COMENTARIO TECNICO

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

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

Primera Parte.

1. Antecedentes

Mucho se ha discutido en la literatura sobre la conveniencia o no de usar un lenguaje de alto nivel para la programación de microprocesadores y microcomputadoras en contraposición al uso de *assembler*.

Esta discusión no puede generalizarse para cualquier lenguaje de alto nivel y nosotros sólo la realizaremos para el caso del lenguaje C.

Los que son partidarios del uso de *assembler* enuncian:

- Los lenguajes de alto nivel desperdician espacio de código y de variables, lo cual lo hacen inadecuado para procesadores con pequeños recursos, especialmente los de 8 bits.
- Es imposible el manejo de *hardware* sin utilizar *assembler*.
- La utilización de un lenguaje de alto nivel impide que los códigos en relación con el *hardware* se realicen a la velocidad adecuada que el *hardware* exige.

Independientemente de la veracidad de los argumentos anteriores, los proyectos de desarrollo con microprocesadores y microcomputadoras han aumentado las exigencias en cuanto a complejidad y hacen que la programación en *assembler* esté en contra de la claridad, mantenibilidad y modificación; la modificación está íntimamente vinculada a la vida de un producto, el cual debe ir adecuándose a las necesidades de distintos usuarios y mercados.

Por otro lado, es usual que el producto tenga un tiempo de vida que, usualmente, es mayor que la tecnología bajo la cual se realizó y, por ejemplo, un producto hecho con una familia de microcomputadoras de 8 bits es probable que deba migrarse a una de 16 bits o, mejor aún, de 32 bits; si ese es el caso, entonces es probable que, si el desarrollo está realizado en *assembler*, deba empezarse nuevamente de cero, con los costos directos y de oportunidad que ello significa.

Por lo tanto, es conveniente tratar de realizar los proyectos casi totalmente en lenguaje C, tratando que aquellas partes del mismo que necesariamente se deben escribir en *assembler* sean muy reducidas.

Después de todo, el lenguaje C fue creado por la *Bell Laboratories* con un objetivo parecido en mente: luego de escribir tres veces el sistema operativo Unix, para distintas arquitecturas, totalmente en *assembler,* se llegó a la conclusión que aquellas partes del Sistema Operativo ligadas directamente al *hardware* eran menores al 5% de todo el Sistema Operativo y que, por lo tanto, no valía la pena escribirlo en *assembler*; por ello, desarrollaron *especialmente* el lenguaje C que debía tener la misma capacidad de expresión que el *assembler* y debía tener una eficiencia cercana a la escritura en *assembler*[iii].

Más aún, por la misma razón se creó la estructura del lenguaje para que pueda ser *independiente de la plataforma* y, de esa forma, poder adaptar el Sistema Operativo a distintas plataformas sin necesidad de cambiar la mayor parte del código del mismo.

2. Optimización

Muchas veces, se mal interpreta el término *optimización*; en efecto, muchos programadores tratan de *optimizar* escribiendo código apretado y de difícil lectura desde el comienzo del proyecto, lo cual no es aconsejable de manera alguna.

El código debe ser lo más claro posible, ya que el código no sólo se escribe para que, una vez traducido por el compilador, se ejecute correctamente respecto de lo especificado sino también para que sea legible para cualquier programador razonable que conozca el lenguaje; debe pensarse que el código que se escribe en un proyecto debe ser leído por otro programador (o por Ud. mismo) luego de cinco años y ser comprendido rápidamente.

Por lo tanto, *optimizar ciegamente* no es un objetivo en sí mismo; después de todo, es más importante lograr que un programa complejo funcione adecuadamente y, si es necesario optimizar alguna parte de él, debe determinarse experimentalmente y se encontrará que las zonas de programa a optimizar son pocas y de poco código.

La optimización a que nos referiremos en este artículo es aquella que puede ser realizada en forma genérica para cualquier plataforma, que no oscurece el código y que se basa más bien en el comportamiento del compilador bajo ciertas construcciones del lenguaje, respetando el *standard* del mismo.

