



Sistemas de Computación

Práctico 2:

Stack frame

Alumnos: Cesana Andrés Agustín, Felipe Pillichody,
Sol Agustina Nou.

Profesores: Jorge Javier Alejandro, Solinas Miguel
Ángel.

1. Consigna

Se debe diseñar e implementar cálculos en ensamblador. La capa superior recuperará información de una API REST. Se recomienda el uso de API Rest mediante Python. Los datos de consulta realizados deben ser entregados a un programa en C que convocará rutinas en ensamblador para que hagan los cálculos de conversión y devuelvan los resultados a las capas superiores. Luego el programa en C o Python mostrará los cálculos obtenidos.

Se debe utilizar el stack para convocar, enviar parámetros y devolver resultados. O sea utilizar las convenciones de llamadas de lenguajes de alto nivel a bajo nivel.

1.1 Contexto

El índice GINI corresponde a una medida de desigualdad de ingresos de un país. Este coeficiente varía entre 0 y 100, donde 0 representa la igualdad perfecta (todos tienen los mismos ingresos) y 100 denota la desigualdad total (una persona acapara todos los ingresos y los demás no tienen ninguno).

Por otra parte una API REST es una interfaz de programación de aplicaciones que se ajusta a los principios de la arquitectura REST y facilita la interacción con servicios web. Su función es ser un intermediario del usuario y los recursos web que se quiere acceder.

2. Primera iteración

2.1 Implementación en Python

En esta primera versión, se desarrolló una aplicación completamente en Python, haciendo uso de la biblioteca *requests* para consumir la API REST del Banco Mundial. Teniendo en cuenta que estos lenguajes son más “amigables” para el programador, el programa en pocas líneas puede:

- Conectarse al endpoint correspondiente de la API y hacer una solicitud HTTP GET
- Extraer del JSON obtenido los valores del índice GINI para Argentina entre 2011 y 2020 y conformar un array de esos valores.
- Mostrar por pantalla ese array de floats
- Sumar 1 a cada uno de los *floats* de ese array.
- Convertir ese array de *floats* a un array de ints
- Sumar 1 a cada uno de los valores *ints* y volver a mostrarlo por pantalla.

2.1.1 Ejecución

<https://github.com/solnou/SdC-Cyber-Core/tree/main/TP2/python>

Compilar y ejecutar desde la carpeta raíz (SdC-Cyber-Core) con:

```
cd TP2/python
```

```
python3 full_python.py
```

```
(venv) sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/python$ python3 full_python.py
Float array from API (GINI index): [40.7, 42.4, 42.7, 43.3, 41.7, 41.4, 42.3, 41.8, 41.1, 41.4, 42.7]
Float values incremented by 1: [41.7, 43.4, 43.7, 44.3, 42.7, 42.4, 43.3, 42.8, 42.1, 42.4, 43.7]
Integer-converted values: [41, 42, 43, 43, 42, 41, 42, 42, 41, 41, 43]
Integer values incremented by 1: [42, 43, 44, 44, 43, 42, 43, 43, 42, 42, 44]
```

Figura 1: Resultado de la ejecución de la implementación en Python

2.2 Implementación en C

En paralelo, se desarrolló una aplicación escrita completamente en lenguaje C. El flujo de esta versión es:

1. **Definición de estructura dinámica (MemoryStruct)**
Se define una estructura que contiene un puntero char *memory y un entero size, que permite almacenar dinámicamente los datos descargados por *curl*.
2. **Callback de escritura (WriteMemoryCallback)**
Esta función es llamada por *curl* a medida que llegan datos. Con *realloc*, se agranda el buffer y se copia el contenido.
3. **Solicitud HTTP con libcurl**
Se configura *curl* para conectarse al endpoint del Banco Mundial y guardar la respuesta en *chunk.memory*.
4. **Parseo del JSON con cJSON**
 - Se parsea el contenido del JSON.
 - Se accede al segundo elemento (*data[1]*), que contiene el arreglo de resultados.
 - Se recorre el arreglo y se extraen los campos *value* válidos (no nulos).
 - Se guardan estos valores como *double* en un arreglo dinámico.
5. **Impresión y cálculo**
 - Se imprime el arreglo de float (double).
 - Se suma 1 a los valores y se vuelve a imprimir.
 - Luego, se convierte a un arreglo de int.
 - Se imprimen los enteros convertidos.
 - Se suma 1 los valores y se los vuelve a imprimir.
6. **Liberación de memoria**
Se libera toda la memoria reservada: buffers de JSON, arreglos dinámicos, respuesta de *curl*.

