientes, vayamos por partes:

Usar menos tamaño, es evidente, si simplemente puedes reducir su uso el problema desaparece.

Usar las variables como "static", tiene la ventaja de que las variables siguen siendo privadas a la función pero tiene dos inconvenientes.

1. El contenido de la/las variable/s permanece inalterado en sucesivas ejecuciones de la función, pudiendo dar lugar a errores de programación. Habría que inicializarlo explícitamente cada vez si queremos que tenga un valor seguro.
2. Las funciones que usan este tipo de datos generalmente no son multithread, ya que se guardan en memoria global las sucesivas ejecuciones de varios threads pueden arrojar resultados erróneos, casques, etc.. Si queremos que estas funciones sean multithread habrá que proteger los accesos a estas variables o a la función en sí mediante primitivas de sincronización como Mutex, etc...

Usar variables globales tiene las mismas pegadas que el anterior punto solo que además estas variables no son privadas y pueden ser usadas por cualquier función, con los consiguientes problemas.

Usar memoria dinámica (reservada en el heap) puede ser una solución interesante pero generalmente estos embebidos que tienen restricciones de stack suelen tener también pegadas con la reserva/liberación de memoria dinámica, pegadas que suelen ser aún más graves que el uso del stack (por ejemplo, crean fragmentación en la memoria y al final nos quedamos sin ella).

## Portabilidad

- Cuando tengas como requisito código portable, evitar C++, siempre intentar usar C. Esto incluye:
  - Ojo con el "inline". Inline esta siempre presente en C++ pero no así en C. En el C ISO no esta incluido. GCC trae esta posibilidad como extensión, por este motivo usar inline reduce la portabilidad de nuestro código. De hecho debemos usar la macro "\_GNUC\_" para comprobar que usamos el gcc y que está disponible esta característica. Además hay que tener en cuenta que el nombre "inline" se usa de diferentes formas en función del ISO que quieras soportar (ISO C89, ISO C99,...). Por otro lado tengo mis dudas (aún no lo tengo claro) de si el "inline" funciona al invocar la función desde un ejecutable que usa una librería dinámica (donde esta la función inline). Para más información ver: "<http://gcc.gnu.org/onlinedocs/gcc/Inline.html>"
  - Hacer los casts necesarios, no asumir que se hacen automaticamente y correctamente siempre. Además de eliminar posibles fuentes de errores mejoras la portabilidad puesto que hay compiladores más "exigentes" con los casts y los tratan como error si no son explicitamente declarados.
  - Usar lo mínimo de C, no usar extensiones aunque sean del estandar (las de C98 no se han implementado en más sitios de los que parece).
  - Declarar las variables al estilo C, es decir, al principio de la función o bloque con llaves, no en medio del código (esto es un funcionalidad C++).

## Optimización

Pese a que soy partidario de optimizar al final, y solo si es necesario, si que hay algunas recomendaciones que podemos seguir mientras programamos y de paso, sin esfuerzo adicional, estamos optimizando.

### Acceso a estructuras anidadas en bucles:

- Acceder a estructuras con cierto nivel de indirección es pesado, se puede mejorar haciendo una sola vez la indirección para todo el bucle y ahorras muchos accesos a memoria que penalizan, ejemplo:

```
// Ejemplo real de código no optimo
for( int i = 0; i < cardData.iNumMov; i++ )
{
    ReportRow row;
    QString aux;

    // Observa cuantas indirecciones para acceder a la estructura de información de tarjeta.
    row.push_back(QLatin1String(cardData.pMov[i].pszTypeMovement));
    row.push_back(QLatin1String(cardData.pMov[i].pszDate));
    row.push_back(QLatin1String(cardData.pMov[i].pszLine));
    row.push_back(QLatin1String(cardData.pMov[i].pszRoute));
    row.push_back(QLatin1String(cardData.pMov[i].pszTravel));
    amount = cardData.pMov[i].iInitialBalance - cardData.pMov[i].iFinalBalance;
    row.push_back(aux.setNum(amount));
    row.push_back(aux.setNum(cardData.pMov[i].iFinalBalance));

    report.push_back(row);
}
// Mismo ejemplo reescrito
for( int i = 0; i < cardData.iNumMov; i++ )
{
    ReportRow row;
    QString aux;
    tCheckTitMov titMov = cardData.pMov[i]; // Hacemos una sola vez la indirección.

    row.push_back(QLatin1String(titMov.pszTypeMovement));
    row.push_back(QLatin1String(titMov.pszDate));
    row.push_back(QLatin1String(titMov.pszLine));
    row.push_back(QLatin1String(titMov.pszRoute));
    row.push_back(QLatin1String(titMov.pszTravel));
    amount = titMov.iInitialBalance - titMov.iFinalBalance;
    row.push_back(aux.setNum(amount));
    row.push_back(aux.setNum(titMov.iFinalBalance));
}
```

```
report.push_back(row);
}
```

