Tecnicatura Superior en Análisis de Sistemas

Programación I

"Administración de Memoria y Apuntadores en ANSI C"
Versión 2.0 Marzo 2008

Lic. Guillermo R. Cherencio

INDICE

Introducción

Según Niklaus Wirth los programas son Algoritmos + Datos; ambos estarán en alguna parte de la memoria del computador durante la ejecución del programa. Cuando escribimos un programa, los datos los representamos con variables y constantes. Cada variable y/o constante será de un tipo de dato determinado, acorde con nuestras necesidades. La cantidad de espacio en memoria requerido para almacenar una variable o constante esta en función de su tipo de dato.

Para almacenar una variable o constante, en algún momento, se deberá solicitar al administrador de memoria del sistema operativo espacio libre en memoria suficiente para almacenar la variable (asignación de memoria). Esta solicitud podrá ser satisfecha o no, dependiendo si hay o no memoria disponible.

Asignación de Memoria en C

El lenguaje C soporta dos tipos de asignación de memoria para almacenar las variables en memoria:

1. Asignación estática (static allocation)
2. Asignación automática (automatic allocation)

Asignación estática

Cuando se declara una variable de tipo *static* o que tiene alcance *global* dentro del programa, se realiza una asignación de memoria estática. Esta asignación se realiza una sola vez y el espacio de memoria nunca más es liberado. Ejemplo:

```
#include <stdio.h>
static char buffer[1024];
static int bufp = 0;
int v_global=4;

int main(int argc, char * argv[]) {

   ...

}
```

Ej1.c

El espacio de memoria requerido para las variables: *buffer, bufp, v_global* es estático. El alcance de las variables *buffer, bufp* esta acotado al archivo Ej1.c que las declaró. El alcance de la variable *v_global* es global en toda la aplicación. Recordemos que una aplicación C puede estar formada por varios archivos .c , .h , etc.

Asignación automática

Cuando se declara una variable local o un argumento de una función, se produce una asignación automática de memoria. La asignación de memoria sucede cuando se ejecuta el bloque de código ({...}) que contiene la declaración y la liberación de memoria sucede cuando termina la ejecución del bloque de código (cuando se sale del mismo y ya no hay posibilidad de volver al mismo, a menos que se trate de otra ejecución). Estas asignaciones de memoria se realizan en una area de memoria llamada Pila (stack), es un area de memoria muy dinámica, pequeña y de alta velocidad. Ejemplo:

```
#include <stdio.h>

int sumar(int,int);
int main(int argc, char * argv[]) {
   printf("Sumo 4+2=%d\n",sumar(4,2));
   exit(0);
}

int sumar(int a,int b) {
   int r = a + b;
   return r;
}
```

Ej2.c

Cuando comienza la ejecución del programa Ej2.c, se ejecuta la función main() y se produce la asignación de memoria automática de los argumentos *argc, argv*; y se libera el espacio de memoria ocupado por estos argumentos cuando termina la ejecución de la función main(); puesto que la declaración de las mismas "alcanzan" a toda la función main(). Lo mismo sucede con los argumentos *a, b* de la función sumar(); se realiza su asignación automática en el momento en que se ejecuta la función sumar() y el espacio ocupado por *a, b* se libera cuando termina la ejecución de la función sumar(). Otro ejemplo:

```
#include <stdio.h>

int main(int argc, char * argv[]) {
   if ( argc > 1 ) {
      char *fuera_de_alcance = "variable fuera de alcance";
   }
   printf("mensaje imposible: %s\n",fuera_de_alcance);
   exit(0);
}
```

Ej3.c

La variable *fuera_de_alcance* es una variable declarada automáticamente en caso de entrar en ejecución el código de bloque ({ char *fuera_de_alcance ...; }) que corresponde la bifurcación "por verdad" de la expresión condicional (argc > 1) ; luego de ello, la variable será liberada y la misma ya no existirá más. Teniendo en cuenta que la instrucción *printf()* se ejecuta siempre (sea o no verdad la expresión condicional (argc > 1)), la misma dará error de compilación, puesto que la variable *fuera_de_alcance* no ha sido declarada dentro del código de bloque que contiene a la instrucción *printf()*; por lo tanto, no es lícito usarla allí.