2.2.1 Ejecución

<https://github.com/solnou/SdC-Cyber-Core/blob/main/TP2/c/main.c>

Compilar y ejecutar desde la carpeta raíz (SdC-Cyber-Core) con:

```
cd TP2/c
make
```

./main

```
(venv) sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/c$ ./main
Float array from API (GINI index):[40.70, 42.40, 42.70, 43.30, 41.70, 41.40, 42.30, 41.80, 41.10, 41.40, 42.70]
Float values incremented by 1: [41.70, 43.40, 43.70, 44.30, 42.70, 42.40, 43.30, 42.80, 42.10, 42.40, 43.70]

Integer-converted values: [41, 43, 43, 44, 42, 42, 43, 42, 42, 42, 43]
Integer values incremented by 1: [42, 44, 44, 45, 43, 43, 44, 43, 43, 43, 44]
```

Figura 2: Resultado de la ejecución de la implementación en C

2.3 Resultados primera iteración

2.3.1 Tiempos

Como parte del análisis de la primera iteración, se midió el tiempo de ejecución total de las implementaciones en Python y C utilizando el comando *time* y el comando *perf* desde la consola de Linux.

El comando *time* nos permite conocer tres métricas:

- **real**: es el tiempo total que pasó desde que se ejecuta el programa hasta que termina. Incluye absolutamente todo: ejecución del código, llamadas a la red, acceso a disco, espera de entrada/salida, etc.
- **user**: es el tiempo que la CPU pasó ejecutando instrucciones del programa, sin entrar al kernel. Incluye procesar datos, hacer operaciones matemáticas, ejecutar bucles, parsear estructuras en memoria, entre otros.
- **sys**: es el tiempo que la CPU pasó ejecutando funciones del sistema operativo. Incluye cosas como: acceder a la red (HTTP request), leer/escribir en disco, reservar o liberar memoria y abrir archivos o sockets.

El comando *perf* es una herramienta avanzada de análisis de rendimiento en sistemas Linux que permite observar en detalle cómo interactúa un programa con el procesador y el sistema operativo durante su ejecución. A diferencia del comando *time*, que ofrece un resumen general del tiempo transcurrido y su distribución entre espacio de usuario y núcleo, *perf* entrega métricas de bajo nivel como la cantidad de ciclos de CPU utilizados, instrucciones ejecutadas, fallos de caché, predicciones erróneas de salto, cambios de contexto, y eventos relacionados con el acceso a memoria.

Estas métricas son fundamentales para identificar cuellos de botella en la ejecución, como demoras en la entrega de instrucciones, mal uso de la memoria caché o sobrecarga del sistema operativo. En el contexto de esta comparación entre las implementaciones en C y Python, *perf* permite evaluar no solo cuál versión es más rápida, sino también cuál utiliza los recursos del sistema de forma más eficiente. Esta herramienta resulta particularmente útil para programas escritos en lenguajes compilados como C, donde se tiene un control más directo sobre la ejecución del código máquina.

En esta ocasión no utilizamos gprof porque no está pensado para programas Python, lo que hubiese hecho injusta la comparación.

