Diseño Basado en Microprocesadores
Práctica 4
Programación en ensamblador x86 de 64 bits (II)

Índice

---

1. Enlazar funciones en C o ensamblador con C++

Como sabemos, existen una serie de lenguajes de programación, entre los que se encuentra C++, que permiten la llamada sobrecarga de funciones o métodos. Una función o método está sobrecargada cuando existen varias versiones de la misma distinguibles entre sí por su signatura (número, tipo y orden de los argumentos). Por ejemplo:

```
int funcion(void);
int funcion(int a);
int funcion(float a);
void funcion(char a, int b);
```

También puede haber funciones con el mismo nombre en diferentes *namespaces*.

Para que el enlazador pueda distinguir entre las distintas versiones de una función sobrecargada, el compilador de C++ añade al nombre base de la función una serie de caracteres que describen su signatura. Este proceso se denomina *name manging* o *name decoration*. Esto plantea problemas cuando deseamos enlazar código en C++ con C o ensamblador, ya que ni el compilador de C ni el ensamblador realizan el mismo proceso. Para solventarlo, basta añadir al prototipo de la función en C++ el modificador `extern "C"`. Por ejemplo, si queremos enlazar con C++ una función en C o ensamblador que se llame `prueba` que recibe dos argumentos de tipo `int` y retorna un `float`, declararemos el siguiente prototipo desde el fichero C++ desde el que deseemos llamarla:

```
extern "C" void prueba(int a, int b);
```

La presencia del modificador `extern "C"` evitará que el compilador de C++ realice el proceso de *name manging* con la función `prueba`.

Puede aplicarse el modificador a varias funciones encerrando sus prototipos entre llaves tras `extern "C"`. Por ejemplo:

```
extern "C"
{
  int una_funcion(int a, int b);
  int otra_funcion(void);
  long otra_funcion_mas(float a);
}
```

Si escribimos un módulo fuente en C o ensamblador con varias funciones que estamos interesados en que puedan usarse indistintamente desde C o desde C++, lo más conveniente es crear un fichero de cabecera similar al siguiente

```
#ifndef MI_FUENTE_H
#define MI_FUENTE_H   // Guarda habitual para evitar inclusión múltiple.

#ifdef __cplusplus    // Aplicar extern "C" sólo si es un compilador de C++.
extern "C"
{
#endif

// Prototipos de las funciones en C o ensamblador.

int una_funcion(int a, int b);
int otra_funcion(void);
long otra_funcion_mas(float a);

#ifdef __cplusplus    // Cerrar extern "C" sólo si es un compilador de C++.
}
#endif

#endif  // MI_FUENTE_H
```

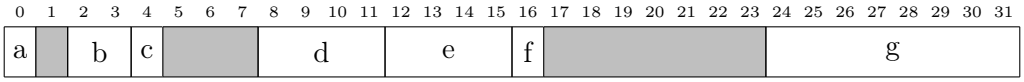
2. *Padding* de estructuras

Aunque en general no es estrictamente necesario, los compiladores para microprocesadores x86 intentan mantener la alineación correcta de los datos en memoria para así aumentar la velocidad a la que se puede acceder a éstos. Recordar que un dato está alineado cuando está almacenado a partir de una dirección de memoria divisible entre el tamaño en bytes del dato. En el caso de las estructuras de C, el compilador puede desplazar los campos de la estructura hasta que queden alineados realizando un rellenando o *padding* con bytes extra entre ellos. Por ejemplo, si declaramos la siguiente estructura

```
struct s1 {
    char   a;
    short  b;
```

```
    char   c;
    float  d;
    int    e;
    char   f;
    double g;
};
```

las instancias de `s1` quedarán almacenadas en memoria de la siguiente forma

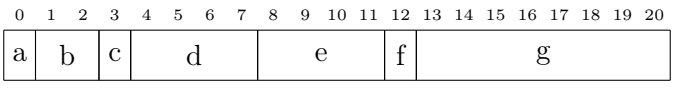


Un compilador de 64 bits siempre alinea el comienzo de una estructura en una dirección divisible entre 8 (uno de 32 bits en una divisible entre 4). Los números en la parte superior representan el desplazamiento relativo de cada byte de la estructura respecto a la primera dirección. Las zonas sombreadas son bytes de relleno que el compilador ha insertado para que la variable `b`, que ocupa dos bytes, quede a partir de una posición divisible entre dos, la variable `d`, que ocupa cuatro bytes, quede a partir de una posición divisible entre cuatro y la variable `g`, que ocupa ocho bytes, quede a partir de una posición divisible entre ocho. El compilador nunca añade bytes de relleno antes del primer miembro, es decir, la dirección de comienzo del primer miembro es la dirección de comienzo de la estructura y no altera el orden de los miembros.

