75.41 Algoritmos y Programación II
Lenguaje C Avanzado

Dr. Mariano Méndez [1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

12 de abril de 2020

1. Introducción

Una sólida comprensión del tipo de dato puntero y la habilidad de usarlos en forma efectiva, separa a un programador novato de C de uno experimentado. Los punteros le dan al lenguaje C una gran flexibilidad. Ellos proveen un importante soporte para el manejo de la memoria dinámica, están ligados muy de cerca con la notación de vectores, y cuando se utilizan para apuntar a funciones proveen una nueva dimensión al control del flujo de un programa.

Un puntero no es más que una variable que almacena una dirección de una posición de memoria. Si bien este concepto puede complicarse al sumarle los operadores, su aritmética y la intrincada notación que en su uso se deriva, éste no tendría que ser el caso, si se asientan unas sólidas bases para su utilización.

1.1. Estructura de la Memoria de un Programa

Para poder entender de lleno el concepto de puntero, se debe conocer y comprender cómo se estructura y clasifica la memoria de un programa a través de sus distintos estadíos de su ciclo de vida.

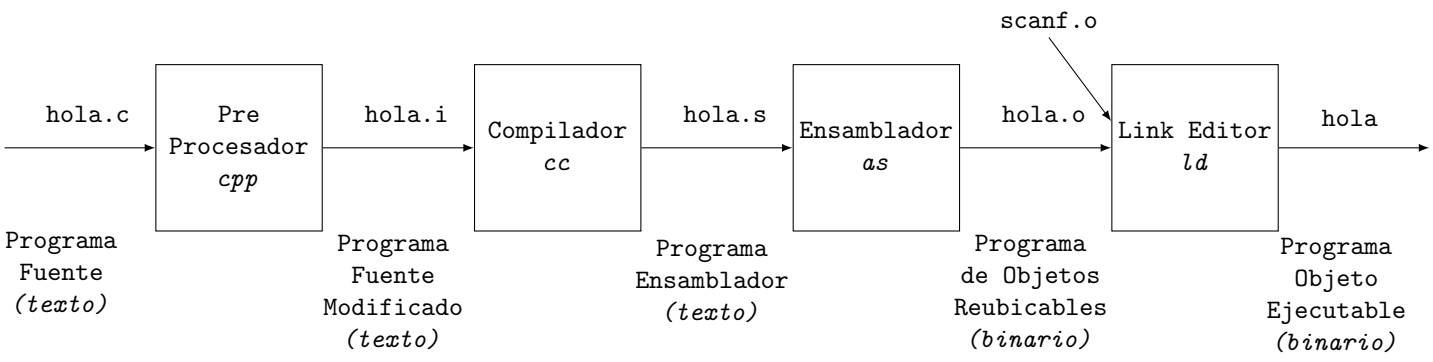
Un programa puede encontrarse en varias etapas:

- Edición
- Compilación
- Ejecución

En cada una de estas etapas el programa tiene una estructura distinta. En edición el programa sólo es un archivo de texto editado en el lenguaje seleccionado por el programador, esto se denomina *código fuente*.

programador

Una vez que el código fuente del programa está listo, se utiliza un conjunto de programas para traducir el código fuente en un lenguaje que una computadora pueda ejecutar. Este programa se denomina **compilador**.



El compilador tiene la tarea de traducir el código fuente de un programa en una estructura que una computadora pueda ejecutar. El proceso de compilación no es sencillo. Está compuesto por varias etapas, en la cual intervienen diversos componentes del compilador. Entre los más importantes están:

- El Lexer
- El Parser
- El Generador de Código Intermedio
- El Generador de Código Objeto

1.1.1. El Lexer: Análisis Lexicógrafo

En esta primera etapa el Lexer es el encargado de leer el archivo de código fuente y generar unidades atómicas. Para ello lee el programa de izquierda a derecha y agrupa en **componentes léxicos** (tókens), que son secuencias de caracteres que tienen un significado, extrae las palabras. Además, todos los espacios en blanco, líneas en blanco, comentarios y demás información innecesaria se elimina del programa. También se comprueba que los símbolos del lenguaje (palabras clave, operadores, etc.) se han escrito correctamente.

