COMENTARIO TECNICO

Optimice su código....
....Pero sin Ofuscarlo!!

Por Ing. Eduardo A. Martínez[i] - Ing. Marcelo E. Romeo[ii]

Segunda Parte.

1.1. En lo posible, no use aritmética de punto flotante.

Si bien el lenguaje permite trabajar en punto flotante a través de los tipos *float*, *double* y *long double* no es recomendable usarlo, por varias razones, entre ellas:

- La unidad aritmético-lógica de la mayor parte de los procesadores que se usan para propósitos específicos no poseen la capacidad de cálculo en punto flotante, por lo cual deben recurrir a uso de funciones de biblioteca muy extensas en código y en tiempo de ejecución o a coprocesadores, con el aumento de costo y de tiempo de ejecución.
- La aritmética de punto flotante es solamente útil cuando hay que mantener una precisión determinada en valores que difieren en varios órdenes de magnitud entre ellas, lo cual no es tan usual.
- La comparación en punto flotante es más bien complicada: determinar que dos números de punto flotante son iguales entre sí (o totalmente desiguales), necesitan de la comparación contra un valor de guarda.

Es muy común encontrar programas que usan punto flotante innecesariamente, como por ejemplo en el caso de manejo de medios de pago; mucha gente piensa que porque un valor esté expresado en fracciones de entero o con decimales, debe obligatoriamente usar punto flotante.

Supongamos, por ejemplo, que estamos realizando el programa que maneja una caja registradora de supermercado: obviamente, el resultado de la venta va en pesos con sus respectivos centavos. Ahora bien, *todos los precios tienen la misma característica*, con lo cual no estamos manejando magnitudes de órdenes distintos (que es lo único que justificaría el uso de punto flotante). Puede, por lo tanto, usarse una *representación interna* que no sea necesariamente la *externa* y colocar todos los montos en centavos en lugar de en pesos.

Si se considera que un *long* signado posee una capacidad de representación de aproximadamente dos mil millones y representamos con ese tipo de variable la venta total de una caja registradora en el día de trabajo en centavos, ello significaría en pesos un valor de veinte millones, más que suficiente para la representación buscada.

Una vez que los cálculos han sido realizados internamente en representación *long*, no existe inconveniente en mostrar los resultados en la representación externa, ya que es fácil convertir a texto con la coma decimal para mostrarlo en el *display* de cajero o imprimirlo en el *ticket*.

1.2. No use variables intermedias o temporales para cambiar la representación.

Continuando con el ejemplo de la caja registradora, supongamos que el precio unitario de los productos de un supermercado debe ser mantenido en un archivo o arreglo dentro de la caja registradora; es usual que un supermercado medio tenga una base de productos posibles cercanos a los 20000, con lo cual debería elegirse la representación más pequeña disponible para dicho precio unitario.

Consideremos el caso de un *unsigned short*: la capacidad de representación será de 0 a 65535 (ya que es de 16 bits); si representa el precio de cada producto en centavos, tendríamos una capacidad de representación en pesos de 655, lo cual puede ser aceptable.
Consideremos, además, que guardaremos el precio neto (sin IVA) por lo cual frente a una venta, deberemos calcular el precio final con IVA.

Obviamente, supondremos que la capacidad de representación de 16 bits alcanza para el precio final; sin embargo, deberemos hacer un cálculo intermedio *sin perder precisión* que involucra la multiplicación por un factor.

Alguien propone el siguiente trozo de código, suponiendo que el número de producto está en *num_prod*:

```
#define NUM_PRODS       20000
#define IVA_VAL    21UL

unsigned short prices[NUM_PRODS], net_price;
unsigned final_price;
unsigned long iva;
int num_prod;

        net_price = prices[ num_prod ];
        iva = IVA_VAL * net_price;
        final_price = net_price + (unsigned short)(iva/100);
```

Obviamente, el resultado va a ser correcto; sin embargo, nos obligó a utilizar una variable intermedia y a repartir el cálculo en varias sentencias del lenguaje, perdiendo, de esa manera, la posibilidad de reutilizar los resultados que quedan en los registros de trabajo del procesador.

Si proponemos el siguiente código, vamos a obtener el mismo resultado, con una mayor optimización por utilizar las construcciones del mismo lenguaje y, posiblemente, una mayor claridad.

```
#define NUM_PRODS       20000
#define IVA             12UL

unsigned short prices[NUM_PRODS];
unsigned final price;

final_price = prices[num_prod];
final_price += (unsigned short)( ( IVA_VAL * final_price )/ 100 );
```

Como conclusión, utilice la capacidad del compilador de cambiar la representación mediante el operador de *promoción forzada*[iii] para no necesitar variables intermedias y poder resolver sus cálculos en una sola expresión[iv].

1.3. Tipos de variables en implementaciones pequeñas.

Hoy en día existen compiladores de C para procesadores relativamente pequeños, como es el caso de microcomputadoras de 8 bits con capacidades de programa y de memoria RAM más bien reducidas[v].