## Aplicaciones multithread

### Runtime de C traicionera

Ojo con las funciones de la runtime de C.

Hay bastantes de ellas que no son multithread, ejemplo localtime y sus amigas. Estas devuelven una estructura global compartida para todos los threads de la runtime, resultado: "Que el último thread que la llama es el que deja la fecha impresa en la estructura".

Es aconsejable revisar las páginas man y ver si son multithread, y si no es así buscar una equivalente como por ejemplo localtime\_r.

Hay otras que tienen condiciones para funcionar correctamente, por ejemplo memcpy requiere que las zonas de memoria origen y destino no se solapen, en ese caso hay que usar memmove.

### Reentrancia y seguridad ante hilos (reentrance and thread safety)

Son dos terminos referentes a como las funciones gestionan sus recursos. Sin embargo son conceptos distintos, una función puede ser reentrante, thread-safe, ambas cosas o ninguna.

#### Reentrancia

Una función es reentrante si no mantiene datos estáticos (variables globales o variables static) durante sucesivas llamadas y si no devuelve punteros a datos estáticos. Todos los datos son suministrados por el usuario de la función y además esta función no llama a otras funciones no reentrantes.

Ejemplos de rutinas no reentrantes son strtok o ctime puesto que manejan datos estáticos.

#### Thread safety

Una función thread-safe es aquella que protege sus recursos contra accesos concurrentes mediante alguna primitiva de sincronización como un mutex, una sección crítica, semáforo, etc.

En lenguaje C las variables locales se crean en la pila, por lo tanto, una función que solo utiliza variables locales y que no usa recursos compartidos es thread-safe automáticamente, ejemplo:

```
/* función automaticamente thread-safe,
por muchos threads que entren no tendrán problemas,
cada uno tendrá su propia copia de las variables
locales */
int suma(int a, int b)
{
    int resultado;

    resultado = a + b;
    return resultado;
}
```

Ahora bien, la cosa cambia cuando usamos recursos compartidos, en este caso hay que protegerlos con primitivas de sincronización.

El concepto de reentrante es "más fuerte" que el de thread-safety porque la reentrancia permite que la función pueda ser llamada ¡incluso desde el mismo thread!, consideremos el siguiente ejemplo que encontrado en internet (es un poco rebuscado puesto que es complejo escribir una función thread-safe y no reentrante):

```

int length = 0;
char *s = NULL;

// Note: Since strings end with a 0, if we want to
// add a 0, we encode it as "\0", and encode a
// backslash as "\\".

// WARNING! This code is buggy - do not use!

void AddToString(int ch)
{
    EnterCriticalSection(&someCriticalSection);
    // +1 for the character we're about to add
    // +1 for the null terminator
    char *newString = realloc(s, (length+1) * sizeof(char));
    if (newString) {
        if (ch == '\0' || ch == '\\') {
            AddToString('\\'); // escape prefix <--- ¡OJO!, aquí se llama de nuevo a la función ¡desde el mismo thread!
        }
        newString[length++] = ch;
        newString[length] = '\0';
        s = newString;
    }
    LeaveCriticalSection(&someCriticalSection);
}

```

Esta función es thread-safe porque la sección crítica previene que dos threads intenten usarla simultáneamente. Sin embargo, no es reentrante.

La llamada interna a AddToString ocurre mientras las estructuras de datos no son estables. Al llamarse a si misma la re-entrada intentará hacer un "realloc" pero utilizará un puntero (s) que ya no es válido (se invalidó en la primera llamada a realloc).