En algunas aplicaciones (por ejemplo, para procesar cabeceras de ficheros o protocolos de red), es necesario evitar el *padding* de una estructura. Para ello, en el compilador gcc podemos usar el atributo `packed` al declarar la estructura. Por ejemplo:

```
struct s1 {
    char   a;
    short  b;
    char   c;
    float  d;
    int    e;
    char   f;
    double g;
} __attribute__((packed));
```

Al hacerlo así, los campos de la estructura quedarán "empaquetados" sin bytes de relleno, tal como se ve a continuación.



3. Ejercicios

3.1. Ejercicio 1

Escribe una función en ensamblador de 64 bits que calcule el histograma de un array de datos de tipo `unsigned char`, es decir, que cuente cuántas veces aparece cada valor posible para un dato de tipo `unsigned char` (de 0 a 255) en un array de datos de ese tipo. El prototipo de la función es:

```
int calcular_histograma(const unsigned char *ptr_datos,
                        unsigned int num_datos,
                        unsigned int *ptr_histograma);
```

donde

`ptr_datos` apunta al array con los datos de entrada.

`num_datos` indica el número de datos contenidos en el array apuntado por `ptr_datos`.

`ptr_histograma` apunta al array donde almacenar el histograma. Se supone que hay espacio para los 256 elementos del histograma.

La función retorna 1 si realizó su trabajo correctamente y 0 si alguno de los argumentos no es correcto.

Por ejemplo, tras ejecutar el siguiente fragmento

```
unsigned char array[10] = {1, 5, 4, 4, 4, 4, 1, 2, 1, 0};
unsigned int histograma[256];
calcular_histograma(array, 10, histograma);
```

el array `histograma` quedaría con los valores `1, 3, 1, 0, 4, 1, 0, 0, 0, 0, 0,..., 0` porque el 0 aparece 1 vez, el 1 aparece 3 veces, el 2 aparece 1 vez, el 3 no aparece ni una sola vez, el 4 aparece 4 veces, el 5 aparece una vez y el resto de valores entre 6 y 255 no aparecen ni una sola vez.

> Llama a la función desde un fichero fuente en C++. Para ello debes crear un proyecto C++ en Eclipse.

3.2. Ejercicio 2

En un programa se han definido las siguientes estructuras:

```
struct estructura {
    short  a;
    char   b[3];
    int    c;
    long   d;
};

struct estructura_empaquetada {
    short  a;
```

```
    char   b[3];
    int    c;
    long   d;
} __attribute__((packed));
```

Escribe un fichero fuente en ensamblador de 64 bits con dos funciones que permitan copiar una instancia de una estructura de un tipo en otra del otro tipo, por ejemplo:

```
int empaquetar(const struct estructura *origen,
               struct estructura_empaquetada *destino);

int desempaquetar(const struct estructura_empaquetada *origen,
                  struct estructura *destino);
```

Tanto una como otra retorna 1 si realizó su trabajo correctamente y 0 si alguno de los punteros es nulo.

Escribe un programa de prueba para mostrar que la copia se realiza correctamente en ambos casos.

3.3. Ejercicio 3

Escribe una función en ensamblador de 64 bits que compruebe si una cadena de caracteres es un palíndromo, es decir, si se lee igual de izquierda a derecha y de derecha a izquierda. El prototipo de la función es:

```
int es_palindromo(const char *cadena);
```

donde `cadena` es el puntero a la cadena de caracteres a comprobar.

La función retorna 1 si la cadena es un palíndromo, 0 si no lo es y -1 en caso de que el puntero `cadena` sea nulo.