Para muchos procesadores de 8 bits, las operaciones a 16 bits no son muy directas de implementar y necesitan varios ciclos de ejecución para lograrlas; de hecho, la ALU es de 8 bits y las operaciones aritméticas y lógicas se realizan eficientemente en 8 bits.

Los compiladores que se realizan para esos procesadores generalmente asignan 16 bits para el tipo *int* y *permiten evaluaciones en 8 bits*, en contraposición a lo establecido por la norma ANSI y a lo mostrado en la Tabla 1.

Ello, más que una desventaja, se puede usar como ventaja para optimizar el código, sin perder la generalidad que debe implicar un buen programa; recuerde siempre que, si bien Ud. debe aprovechar las ventajas de la actual implementación, debe ser suficientemente transparente a las mismas de manera de poder migrar rápidamente y sin mayores sufrimientos a otras implementaciones.

Es por ello, que los autores, en distintos proyectos, han mantenido un archivo de inclusión denominado *cutils.h* que permite tener definiciones generales útiles para todo proyecto, así como definiciones de tipo que permiten aprovechar las características de las implementaciones para 8 bits sin perder la generalidad necesaria.

A continuación, se muestra parte de dicho archivo:

```
#ifndef __CUTILS_H__
#define __CUTILS_H__

#define forever       for(;;)

typedef unsigned char       uchar;
typedef signed char         schar;
typedef unsigned int uint;
typedef unsigned long       ulong;
```

```
#ifdef __TINY_PROC__
typedef uchar                MUint
typedef schar        Mint;
#else
typedef uint         MUint;
typedef int          Mint;
#endif

#endif
```

La definición de *forever* es para mostrar que en este archivo se colocan definiciones que son útiles para cualquier programa en C (las otras definiciones se han extraído para mantener corto y claro el texto).

Como colocar ciertas definiciones de tipo son más bien tediosas por la cantidad de texto que involucran, existen cuatro *typedef* para la redefinición de tipos usuales.

Lo más importante de este archivo, para las finalidades de este artículo, son aquellas que se seleccionan por la existencia o no de la definición de la variable del preprocesador __TINY_PROC__.En efecto, si no está definida dicha variable, los tipos MUint y Mint se hacen respectivamente iguales a *uint* y a *int*; si está definida, se hacen respectivamente igual a *uchar* y *schar*.

Para ver la utilidad de estas definiciones, vamos a considerar el siguiente sencillo módulo de demostración:

```
/*
 *    Use of square function
 */

#include “cutils.h”

static Mint
square( Mint val )
{
        return val * val;
}

int
main( void )
{
        Mint i, a;

        for( i = 0 ; i < 4 ; ++i )
                a = square( i );

        return 0;
}
```

Consideremos dos casos:

1.3.1. __TINY_PROC__ no está definida

Si la variable __TINY_PROC__ no está definida, lo que va a ver el compilador como código será:

```
static int
square( int val )
{
        return val * val;
}

int
main( void )
{
        int i, a;

        for( i = 0 ; i < 4 ; ++i )
                a = square( i );

        return 0;
}
```

Este código, entonces, es el normal que está de acuerdo a las consignas de la norma ANSI C sobre evaluación de tipos de variables.

1.3.2. __TINY_PROC__ está definida

En este caso, el código que verá el compilador será:

```
static signed char
square( signed char val )
{
        return val * val;
}

int
main( void )
{
        signed char i, a;

        for( i = 0 ; i < 4 ; ++i )
                a = square( i );

        return 0;
}
```

El cual aprovecha la característica de los procesadores de 8 bits de ser más eficiente desde el punto de vista deevaluación aritmética en 8 bits que en 16 bits.

Se puede demostrar que, en general, ésta es una de las mejores optimizaciones en código para procesadores de 8 bits; sin embargo, deben tenerse en cuenta estas restricciones:

- Los tipos *Mint* y *MUint* sólo deben utilizarse para la definición de variables automáticas de trabajo, de argumentos de funciones y de retornos de funciones siempre y cuando los valores a alojar en ellas estén de acuerdo con lo que soportan los tipos *uchar* y *schar*;
- De ninguna manera, deben ser utilizados para la declaración de almacenamiento, especialmente de arreglos o de miembros de estructuras; en esos casos, sólo deben utilizarse los tipos de C directamente

La ventaja fundamental de esta forma de proceder es que se es absolutamente transparente para la compilación en distintas plataformas y que sólo debe definirse o no la variable del preprocesador __TINY_PROC__, lo cual puede hacerse perfectamente en el comando de compilación mismo.

1.4. Almacenamiento estático contra argumentos de funciones.

Un de las ventajas más potentes de un lenguaje de alto nivel como C es la posibilidad de escribir funciones que no estén especializadas sobre variables fijas, sino que operen sobre argumentos que se le pasan en el momento de invocación.