Asignación Dinámica de Memoria

La asignación dinámica de memoria no es soportada directamente por las variables C, pero esta disponible a través de la biblioteca (librería) standard (stdlib.h). La asignación dinámica es una técnica mediante la cual los programas pueden solicitar memoria al administrador de memoria del sistema operativo –en tiempo de ejecución (runtime)- acorde con sus necesidades. Es ideal cuando el problema a resolver no permite saber de antemano (antes de la ejecución del programa) cuánta memoria necesitamos para resolverlo. Supongamos que debemos hacer un programa que permita cargar en memoria el contenido de archivos de texto indicados –en tiempo de ejecución (runtime)- por el usuario; como no sabemos la longitud de los archivos hasta que el usuario nos indique qué archivo hay que cargar (y para ello requiere que el programa ya se encuentre en ejecución), la aplicación no tiene forma de asignar previamente la memoria que necesitará de antemano. Solo en estos casos debemos hacer uso de la asignación dinámica de memoria, utilizando las funciones de la librería standard (stdlib) tales como: *malloc()* y *calloc()*.

Pasaje de Parámetros

En el lenguaje C, todo el código que hacemos son funciones. Las funciones pueden[1] devolver algún valor y también pueden recibir información desde el exterior de la misma a través de sus argumentos. Ejemplo:

```
#include <stdio.h>

/* declaro funciones (prototipos) */[2]
void no_dev_no_info();
int dev_no_info();
int dev_con_info(int);
/* fin declaro */

int v_global = 4;

int main(int argc, char * argv[]) {
   // uso funciones
   no_dev_no_info();
   int retorno = dev_no_info();
   int paso_info = 4;
   int otro_retorno = dev_con_info(paso_info);
   printf("main():\n");
   printf("dev_no_info() devolvio %d\n",retorno);
   printf("dev_con_info(%d) devolvio %d\n",paso_info,otro_retorno);
   return 0;
}
/* defino funciones */
void no_dev_no_info() {
   printf("no_dev_no_info():\n");
   printf("funcion que hace algo\n");
   printf("pero no recibe ningun argumento ni devuelve nada\n");
   printf("no necesito usar la instruccion return");
}
int dev_no_info() {
   printf("dev_no_info():\n");
   printf("funcion que hace algo\n");
   printf("no recibe ningun argumento y devuelve un entero\n");
   printf("necesito usar la instruccion return <expr> donde\n");
   printf("<expr> debe ser de tipo int\n");
   return 0;
}
```

[1] Las funciones que no devuelven ningún valor tienen como tipo de retorno void. Ejemplo: void no_devuelvo_nada(); es una declaración de una función que no regresa ningún valor (esto en otros lenguajes procedurales se dice que no_devuelvo_nada() es una rutina).

[2] Obsérvese que en las declaraciones de las funciones no es obligatorio "nombrar" a los argumentos de las funciones, en la declaración sólo indicamos "el formato o prototipo" de la función: cómo se llama, que tipos de argumentos tiene, cuántos y en qué orden y que valor retorna.

```
int dev_con_info(int a) {
   printf("dev_con_info():\n");
   printf("funcion que hace algo\n");
   printf("no recibe ningun argumento y devuelve un entero\n");
   printf("necesito usar la instruccion return <expr> donde\n");
   printf("<expr> debe ser de tipo int\n");
   return 0;
}
/* fin defino funciones */
```

Ej4.c

En Ej4.c podemos ver que la función *main()* realiza un "pasaje de parametros" cuando invoca a la función *dev_con_info()* (`int otro_retorno = dev_con_info(paso_info);`) utilizando la variable *paso_info*. En esta situación se esta utilizado memoria automática. Cada vez que se encuentra –durante la ejecución del programa- un nuevo código de bloque[3] o llamada a una función, se "solapa", se "apila", se crea, se asigna, un area de memoria con espacio suficiente para almacenar todas las variables y constantes requeridas para la ejecución de ese código de bloque o función. A esta area de memoria se la denomina Pila o Stack.