2.1. Sentencias de autoincremento y autodecremento

Es muy común en los programas el incremento o el decremento de una variable en el valor 1; si bien la mayoría de los lenguajes lo permiten hacer mediante la operación suma o resta como en el caso siguiente[iv]:

```
c = c + 1;
i = i – 1;
```

Sin embargo, el compilador es probable que utilice un código que esté preparado para una suma o resta genérica de una variable respecto a una constante genérica; ello generará un código más voluminoso y más lento que si el compilador supiese de antemano que la instrucción se refiere al valor 1; es por ello, que el lenguaje C acepta operadores de auto incremento y de auto decremento como en los siguientes casos:

```
c++;
i--;
```

Un problema con este tipo de sentencia es que el lenguaje C la considera, a su vez, una expresión que se evalúa, como por ejemplo:

```
i = 10;
while( i-- )
   process( i );
```

La evaluación de i-- da como resultado para la evaluación del *while* el valor de i antes de decrementar en 1, si bien después de terminar la evaluación, efectivamente i queda decrementado en 1; como consecuencia de ello, el lazo se realizará 10 veces.

Muchas veces, estas instrucciones de auto incremento y de auto decremento se escriben en sentencias donde la evaluación de las mismas no es utilizada, como en el siguiente trozo de código:

```
for( i = 0; s[i] != '\0'; i++ )
        process( s[i] );
```

En este caso, expresar el incremento como i++ ó ++i es exactamente lo mismo de acuerdo al resultado esperado, ¿qué opción se tomaría?

Debemos pensar, entonces, ¿cómo haría el compilador para traducir estas sentencias[v]?

Si suponemos que la variable a incrementar o decrementar se encuentra en memoria, una secuencia posible en un *assembler* genérico sería:

```
LD    register,(memory)
INC   register
STO   (memory),register
```

quedando el valor incrementado en el registro; ahora bien, si lo que se pretende utilizar como resultado de la expresión del auto incremento es el valor luego de incrementar (caso de ++i), ya estaría en el registro y se podría seguir operando con él; si el caso es que la evaluación es el de i++, entonces el compilador debería agregar una instrucción de decremento de registro para poder utilizar el valor:

```
DEC   register
```

Por lo tanto, si no se va a utilizar el resultado de la evaluación de auto incremento o auto decremento y sólo se utilizará su capacidad de incrementar o decrementar una variable en el valor 1, siempre es mejor utilizar las operaciones de preincremento y predecremento.

2.2. Tipos de variables

Quien se acerca por primera vez al lenguaje C, queda un poco despistado por la gran cantidad de tipos de variables, especialmente en lo que hace a variables enteras; en efecto, se pueden definir los siguientes tipos de variables enteras:

Tipo	Tamaño	Evaluación
char	8 bits	No existe
short	16 bits	No existe
int	16 o 32 bits	Existe
long	32 bits	Existe

Tabla 1

Los tamaños son los especificados por la norma ANSI[vi], donde se ve que el único tipo que no tiene definido *a priori* el tamaño es el caso del *int*. De acuerdo a lo que dice el libro de Kernighan y Ritchie, *int es el tipo natural de la máquina*; de manera de hacer esta discusión simple, encontraremos que en los procesadores más pequeños (por ejemplo, típicamente los procesadores de 8 bits) *int* es de 16 bits y, en los otros, *int* es de 32 bits.

De acuerdo a la norma ANSI, no en todo tipo de variables enteras existe capacidad de evaluación, como se puede apreciar en el cuadro anterior: sólo en los tipos *int* y *long* existe capacidad de evaluación; de esta forma, si en una expresión existen variables *char* o *short*, el valor de ellas será promovido a *int* automáticamente antes de proceder a realizar la evaluación.

Como resultado, se llega a la conclusión que los tipos *char* y *short* deben ser utilizados para definición de almacenamiento en memoria *exclusivamente*.

Todos lo cálculos aritméticos en este tipo de variables se realizan internamente en *complemento a 2*, ya que, salvo por el caso particular del *char*, por omisión[vii] las variables enteras se consideran *signadas*; si se pretende que se consideren sin signo, debe anteponerse el prefijo *unsigned*.

El caso del *char* es especial; por razones históricas, la norma no define explícitamente si el *char* es signado o no. De hecho, los programas no deberían presuponer nada respecto del signo del *char* y comportarse correctamente, independientemente que el compilador la considere signada o no signada.

Si realmente se pretende establecer que un *char* sea signado o no, debe anteponerse los prefijos *signed* o *unsigned* explícitamente[viii]; lo interesante de este caso, además, es que entonces *char*, *signed char* y *unsigned char* son ***tres tipos distintos de variables***. En el cuadro siguiente se muestran, entonces, todos los tipos reconocibles por el compilador y considerados distintos por el mismo.