### Escribir funciones listas para multi-threading

Una buena costumbre es dejar listas tus funciones para hacerlas multithread con el menos esfuerzo posible, es posible que ahora no haya que hacerla multithread pero si en el futuro, con un mínimo esfuerzo el cambio luego será más fácil:

- Evita los múltiples puntos de salida de la función, si puedes hacerlo en uno solo mejor.
- Evita en la medida de lo posible los break en los bucles
- Evita el goto en la medida de lo posible

Ejemplo:

```

/* Esta función usa recursos compartidos globales.
Devuelve 0 si todo ha ido bien y -1 si error */
int funcion_dummy(int param)
{
    int result = 0;

    if( READY == check_recurso_global() ){
        if( param > UPPER_LIMIT ) {
            result = -1;
        } else if (param < LOWER_LIMIT ) {
            result = -1;
        } else {
            actuar_sobre(recurso_global,param);
        }
    } else {
        result = -1;
    }
}

```

```
    return result;
}
```

Supongamos ahora que queremos hacerla thread-safe:

```
/* Esta función usa recursos compartidos globales.
Devuelve 0 si todo ha ido bien y -1 si error */
int funcion_dummy(int param)
{
    int result = 0;

    MUTEXLOCK(&mutex);
    if( READY == check_recurso_global() ){
        if( param > UPPER_LIMIT ) {
            result = -1;
        } else if (param < LOWER_LIMIT ) {
            result = -1;
        } else {
            actuar_sobre(recurso_global,param);
        }
    } else {
        result = -1;
    }
    MUTEXUNLOCK(&mutex);
    return result;
}
```

\*¡Tan solo hemos añadido dos líneas!\*

Imaginemos que la hubiésemos escrito de otra forma al principio:

```
/* Esta función usa recursos compartidos globales.
Devuelve 0 si todo ha ido bien y -1 si error */
int funcion_dummy(int param)
{
    if( READY != check_recurso_global() )
        return -1;

    if( param > UPPER_LIMIT ){
        return -1;
    } else if (param < LOWER_LIMIT ) {
        return -1;
    } else {
        actuar_sobre(recurso_global,param);
    }

    return 0;
}
```

Al hacerla ahora reentrante:

```
/* Esta función usa recursos compartidos globales.
Devuelve 0 si todo ha ido bien y -1 si error */
int funcion_dummy(int param)
{

    MUTEXLOCK(&mutex);
```

```

if( READY != check_recurso_global() ){
    MUTEXUNLOCK(&mutex); // 1 linea más
    return -1;
}

if( param > UPPER_LIMIT ){
    MUTEXUNLOCK(&mutex); // 1 linea más
    return -1;
} else if (param < LOWER_LIMIT ) {
    MUTEXUNLOCK(&mutex); // 1 linea más
    return -1;
} else {
    actuar_sobre(recurso_global,param);
}
MUTEXUNLOCK(&mutex);
return 0;
}

```

Tres líneas más que con la anterior forma, como se puede apreciar es más complejo adaptar este tipo de código.

## Herramientas

### Lex & Yacc

Tras pruebas intensivas hemos detectado dos problemas relacionados con el parser generado.

1. Reserva mucha memoria en el stack (para la pila de reducción-desplazamiento del parser), y eso no es aceptable para este embebido en particular (ARM7 sin MMU). Es posible ajustar la cantidad de memoria de la pila pero en cualquier caso sigue siendo bastante más de 4K. (ronda los 18K).
2. Se hacen mallocs, frees y reallocs al proporcionarle entradas desde cadenas de memoria, he cambiado las rutinas de entrada desde cadenas (YY\_INPUT) para que no se use memoria dinámica.

A pesar de estas modificaciones seguimos teniendo dudas sobre la estabilidad del parser en un embebido sin MMU por lo que el futuro de este es incierto. A día de hoy se ha reescrito otro parser a mano para sustituir a este hasta resolver por completo todos los interrogantes.

Seguiré investigando porque tengo constancia de su utilización en entornos embebidos. Tal vez mi reducida experiencia con estas herramientas sea el problema y tras unas pocas ajustes de configuración pueda generar un parser adecuado a este embebido.