Ambos programas realizan las mismas tareas y los tiempos resultantes fueron:

```
(venv) sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/c$ time ./main
Float array from API (GINI index): [40.70, 42.40, 42.70, 43.30, 41.70, 41.40, 42.30, 41.80, 41.10, 41.40, 42.70]
Float values incremented by 1: [41.70, 43.40, 43.70, 44.30, 42.70, 42.40, 43.30, 42.80, 42.10, 42.40, 43.70]

Integer-converted values: [41, 43, 43, 44, 42, 42, 43, 42, 42, 42, 43]
Integer values incremented by 1: [42, 44, 44, 45, 43, 43, 44, 43, 43, 43, 44]

real    0m0.363s
user    0m0.039s
sys      0m0.014s
(venv) sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/c$ perf stat ./main
Float array from API (GINI index): [40.70, 42.40, 42.70, 43.30, 41.70, 41.40, 42.30, 41.80, 41.10, 41.40, 42.70]
Float values incremented by 1: [41.70, 43.40, 43.70, 44.30, 42.70, 42.40, 43.30, 42.80, 42.10, 42.40, 43.70]

Integer-converted values: [41, 43, 43, 44, 42, 42, 43, 42, 42, 42, 43]
Integer values incremented by 1: [42, 44, 44, 45, 43, 43, 44, 43, 43, 43, 44]

Performance counter stats for './main':

      41.06 msec task-clock                    #    0.095 CPUs utilized
         15      context-switches              #   365,362 /sec
          4      cpu-migrations                 #   97,430 /sec
        948      page-faults                   #   23,091 K/sec
    161.941.595 cycles                         #    3,944 GHz
    38.030.785 stalled-cycles-frontend         #   23.48% frontend cycles idle
    264.827.288 instructions                  #    0.14  stalled cycles per insn
         52.181.944 branches                   #    1,271 G/sec
          624.798 branch-misses                #    1.20% of all branches

    0.433549758 seconds time elapsed

    0.029532000 seconds user
    0.012656000 seconds sys
```

Figura 3: Ejecución de la implementación en C

```
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/python$ time python3 full_python.py
Float array from API (GINI index): [40.7, 42.4, 42.7, 43.3, 41.7, 41.4, 42.3, 41.8, 41.1, 41.4, 42.7]
Float values incremented by 1: [41.7, 43.4, 43.7, 44.3, 42.7, 42.4, 43.3, 42.8, 42.1, 42.4, 43.7]

Integer-converted values: [41, 42, 43, 43, 42, 41, 42, 42, 41, 41, 43]
Integer values incremented by 1: [42, 43, 44, 44, 43, 42, 43, 43, 42, 42, 44]

real    0m0.718s
user    0m0.190s
sys      0m0.042s
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/python$ perf stat python3 full_python.py
Float array from API (GINI index): [40.7, 42.4, 42.7, 43.3, 41.7, 41.4, 42.3, 41.8, 41.1, 41.4, 42.7]
Float values incremented by 1: [41.7, 43.4, 43.7, 44.3, 42.7, 42.4, 43.3, 42.8, 42.1, 42.4, 43.7]

Integer-converted values: [41, 42, 43, 43, 42, 41, 42, 42, 41, 41, 43]
Integer values incremented by 1: [42, 43, 44, 44, 43, 42, 43, 43, 42, 42, 44]

Performance counter stats for 'python3 full_python.py':

    191.53 msec task-clock                    #    0.199 CPUs utilized
         29      context-switches              #   151,410 /sec
          2      cpu-migrations                 #   10,442 /sec
         4.516      page-faults                 #   23,578 K/sec
    803.604.907 cycles                         #    4,196 GHz
    180.031.566 stalled-cycles-frontend         #   22.40% frontend cycles idle
    743.318.077 instructions                  #    0.92  insn per cycle
         152.504.299 branches                   #   796,231 M/sec
          4.408.937 branch-misses                #    2.89% of all branches

    0.964894498 seconds time elapsed

    0.160073000 seconds user
    0.032810000 seconds sys
```

Figura 4: Ejecución de la implementación en Python

A pesar de que ambos programas son funcionalmente equivalentes y generan los mismos resultados, se observan diferencias en cuanto al rendimiento. La versión en C muestra un mejor desempeño en términos generales, destacándose especialmente en el uso de la CPU. El programa en Python gastó 190 ms ejecutando su lógica, mientras que el programa en C solo necesitó 39 ms, lo que evidencia que Python tiene un mayor costo computacional para tareas simples como la realización de sumas o el recorrido de arrays.

En cuanto a la métrica sys, ambos programas pasaron un tiempo similar solicitando al sistema operativo, con solo pequeñas diferencias. Estos resultados indican que las diferencias de rendimiento no provienen principalmente de la interacción con el sistema operativo, sino del procesamiento de la lógica propia de cada programa.

