C: Tipos de variables y su ubicación en memoria

1. Tipos de variables en C

Una variable en C pueden clasificarse según su alcance o visibilidad en:

- **Local**: puede accederse solamente desde la función que la ha definido.
- **Global**: puede accederse desde varias funciones.

A su vez, una variable local puede ser de dos tipos, según si su valor se pierde o se mantiene una vez que ha finalizado la ejecución de la función f() que la ha definido:

- **Auto** (por defecto): se crea espacio para la variable en cada llamada a la función f(), y se libera dicho espacio cuando finaliza la función con lo que se pierde el valor de la variable.
- **Static**: el valor de una variable de cada tipo se mantiene entre distintas llamadas a la función f() que la ha definido.

A continuación se muestra un ejemplo que ilustra el comportamiento de estos tipos de variables locales:

```
#include <stdio.h>
#include <stdlib.h>

void incrementar()
{
      int i = 0;
      static int j = 0;  /* esta inicialización solamente se realiza la
                            primera vez que se ejecuta incrementar() */

      i++;
      j++;
      printf("i: %d, j: %d\n", i, j);
}

int main(int argc, char *argv[])
{
    incrementar();
    incrementar();
}
```

Si compilamos y ejecutamos el código anterior:

```
$ gcc local.c -o local
$ ./local
i: 1, j: 1
i: 1, j: 2
```

Por su parte, también pueden distinguirse dos tipos de variable global según su visibilidad:

- **Auto** (por defecto): la variable puede ser referenciada por todas las funciones de la

aplicación. Si la función que referencia la variable está definida en un fichero distinto debe declarar la variable como **extern**.

- **Static**: la variable puede ser referenciada solamente por las funciones definidas en el mismo fichero que la variable. No puede ser accedida por funciones definidas en otros ficheros de la aplicación.

A continuación se muestra un ejemplo que ilustra el comportamiento de estos tipos de variables globales:

```
/* fichero global1.c */

#include <stdio.h>
#include "global2.h"

/* declaración de función definida
   en otro fichero */
extern void decrementar();

/* declaración de variables */
int i;         /* auto */
static int j;  /* visible solamente
                  por las funciones
                  de este fichero */

void incrementar()
{
    i++;
    j++;
    printf("(inc) i: %d, j: %d\n",
                           i, j);
}

int main(int argc, char *argv[])
{
    /* inicializ. de variables */
    i = 9;
    j = 9;

    incrementar();
    decrementar();
    incrementar();
}
```

```
/* fichero global2.c */

#include <stdio.h>

extern int i; /* declaración de
                 variable global
                 definida en el
                 fichero global1.c
               */

void decrementar()
{
    i--;
    printf("(dec) i: %d\n", i);
}
```

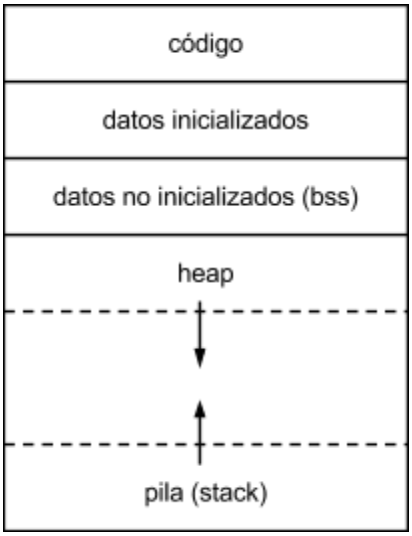
Si compilamos y ejecutamos el código anterior:

```
$ gcc global1.c global2.c -o global
$ ./global
(inc) i: 10, j: 10
(dec) i: 9
(inc) i: 10, j: 11
```

2. Espacio de direcciones de un proceso

La siguiente figura muestra la disposición de un proceso Unix/Linux en memoria (espacio de

direcciones de un proceso):



Las regiones de memoria son las siguientes:

- **Código** (**text**): también llamado segmento de texto. Contiene las instrucciones del programa.
- **Datos inicializados** (**data**):contiene las variables globales y estáticas del proceso que **sí** han sido inicializadas en su declaración..
- **Datos no inicializados** (**bss**): contiene las variables globales que **no** han sido inicializadas en su declaración y las variables locales estáticas.
- **Heap**: espacio dedicado a las variables cuya asignación de memoria se hace de forma dinámica en tiempo de ejecución (ver siguiente sección). Esta situación ocurre cuando no se conoce en tiempo de compilación el tamaño de una variable (por ejemplo, el número de elementos de un vector).
- **Pila** (**stack**): espacio de memoria que almacena las variables locales (las de tipo auto, cuya asignación de memoria se realiza de forma automática) y las direcciones de retorno de subrutinas.