La Pila o Stack

La ejecución de un programa C siempre requerirá una forma de resolver las llamadas entre funciones. Por razones de modularidad, seguridad y robustez, la filosofía que ha predominado en este problema es hacer que las funciones se ejecuten en un espacio de memoria separado. Dicho espacio de memoria separado se asigna utilizando asignación automática. Si cada función se ejecuta en un espacio de memoria separado, entonces ¿cómo es posible intercambiar información entre las funciones?, esto es posible a través de:

1. Variables y constantes definidas en areas de memoria estática (asignación estática)
2. Pasaje de parámetros
3. Valores retornados por las funciones

En el Ej4.c, la función main() tiene su espacio de memoria el cual esta separado del espacio de memoria de las otras funciones. Es ilícito mencionar la variable paso_info dentro de la función dev_con_info()[4].

[3] Recordemos que un código de bloque es todo aquello que está delimitado entre llaves: { ... } , pueden ser una o más instrucciones C y hasta incluso, se pueden anidar n código de bloques en su interior.

[4] En este caso, también es ilícito hacerlo en cualquier otra función que no sea main().

¿Qué información tiene disponible una función?:

1. Los argumentos de la función (si existiera alguno).
2. Las variables locales o automáticas de la función[5]. Las cuales representan un espacio de almacenamiento temporario, puesto que finalizada la función estos valores se liberan de la memoria.
3. Las variables y constantes definidas en areas de memoria estática.

Ejemplo de Funcionamiento de la Pila o Stack

Volvamos nuevamente al Ej4.c, antes de comenzar la ejecución de la función main(), se produce una asignación automática de memoria en una area de memoria llamada Pila o Stack, allí se asignarán las siguientes variables:

Referencia a variable o constante	**Dirección de memoria[6]**	**Contenido inicial[7]**
retorno	100	0
paso_info	104	0 *(luego cambiado a 4)*
otro_retorno	108	0
“main():\n”	200	main():\n\0[8]
“dev_no_info() devolvio %d\n”	210	dev_no_info() devolvio %d\n\0
“dev_con_info(%d) devolvio %d\n”	240	dev_con_info(%d) devolvio %d\n\0
0	250	0 *(valor de retorno de main())*
4	254	4
argc	258	1
argv	262	300[9]
“ej4.exe”[10]	300	ej4.exe\0
v_global	10000	4

Cuando la función main() ejecute a la función *no_dev_no_info()* , se producirá una asignación automática y la Pila o Stack tendrá lo siguiente:

[5] Las variables o constantes declaradas dentro de la propia función.
[6] Las direcciones de memoria no son reales, simplemente son dadas a modo de facilitar la explicación.
[7] Cuando se declara una variable, a dicha variable se le asigna un valor por defecto (implícito) y luego dicho valor se reemplazará por el valor asignado explícitamente en el programa.
[8] Recuerde que todas las cadenas de caracteres (strings) terminan con un \0 (caracter nulo).
[9] Obsérvese que argv es un puntero, por lo tanto, guarda una dirección de memoria.
[10] Se supone que el programa Ej4.c fue compilado y ejecutado como ej4.exe sin pasarle ningún argumento en la línea de comandos del sistema operativo.

Referencia a variable o constante	Dirección de memoria[11]	Contenido inicial[12]
“no_dev_no_info():\n”	400	no_dev_no_info():\n\0
“funcion que hace algo\n”	420	funcion que hace algo\n\0
“pero no recibe ningun argumento ni devuelve nada\n”	444	pero no recibe ningun argumento ni devuelve nada\n\0
“no necesito usar la instruccion return”	494	no necesito usar la instruccion return\0
v_global	10000	4

Cuando la función main() ejecute a la función *dev_con_info ()* , se producirá una asignación automática y la Pila o Stack tendrá lo siguiente:

Referencia a variable o constante	Dirección de memoria[13]	Contenido inicial[14]
a	600	0 *(luego cambiado a 4)*
0	604	0 *(valor de retorno)*
“dev_con_info():\n”	610	dev_con_info():\n\0
“funcion que hace algo\n”	627	funcion que hace algo\n\0
“no recibe ningun argumento y devuelve un entero\n”	651	no recibe ningun argumento y devuelve un entero\n\0
“necesito usar la instruccion return <expr> donde\n”	702	necesito usar la instruccion return <expr> donde\n\0
“<expr> debe ser de tipo int\n”	753	<expr> debe ser de tipo int\n\0
v_global	10000	4

Pasaje por Valor

Observe el argumento *a* de la función *dev_con_info()*, dicho argumento tiene una dirección de memoria distinta que la variable *paso_info* de la función *main()*. Esto implica que en cada llamada la función *dev_con_info()*, el valor de la variable *paso_info* es copiado a otra dirección de memoria asignada dinámicamente en el espacio de memoria de la función *dev_con_info()*. Se dice que *paso_info* fue usada desde *main()* para pasarle esa información a la función *dev_con_info()* (pasaje de parámetros). En este caso, se trata de un pasaje de parámetros por valor.

[11] Las direcciones de memoria no son reales, simplemente son dadas a modo de facilitar la explicación.
[12] Cuando se declara una variable, a dicha variable se le asigna un valor por defecto (implícito) y luego dicho valor se reemplazará por el valor asignado explícitamente en el programa.
[13] Las direcciones de memoria no son reales, simplemente son dadas a modo de facilitar la explicación.
[14] Cuando se declara una variable, a dicha variable se le asigna un valor por defecto (implícito) y luego dicho valor se reemplazará por el valor asignado explícitamente en el programa.

Pasaje por Referencia

Volviendo al ejemplo anterior, si el argumento *a* de la función *dev_con_info()* tuviera la dirección de memoria 104 (dirección de memoria de la variable *paso_info* de *main()*), entonces se hubiera tratado de un pasaje de parámetros por referencia. Es decir, cuando el pasaje de parámetros se hace por referencia, los valores no se copian, sino que se pasan sus direcciones de memoria de forma tal de apuntar a la misma dirección.

Por Valor vs. Por Referencia

El pasaje de parámetros por referencia es más rápido que el pasaje de parámetros por valor, por el hecho de evitarse el tiempo que tarda la asignación automática de memoria.

El pasaje de parámetros por referencia gasta menos memoria que el pasaje de parámetros por valor.

El pasaje de parámetros por referencia viola el concepto filosófico de que cada función corra dentro de su espacio de memoria, porque deja abierta una puerta (a través de los argumentos) para acceder y modificar datos que estarían fuera del alcance de la función. Esto puede crear serios problemas en la programación y prueba de las aplicaciones, puesto que el mal funcionamiento de una función podría afectar a otras, haciendo que los valores locales de una función cambien luego de la ejecución de otra función.

El lenguaje C utiliza el pasaje de parámetros por valor.

Apuntadores o Punteros

Un apuntador o puntero es una variable cuyo contenido es la dirección de memoria de otra variable. Si la variable *A* contiene la dirección de memoria de la variable *B*, entonces se dice que *A* apunta a *B*. Un puntero debe tener un tipo de dato compatible con la variable apuntada. Por Ejemplo, no se puede hacer que un apuntador de tipo *int* apunte a una variable de tipo *char*.

Apuntadores y cadenas de caracteres

Para declarar un puntero a una cadena de caracteres hay que declararlo de tipo char *, ejemplo:

```
char *apuntador_a_cadena;
```