2.3.1 Peso

```
(venv) sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/c$ ls -lh main
-rwxrwxr-x 1 sol sol 19K abr 20 18:58 main
(venv) sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/c$ ls -lh main.c
-rw-rw-r-- 1 sol sol 3,5K abr 20 18:58 main.c
```

Figura 5: Detalles de la implementación en C

```
sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/python$ ls -lh full_python.py
-rw-rw-r-- 1 sol sol 989 abr 20 18:57 full_python.py
```

Figura 6: Detalles de la implementación en Python

En el caso del código en C, el ejecutable generado tiene un tamaño de 19kB, mientras que el archivo fuente ocupa 3.5kB. En contraste, el script en Python apenas alcanza los 989bytes. Esta diferencia de tamaño es significativa y se explica tanto por la naturaleza del lenguaje como por el proceso de ejecución.

Por un lado, el código en C es compilado: al ser traducido a lenguaje máquina, se genera un archivo binario autónomo que incluye instrucciones optimizadas y, en muchos casos, dependencias incorporadas directamente. Esto incrementa notablemente el tamaño del archivo final, más allá de lo que ocupa el código fuente en sí. Python, en cambio, es un lenguaje interpretado: el código fuente se mantiene en formato texto y se ejecuta mediante un intérprete externo, lo cual permite que el archivo fuente sea mucho más liviano.

Además, Python permite escribir soluciones más breves y legibles, con menos líneas de código, gracias a su sintaxis de alto nivel. También cuenta con herramientas como el recolector de basura (garbage collector), que automatiza la gestión de memoria. En C, por el contrario, el programador debe encargarse explícitamente de asignar y liberar memoria, lo que suele aumentar la longitud y complejidad del código.

En resumen, tanto el tamaño del ejecutable como el del código fuente reflejan no solo diferencias técnicas entre lenguajes compilados e interpretados, sino también distintos niveles de abstracción y responsabilidades del programador.

2.3.2 Uso de recursos

A pesar de que Python resultó ser más fácil de programar, con una diferencia notable en la longitud de código, muestra un mayor consumo de recursos. Esto puede atribuirse al uso de librerías, el intérprete de Python y la mayor abstracción del lenguaje, lo que genera una sobrecarga comparado con la implementación en C, que permite un control más preciso y eficiente de los recursos del sistema.

La implementación en C es claramente más eficiente en cuanto al uso de recursos. Tiene un mayor IPC, y con menos eventos de penalización como fallos de página o predicciones fallidas, como vimos con la salida de *perf* en la *Figura 3*. Python, por su parte, requiere más tiempo de CPU, tiene menor rendimiento en el uso del procesador y depende mucho más del entorno de ejecución (intérprete, cargadores dinámicos, etc.).

3. Segunda iteración

En esta sección, se trabajó exclusivamente con la versión en C, y el objetivo fue replicar la funcionalidad previamente desarrollada, pero incorporando funciones escritas en código ensamblador x86. La intención de este enfoque fue eliminar parte de la abstracción que impone el lenguaje C y aprovechar las libertades que ofrece el acceso a bajo nivel. De este modo, se buscó evaluar hasta qué punto es posible optimizar el rendimiento al tener un control más directo sobre los recursos del sistema.

3.1 Capa superior

La capa superior está implementada en lenguaje C. En esta capa, se define la lógica para realizar la llamada a la API REST y manejar los datos obtenidos. Se utiliza la función *suma_1_gini* que toma un valor flotante como entrada, le suma 1 y lo devuelve como un entero. Esta función está implementada en ensamblador y se declara como externa. Luego el programa en C se encarga de imprimir los resultados, e iterar sobre todos los valores.

<https://github.com/solnou/SdC-Cyber-Core/blob/main/TP2/c%2Basm/main.c>

3.2 Capa inferior

En esta capa implementamos funciones en lenguaje ensamblador x86 (32 bits), utilizando el ensamblador NASM. El objetivo principal es trabajar a bajo nivel, respetando las convenciones de llamada por stack, y realizando las siguientes tareas:

- Sumarle uno al valor con el que fue llamada a la función
- Convertir el valor de float a int.
- Retornar el resultado por medio del registro EAX.