Tipo	Tamaño	Signo
char	8 bits	No determinado
unsigned char	8 bits	No signado
signed char	8 bits	Signado
short	16 bits	Signado
unsigned short	16 bits	No signado
int	16 o 32 bits	Signado
unsigned int	16 o 32 bits	No signado
long	32 bits	Signado
unsigned long	32 bits	No signado

Tabla 2

Por último, debe comprenderse qué ocurre frente a la evaluación donde figuran tipos distintos, como en el caso:

```
char c;
unsigned char c1;
long el;

    c = c1 * el;
```

Todo *char* o *short* se debe convertir su valor a *int*; por lo tanto, el valor de *c1* se promoverá a *int* signado (ya que un *char no signado* no pierde capacidad de representación en un *int signado*). Como *el* es de tipo *long signado*, a su vez el valor de *c1* se convierte a *long signado*, la multiplicación se realiza con dos operandos *long signados* y el resultado es *long signado*. Ahora bien, ese valor se debe colocar en un lugar cuya capacidad no es suficiente para almacenar el resultado de un *long signado* (sin modificar la representación), con lo cual se *forzará* el resultado en 8 bits dejando el resto de la representación en 8 bits sin signo (si el *char* en esa implementación es *no signado* o en 7 bits con signo (si el *char* en esa implementación es con signo)[ix].

A esta altura, la pregunta que queda por dilucidar es para qué se usa el tipo *int* si no está determinado su tamaño. Debemos recordar lo que dicen Kernighan y Ritchie: es el *tipo natural de la máquina*.

¿Qué debe interpretarse por el *tipo natural*? Es aquel tipo para el cual la unidad aritmética-lógica del procesador tiene implementadas la mayor parte de las operaciones aritméticas y lógicas y, por lo tanto, es el tipo para el cual esas operaciones son más eficientes[x].

Por ello, es que el tipo *int* se utiliza para las operaciones más usuales, cuales son, por ejemplo, contadores para los ciclos *for* y pasaje de parámetros a funciones y valores de retorno de las mismas funciones[xi].

Por ejemplo, en el siguiente ciclo *for*:

```
int i, a;

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

Es obvio que se podría haber usado como contador un *char*, pues los valores a almacenar en el mismo van de 0 a 3; sin embargo, no existe evaluación en *char* y, por lo tanto, antes de la comparación con 4, debería promocionarse el valor de i a *int*, lo cual significaría una pérdida de eficiencia en el código generado por el compilador y, por ende también, en el tiempo de ejecución.

De la misma manera, si suponemos que la función *square* eleva al cuadrado el argumento, tanto el argumento (por las mismas razones anteriores) como el valor de retorno podrían ser *char* y la función definirse como sigue (lo cual es correcto pero no conveniente):

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

Sin embargo, vemos que continuamente el compilador debe estar promoviendo para evaluar y demoviendo para asignar, lo cual es ineficiente; es por ello que la función debería escribirse como:

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

Si busca los prototipos de las funciones de biblioteca *standard* de C, encontrará que ninguna de ellas recibe valores *char* o *short* ni retornan *char* o *short*.

Continuará.....

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] Por ello, muchos no consideran al lenguaje C un lenguaje de alto nivel, sino más bien un *assembler* de alto nivel.
[iv] Todas las sentencias están escritas como en el caso del lenguaje C, aunque se haga referencia a otros lenguajes.
[v] Es interesante pedirle al compilador cuál es el código *assembler* que genera para comprender más claramente el problema.
[vi] No se ha incluído la extensión al caso de los procesadores de 64 bits
[vii] *default*
[viii] De hecho, el prefijo *signed* fué definido en la norma ANSI solo por el caso del *char*.
[ix] Obviamente, deben evitarse estos casos de asignación, ya que el resultado puede muy bien no representar el resultado de la operación esperada.
[x] Quizás valga la pena recordar que el lenguaje C se desarrolló con procesadores de 16 bits y, para esos procesadores, el *int* era de esa dimensión.
[xi] Siempre que, obviamente, alcance el tamaño de ese tipo o no se necesite pasar tipos de más alto rango, como flotantes o punteros.