Este link <http://www.embedded.com/story/OEG20030220S0036> trata sobre lex & yacc en sistemas embebidos, tal vez me aclare conceptos.

## Sobre el uso de memoria dinámica

Personalmente no tengo nada en contra del uso de memoria dinámica (malloc, calloc, realloc, etc...), es más, considero que dota a los programas de mayor flexibilidad puesto que consumimos solo lo necesario y nos evitamos tener que marcar límites arbitrarios para las operaciones que requieran del uso de memoria (por ejemplo usar #defines para establecer tamaños máximos de buffers).

Ahora bien, hay dos excepciones en las que se desaconseja su uso:

1. En máquinas sin MMU (<http://es.wikipedia.org/wiki/MMU>)
2. En operaciones que se repiten muchas veces durante el funcionamiento del programa

La explicación es la siguiente:

En el primer caso se producirá fragmentación de la memoria (totalmente inevitable) y acabemos por quedarnos sin memoria disponible.

En el segundo caso, la constante reserva y liberación de memoria, además de ser menos eficiente, puede llevar a que un bug en esa parte del código nos cause fugas de memoria que puedan "tumbar" la aplicación.

## Notación a utilizar para la programación

Tras haber probado varios estilos (notación húngara, GNU, etc...) me decantado por lo siguiente:

La idea es hacer un código sencillo y fácil de entender para todos.

Todos los programadores del proyecto deben mantenerse consistentes con este estilo.

- Estilo K&R para indentación, llaves y demas. [http://en.wikipedia.org/wiki/Indent\\_style](http://en.wikipedia.org/wiki/Indent_style)
- Usamos tabuladores, no espacios para indentar.
- Nos limitamos a 80 caracteres por línea.
  - Es el standard de facto y muchos editores lo utilizan.
  - No uses, por este motivo, cajetines con simbolos puesto que pueden no verse bien en todos los editores

```
/******  
***/  
/* Sorting algorithm: based on qsort  
*/  
/******  
***/  
  
Mejor usamos:  
  
/*  
 * Sorting algorithm: based on qsort  
*/
```

- Usamos comentarios C, no comentarios CPP. No todos los compiladores soportan comentarios CPP. ANSI C no admite el comentario de línea "///  
• Usamos espacios para separar los componentes de una expresión, comparad:

```
for (wn++; wn < 100; field[wn+\+] = '\0');  
  
for(wn++;wn<100;field='\0');
```

- Para las variables
  - Cuanto menor vida tienen (son locales o globales) más corto es el nombre
  - Cuanto más importante y en más lugares se usa, más largo es el nombre
- Para las funciones
  - Deben corresponderse, a ser posible, con un verbo. De esta forma son autodestructivas:

```
string_is_int("1234") /* Claramente sabemos que devuelve un true o false */  
CheckString("1234") /* No sabemos que devuelve o que hace. ¿Que hace CHECK? */
```

- No usamos capitalización (Camel Case) ni notación húngara. Usamos minúsculas y las separamos con subrayados

```
if (next_entry == maximum_value)
```

- Para evitar una sobreutilización del subrayado, cuando la variable es muy corta obviamos el subrayado.

```
str[i] = toupper(ch);
```

- Si puedes declararlas en una línea, ¡hazlo!. En caso contrario, antes de pasarte de los 80 caracteres usa varias líneas.

```
static int dispatch_instruction (Instruction *i,  
long pc, Process *p)  
{ ... }
```

- Comentarios
  - Comenta la cabecera de cada función para dejar claro como usarla y que ofrece
  - Dentro de la función límitate a solo comentar cosas fuera de lo habitual, deja que el código hable por si mismo siempre que sea posible.
- Defines y enums
  - Van siempre en MAYUSCULA:

```
enum {  
    MAX_SYMBOLS = 1023,  
    BLOCK_SIZE = 4096  
};  
#define WHITE 0x00FFFFFF
```

- Definiciones de nuevos tipos
  - En minúsculas y finalizados en "\_t"

```
typedef struct _tipo_nuevo_t  
{  
    int campo1;  
    char* campo2;  
    float campo3;  
}tipo_nuevo_t;
```