Luego se puede asginar el puntero con una constante de tipo char * :

```
char *apuntador_a_cadena = "cadena de caracteres apuntada";
```

otra forma podría ser:

```
char *apuntador_a_cadena;
apuntador_a_cadena = "cadena de caracteres apuntada";
```

en realidad, *apuntador_a_cadena* guarda la dirección de memoria del primer caracter de la cadena apuntada. Cada uno de los caracteres de una cadena puede referenciarse a través de un sub-indice que comienza en la posición 0 (cero), ejemplo:

```
char *apuntador_a_cadena;
apuntador_a_cadena = "cadena de caracteres apuntada";
char primer_caracter = apuntador_a_cadena[0];
```

primer_caracter guardará en su interior el caracter 'c' correspondiente a la primer letra de la cadena apuntada por *apuntador_a_cadena*. Las cadenas de caracteres son en realidad arreglos de caracteres. Un arreglo o vector es una variable que en vez de contener un único valor, permite almacenar n valores (desde 0 hasta cantidad de elementos – 1). Cada uno de los valores almacenados se los puede ubicar a través del subíndice (0 ... n -1), el cual es un valor de tipo *int* que se usa para referirse al n-ésimo elemento del arreglo o vector, ejemplo:

```
char *apuntador_a_cadena;
apuntador_a_cadena = "cadena de caracteres apuntada";
char primer_caracter = apuntador_a_cadena[0];
char tercer_caracter = apuntador_a_cadena[2];
int i;
for(i=0;i<strlen(apuntador_a_cadena);i++) {
   printf("caracter [%d] = %c\n",i,apuntador_a_cadena[i]);
}
```

Apuntadores y aritmética de direcciones

Los apuntadores son variables que guardan direcciones de memoria de otras variables. Un apuntador se puede incrementar, ello provocará que apunte a la siguiente dirección de memoria, pero con la particularidad de tener en cuenta el tipo de dato. Supongamos que un *int* ocupa 4 bytes, si tengo un puntero a *int* que apunta a la dirección de memoria 100 y luego incremento en 1 a dicho puntero, en realidad apuntará a la dirección 104 (en vez de 101), puesto que un *int* ocupa 4 bytes. Otra forma análoga (y más eficiente) de escribir el código del punto anterior:

```
char *apuntador_a_cadena;
apuntador_a_cadena = “cadena de caracteres apuntada”;
char primer_caracter = *(apuntador_a_cadena+0); // no es necesario +0
char tercer_caracter = *(apuntador_a_cadena+2);
int i;
for(i=0;i<strlen(apuntador_a_cadena);i++) {
   printf(“caracter [%d] = %c\n”,i,*(apuntador_a_cadena+i));
}
```

Apuntadores y pasaje de parámetros

El pasaje de parámetros continua siendo por valor, lo que implica que al invocar una función pasando un puntero o apuntador como argumento de la función, dicho puntero será copiado y dentro de la función invocada se trabajará sobre la “copia del mismo”, al igual que sucede con las variables normales.

Esto permite “emular” de alguna forma el pasaje de parámetros por referencia. Supongamos que tenemos un apuntador de tipo *char* * a una cadena de caracteres muy grande, es mucho más económico copiar el puntero que toda la cadena de caracteres, ejemplo:

```
void imprimo_largo(char *);
int main() {
   char *buffer = “cadena de caracteres muy muy grande ... .”;
   imprimo_largo(buffer);
   return 0;
}
void imprimo_largo(char *b) {
   printf(“largo de buffer: %d\n”,strlen(b));
}
```

Al ser invocada la función *imprimo_largo()*, el apuntador *buffer* es copiado en *b*, para luego, calcular su longitud. Esto es más económico que tener que copiar todo el contenido al cual apunta *buffer*.

Otra implicancia importante es que todo lo que hagamos sobre *b*, afectará sólo a *b*, puesto que es una copia de *buffer* y éste no será afectado. Por ejemplo, si hiciéramos:

```
void imprimo_largo(char *);
int main() {
   char *buffer = "cadena de caracteres muy muy grande ... .";
   imprimo_largo(buffer);
   return 0;
}
void imprimo_largo(char *b) {
   b = "otra cadena distinta a la anterior";
   printf("largo de buffer: %d\n",strlen(b));
}
```

La función imprimirá el largo de la cadena *"otra cadena distinta a la anterior"* en vez de la original, pero luego de terminada la ejecución de la función *imprimo_largo()*, el apuntador *buffer*, en la función *main()* seguirá intacto, puesto que sólo se modificó la copia del mismo dentro de la función *imprimo_largo()*.