Estas funciones son invocadas desde C, y durante la depuración con GDB se observa el estado del stack antes, durante y después de su ejecución

<https://github.com/solnou/SdC-Cyber-Core/blob/main/TP2/c%2Basm/suma.asm>

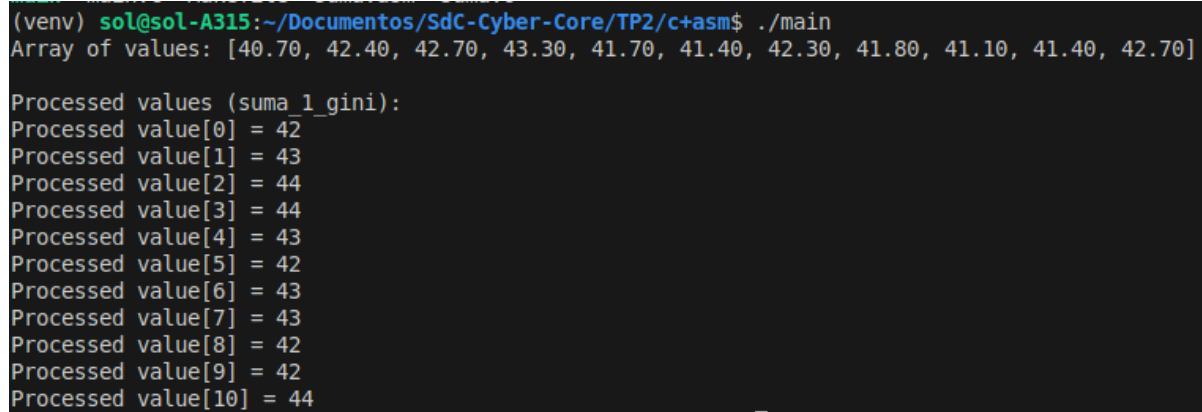
3.3 Ejecución

Compilar y ejecutar desde la carpeta raíz (SdC-Cyber-Core) con:

```
cd TP2/c+asm
```

```
make
```

```
./gini
```



```
(venv) sol@sol-A315:~/Documentos/SdC-Cyber-Core/TP2/c+asm$ ./main
Array of values: [40.70, 42.40, 42.70, 43.30, 41.70, 41.40, 42.30, 41.80, 41.10, 41.40, 42.70]

Processed values (suma_1_gini):
Processed value[0] = 42
Processed value[1] = 43
Processed value[2] = 44
Processed value[3] = 44
Processed value[4] = 43
Processed value[5] = 42
Processed value[6] = 43
Processed value[7] = 43
Processed value[8] = 42
Processed value[9] = 42
Processed value[10] = 44
```

Figura 7: Resultado de la ejecución de la implementación en C + asm

3.4 Debug

Esta parte del trabajo busca revisar cómo fueron utilizados los registros, el stack y verificar que se hayan respetado las convenciones de llamadas.

Al iniciar la depuración con GDB, se coloca un punto de interrupción (*breakpoint*) en la función `suma_1_gini`. Al alcanzarse dicho punto, se utiliza el comando *info registers* para observar el contenido de los registros del procesador. En este contexto, los registros se emplean para almacenar temporalmente valores intermedios, como los elementos del array que están siendo procesados. Observamos estas salidas en la primera llamada al método y en una segunda instancia, analizamos los registros después de esa llamada y antes de la segunda llamada:


```

Array of values: [40.70, 42.40, 42.70, 43.30, 41.70, 41.40, 42.30, 41.80, 41.10, 41.40, 42.70]

Processed values (suma_1_gini):

Thread 1 "main" hit Breakpoint 2, 0x080496a3 in suma_1_gini ()
(gdb) info registers
eax            0x4222cccd          1109576909
ecx            0xf7ec38a0         -135513952
edx            0x1                1
ebx            0x8166ba0          135687072
esp            0xffffcc08         0xffffcc08
ebp            0xffffcc08         0xffffcc08
esi            0xb                11
edi            0x0                0
eip            0x80496a3          0x80496a3 <suma_1_gini+3>
eflags         0x200296          [ PF AF SF IF ID ]
cs             0x23               35
ss             0x2b               43
ds             0x2b               43
es             0x2b               43
fs             0x0                0
gs             0x63               99
(gdb) x/4x $esp
0xffffcc08: 0xffffcc58  0x08049312  0x4222cccd  0x0804a033
(gdb) continue
Continuando.
Processed value[0] = 42

Thread 1 "main" hit Breakpoint 2, 0x080496a3 in suma_1_gini ()
(gdb) info registers
eax            0x4229999a          1110022554
ecx            0x0                0
edx            0xf6c39a00         -154953216
ebx            0x8166ba0          135687072
esp            0xffffcc08         0xffffcc08
ebp            0xffffcc08         0xffffcc08
esi            0xb                11
edi            0x1                1
eip            0x80496a3          0x80496a3 <suma_1_gini+3>
eflags         0x200296          [ PF AF SF IF ID ]
cs             0x23               35
ss             0x2b               43
ds             0x2b               43
es             0x2b               43
fs             0x0                0
gs             0x63               99
(gdb) x/4x $esp
0xffffcc08: 0xffffcc58  0x08049312  0x4229999a  0x0804a033