El siguiente código permite observar cómo las variables de distinto tipo se almacenan en diferentes regiones de memoria[1]:

```
#include <stdio.h>

char uninit1, uninit2;
char init1 = 1, init2 = 2;

int main (void)
{
  char local1 = 1, local2 = 2;

  printf("bss\t%p\t%p\n", &uninit1, &uninit2);
  printf("data\t%p\t%p\n", &init1, &init2);
  printf("auto\t%p\t%p\n", &local1, &local2);
}
```

[1] Código tomado de http://stackoverflow.com/questions/12798486/bss-segment-in-c

Si compilamos y ejecutamos el código anterior:

```
$ gcc memory_layout.c -o memory_layout
$ ./memory_layout
bss     0x6008f0        0x6008f1
data    0x6008dc        0x6008dd
auto    0x7fff2bab04cf  0x7fff2bab04ce
```

La orden `size` permite obtener el tamaño de las secciones `text`, `data` y `bss` de un fichero objeto:

```
$ size memory_layout
   text    data     bss     dec     hex filename
   1244     496      24    1764     6e4 memory_layout
```

Hay más espacio

3. Asignación de memoria

Existen tres modos de asignación de memoria a las variables en C:

- **Estática**: a las variables globales y estáticas de un proceso se les reserva memoria en data o bss (según estén inicializadas o no). La asignación de memoria se realiza en tiempo de compilación, por lo que debe conocerse el espacio que requieren las variables (por ejemplo, el número de elementos de un vector).
- **Automática**: a las variables locales de tipo auto se les reserva espacio en la pila en tiempo de ejecución cuando se entra en la función f() que las declara. Una vez que finaliza la ejecución de la función f(), se libera el espacio reservado para dichas variables.
- **Dinámica**: a las variables de cualquier tipo cuyo tamaño no se conoce en tiempo de compilación se les asigna espacio en el heap durante la ejecución del proceso. Esto se realiza mediante una llamada al sistema -`malloc()`/`calloc()`-, y las referencias al espacio asignado se efectuarán mediante un puntero. Mediante otra llamada al sistema puede liberarse el espacio asignado a variables que ya no van a utilizarse -`free()`-.

El siguiente código muestra los tres modos de asignación de memoria: estática (en tiempo de compilación), automática, y dinámica (en tiempo de ejecución). En concreto se declaran tres vectores, dos globales (`x`, `y`) y otro local (`z`) en la función `main()`. Se reserva memoria en tiempo de compilación para el vector `x` (se conoce su tamaño), mientras que para el vector `y` se asigna memoria en el heap en tiempo de ejecución (no se conoce su tamaño al compilar el código). Al vector `z` se le asigna memoria en la pila una vez que comienza la ejecución de `main()`:

```
#include <stdio.h>
#include <stdlib.h>

#define SIZEARRAY    8192
int x[SIZEARRAY];   /* variable global, memoria asignada al vector en
```

```
                        tiempo de  compilación (en bss) */
int *y;              /* variable global, el vector no tiene espacio
                        asignado en memoria (el puntero sí) */

void fill_array(int *v, int num_elem)
{
    int i = 0;

    for (i = 0; i < num_elem; i++)
        v[i] = (int) rand();
}

void mean_array(int *v, int num_elem)
{
    int i = 0, sum = 0;

    for (i = 0; i < num_elem; i++)
        sum += v[i];

    printf("mean=%.2f\n", (float) sum/num_elem);
}

int
main(int argc, char *argv[])
{
    int num_elem = -1;
    int z[SIZEARRAY];   /* variable local (en la pila) */

    if (argc == 2) num_elem = atoi(argv[1]);
    if (num_elem == -1) num_elem = 8192;

    /* asignación dinámica de memoria a la variable y */
    y = (int *) calloc(num_elem, sizeof(int));
    if (y == NULL)
    {
      printf("Error: no se ha podido asignar memoria a x (calloc)\n");
      exit(0);
    }

    fill_array(x, SIZEARRAY);
    fill_array(y, num_elem);
    fill_array(z, SIZEARRAY);

    mean_array(x, SIZEARRAY);
    mean_array(y, num_elem);
    mean_array(z, SIZEARRAY);

    exit(0);
}
```

3. Bibliografía

En el siguiente enlace se describe la distribución de memoria en un programa Linux, y más en detalle, el espacio de direcciones de un proceso Linux:
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

En el siguiente enlace explican de otra forma la asignación de memoria a variables en C:
http://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-and-C.html#Memory-Allocation-and-C