Para ello consideremos un sencillo manejador de la estructura de dato pila implementado en un módulo de compilación separado denominado *stack.c* que opera sobre tipos de datos *long*:

```
/*
 *      stack.c
 */

void
clear_stack( long *pstack, long **psp )
{
        *psp = pstack;
}

int
push_stack( long *pstack, long **psp, int max, long val )
{
        if( *psp < pstack + max )
        {
                *(*psp)++ = val;
                return OK;
        } else
                return ERROR;
}
```

```
int
pop_stack( const long *pstack, long **psp, long *pval )
{
        if( *psp == pstack )
                return EMPTY;
        *pval = *--(*psp);
        return OK;
}
```

Como se puede observar, estas funciones de manejo de pila tienen la suficiente generalidad como para manejar varias pilas simultáneamente, sin estar especializadas a una pila en particular.

Sin embargo, el precio que se paga por ello es que las funciones llevan varios argumentos y como los argumentos deben ser pasados a través del *stack* de la máquina, ello significa una cantidad de código no despreciable para que el programa invocador pase los argumentos y para que la función invocada rescate esos argumentos, con la consiguiente menor velocidad de ejecución.

Si el caso en cuestión es sólo para mantener una sola pila, se puede especializar haciendo que las estructuras de datos de la pila y de su puntero al *stack* se encuentren como variables estáticas dentro del mismo módulo que implementa la pila.

```
/*
 *      stack.c
 */

#define NUM_ENTRIES    50

static long stack[ NUM_ENTRIES ], *sp;

void
clear_stack( void )
{
        sp = stack;
}

int
push_stack( long val )
{
        if( sp < stack + NUM_ENTRIES )
        {
                *sp++ = val;
                return OK;
        } else
                return ERROR;
}

int
pop_stack( long *pval )
```

```
{
        if( sp <= stack )
                return EMPTY;
        *pval = *--sp;
        return OK;
}
```

Si bien este es un recurso que permite una buena optimización de programas, se debe recordar que las funciones quedan especializadas a estructuras de datos específicas y que, por ende, pierden generalidad; sin embargo, es muy común en desarrollos tener que buscar una solución de compromiso entre optimización de código y generalidad; decídase por la primera cuando la segunda Ud. haya establecido *con seguridad* que no es ni va a ser necesaria.

1.5. Optimizaciones del compilador

Actualmente, los compiladores tienen pasadas de optimización cuando han creado el código objeto, con la finalidad de evitar instrucciones superfluas o sin acción; si bien este trozo de código es más bien simplista, sin embargo el optimizador del compilador es capaz de encontrarlo y de anular las instrucciones innecesarias:

```
if( some_condition_met )
        variable = some_constant;
else
        variable = some_constant;
```

donde las sentencias del *if* y del *else* son *exactamente iguales*.

El optimizador más sencillo del compilador cambiará las cuatro sentencias en el objeto que dejará el código para ejecutar sencillamente:

```
variable = some_constant;
```

Obviamente, hoy en día los optimizadores de los compiladores son mucho más sofisticados que lo que se evidenció en este simple ejemplo; sin embargo, no existe un *standard* de los pasos de optimización de un compilador y, además, cuanto más sofisticado es el optimizador, más propenso a errores es el compilador; la mayor parte de las fallas de un compilador se encuentran en sus optimizadores.

Por lo tanto, se aconseja estudiar detenidamente las opciones de optimización de su compilador, probar su eficiencia y quedarse con aquellas opciones que sean capaces de mejorar la eficiencia de su código sin llegar a extremos inútiles y peligrosos.

1.6. Conclusiones

Este artículo solamente ha logrado rasguñar la superficie del problema de optimización; se han mostrado algunos casos que no se pretende que involucren todos los casos posibles.

Se han mostrado técnicas de optimización que no cambian la legibilidad del código y que aprovechan las definiciones del lenguaje y tratan de comprender cómo un compilador puede interpretar las sentencias que escribimos en un lenguaje de alto nivel como C.

El lenguaje C tiene la ventaja que una misma acción puede ser realizada de muchas formas distintas y, por lo tanto, es importante que Ud. comprenda qué implica cada una de las formas, de manera de aprovechar la que mejor convenga a las circunstancias y ambiente de su programa.

Todo compilador tiene una forma de requerirle que entregue el código *assembler* traducido sin optimización; hágalo para ciertas estructuras de control de lenguaje y comprenda íntimamente qué significa la escritura de una simple sentencia del lenguaje; esa comprensión permitirá que Ud. escriba código legible, mantenible y eficiente en un lenguaje más cómodo que *assembler*.

El esfuerzo bien vale la retribución que se obtendrá.

Fin.....

Nota de Redacción: El lector puede descargar este artículo y artículos anteriores desde la sección ***"Artículos Técnicos"*** en el sitio web de **EduDevices (www.edudevices.com.ar)**

Referencias:

---

[i] Universidad de Belgrano
[ii] Universidad de Belgrano – Universidad de San Martín – UTN - FRBA
[iii] *cast*
[iv] Siempre y cuando esta única expresión siga siendo de lectura clara. la exageración de esta propuesta lleva nuevamente a código difícil de leer (a los programadores que abusan de ella se los llama *one_liners*);
[v] Los autores han desarrollado programas enteramente en C para una microcomputadora de 8 bit con 32 KBytes de flash y 1 Kbyte de RAM