```

Figura 8: Análisis de los resultados con GDB

Analizando el primer breakpoint, en la llamada al método *suma_1_gini* podemos observar el valor del registro de propósito general:

eax = 0x4222cccd. Este valor hexadecimal representa un número en formato IEEE 754 de 32 bits (float). Si lo interpretamos como tal: $0x4222cccd \approx 40.7$. Este valor coincide con el primer elemento del array. Esto tiene sentido ya que estamos situados en la primera llamada a la función *suma_1_gini*, siendo consistente que en *eax* tengamos *values[0]*. Podemos decir entonces, que *eax* está siendo usado para **almacenar el valor actual del array que se está procesando**.

Siguiendo los registros que devuelve *info registers* observamos otros valores:

esi = 0xb → 11 (decimal). Este valor corresponde con la **longitud del array** de valores sobre el que vamos a iterar.

edi = 0x0. Representa el **índice actual** en una iteración. Es decir, el valor de edi sería igual al actual valor de i dentro del for que recorre el array de valores.

ebp = esp = 0xffffcc08. El hecho de que ambos registros coincidan sugiere que estamos justo al comienzo de la ejecución de la función. Aún no se han realizado operaciones push, ni se han reservado variables locales en la pila. Este comportamiento es típico inmediatamente después de un call.

Cabe destacar que el valor de esp no cambia entre llamadas sucesivas a la función porque esta se invoca y retorna de manera secuencial desde el código en C, sin anidar llamadas. Por lo tanto, no se acumulan *stack frames*. Si las llamadas a *suma_1_gini* fueran recursivas o estuvieran anidadas, sí se observarían desplazamientos en esp.

De la siguiente iteración del mismo punto de ruptura (antes de llamar a la función .asm) observamos que lo analizado antes tiene consistencia, ahora los valores de los registros son:

- **eax = 0x4229999a** ≈ 42.4: ahora se está procesando el segundo valor del array.
- **edi = 1**: el índice de iteración aumentó en 1.

En resumen, los registros reflejan claramente cómo se manipulan los datos dentro de la función *suma_1_gini()*:

Registro	Funcion
eaz	contiene el valor actual procesado del array en formato float.
edi	representa el índice del bucle
esi	almacena la longitud total del array.
esp/ebp	muestran el estado de la pila, útil para analizar llamadas a funciones
eip	señala la dirección de ejecución actual, útil para identificar en qué función estás

3.5 Convención de llamadas:

La convención de llamadas hace referencia a cómo una función recibe parámetros y devuelve valores (parámetros, valores de retorno y direcciones de retorno).

Como se pasan los argumentos a una función:

- Por registros (por ejemplo, usando R0, R1, etc. en ARM).
- Por la pila (*stack*), empujando los argumentos en un orden específico.

Qué registros modificar y cuáles conservar:

- Algunos registros deben mantenerse igual antes y después de la función (preservados).
- Otros pueden ser usados libremente por la función llamada.

Dónde se devuelve el valor de la función:

- Por lo general en un registro (como R0 en ARM o EAX en x86).

La convención de llamadas es fundamental porque permite que funciones escritas en ensamblador interactúen con código en C. La buena convención asegura compatibilidad entre distintos módulos o librerías y es clave para entender cómo funcionan interrupciones, depuración, y llamadas al sistema operativo.