Apuntadores y asignación dinámica de memoria

Dentro de la biblioteca standard <stdlib.h> hay una serie de funciones que se utilizan para la asignación dinámica de memoria, las más usadas son:

```
void * calloc(size_t numero_de_objetos,size_t tamanio);
void * malloc(size_t tamanio);
void  free(void *puntero);
```

El tipo de dato *size_t* es un valor entero. Las funciones *calloc()* y *malloc()* sirven para solicitar memoria al administrador de memoria del sistema operativo mientras el programa esta en ejecución. *Tamanio* se refiere a la cantidad de memoria a solicitar. La función *calloc()* esta orientada a ser utilizada para asignar memoria para arreglos o vectores. *Malloc()* esta orientada a variables atómicas (no arreglos). *Calloc()* asigna memoria y el buffer asignado queda inicializado con valor *0* (*null*), mientras que la memoria asignada con *malloc()* no se inicializa a ningún valor en particular (puede contener datos previos, basura). *Free()* permite liberar el espacio de memoria previamente asignado. Tanto *calloc()* como *malloc()* devolverán *null* si no fue posible satisfacer el pedido de asignación de memoria, caso contrario, devolverán un puntero a la memoria asignada. Veamos un ejemplo:

```
...
int main() {
   char *buffer;
   buffer = (char *) malloc(15000);  // casting a char *
   if ( buffer != NULL ) {
      printf("pude asginar 15000 bytes ok!\n");
      strcpy(buffer,"copio estos caracteres");
      printf("buffer contiene: %s\n",buffer);
      free(buffer);
      printf("he liberado la memoria usada por buffer\n");
   } else {
      printf("no hay suficiente memoria!\n");
   }
   return 0;
}
```

Las funciones de asignación de memoria son genéricas (no hay una función de asignación por cada tipo de dato), por ello, son de tipo *void* * (lo que significa que devuelve un puntero de tipo *void* , que deberá ser transformado (casting) al tipo de dato en particular según nuestras necesidades, en este caso, *char* *); por ello, al valor devuelto por *malloc()* o *calloc()* debemos "castearlo" a un tipo de dato en particular. Recordemos que los caracteres siempre terminan en *\0* (cero binario, *null*), por lo tanto, buffer puede apuntar a un arreglo de 14999 caracteres más un *null* final. Si buffer es distinto de *null*, implica que la asignación fue exitosa y por lo tanto, puedo hacer uso de *buffer* y su posterior liberación de memoria (*free(buffer)*). Cada vez que se solicita memoria y ésta es otorgada, luego de usarla debemos liberarla usando *free()*, sino nuestro programa colapsará por falta de memoria. Debemos chequear cada asignación de memoria. Generalmente el argumento de *calloc()* y *malloc()* son datos provenientes del exterior (ingresos por teclado, etc.) o bien resultado de cálculos internos dentro del programa. Debemos usar *calloc(), malloc()*, memoria dinámica, cuando necesitamos asignar memoria y la cantidad a asignar sólo la conocemos en tiempo de ejecución (*runtime*).

Apuntadores y funciones

También se pueden apuntar funciones. Supongamos que tenemos la función *void funcion();* , es posible crear un apuntador a dicha función llamado *pfuncion* y ejecutar a *funcion()* a través del uso del puntero, ejemplo:

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

void funcion();
```

```
int main() {
   void (*pfuncion)(); // declaro puntero a función de tipo void sin
                       // argumentos
   pfuncion = funcion; // pfuncion apunta a la función funcion()
   pfuncion(); // ejecuto funcion() utilizando el puntero pfuncion
   return 0;
}
void funcion() {
  printf("hola funcion()");
}
```

Bibliografía

- Kernighan, Brian W.; Ritchie, Dennis M., “El lenguaje de programación C” Segunda Edición, Prentice-Hall Hispanoamericana S.A., AT & T Bell Laboratories, Murray Hill, New Jersey, 1991, ISBN-968-880-205-0 (inglés), ISBN 0-13-110362-8 (español)