1.1.2. El Parser: Análisis Sintáctico y Semántico

En esta fase los caracteres o componentes léxicos se agrupan jerárquicamente en frases gramaticales que el compilador utiliza para sintetizar la salida. Se comprueba si lo obtenido de la fase anterior es sintácticamente correcto (obedece a la gramática del lenguaje). Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico.

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los *tipos* para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje. Por ejemplo, las definiciones de muchos lenguajes de programación requieren que el compilador indique un error cada vez que se use un número real como índice de una matriz. Sin embargo, la especificación del lenguaje puede imponer restricciones a los operandos, por ejemplo, cuando un operador aritmético binario se aplica a un número entero y a un número real. Revisa, también, que los arreglos tengan definido el tamaño correcto.

1.1.3. Generación de de Código Intermedio

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta que debe tener dos propiedades importantes: ser fácil de producir y ser fácil de traducir al programa objeto.

La representación intermedia puede tener diversas formas. Existe una llamada código de tres direcciones que es como el lenguaje ensamblador de una máquina en la que cada posición de memoria puede actuar como un registro.

Código Fuente

```
short main(){
    short i=5, j=7, k;
    k = i + j;
    return k;
}
```

Código Intermedio

```
push   %rbp
mov    %rsp,%rbp
movw   $0x5,-0x6(%rbp)
movw   $0x7,-0x4(%rbp)
movzwl -0x6(%rbp),%edx
movzwl -0x4(%rbp),%eax
add    %edx,%eax
mov    %ax,-0x2(%rbp)
movzwl -0x2(%rbp),%eax
pop    %rbp
retq
```

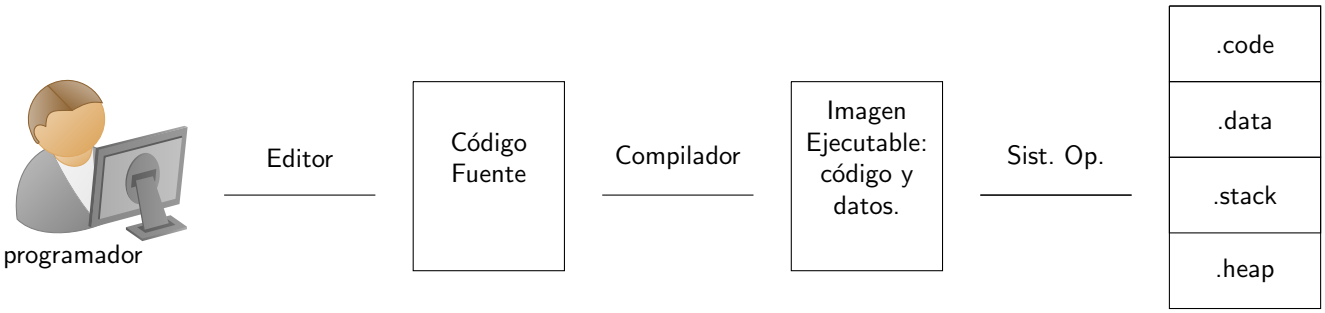
Código Objeto

```
5fa: 55
5fb: 48 89 e5
5fe: 66 c7 45 fa 05 00
604: 66 c7 45 fc 07 00
60a: 0f b7 55 fa
60e: 0f b7 45 fc
612: 01 d0
614: 66 89 45 fe
618: 0f b7 45 fe
61c: 5d
61d: c3
```

Una vez compilado el código fuente del programa, gracias a la acción del compilador y el linkeditor, toma forma como un archivo en formato objeto, el cual es posible ejecutar en una computadora (comando: objdump -d). Por último, este programa objeto que está guardado en un dispositivo de almacenamiento es capaz de cobrar vida, al ser ejecutadas sus instrucciones por el procesador de la computadora. En esta etapa pasa a ser una entidad con "vida"(dinámica) la cual se divide en ciertas secciones. Éstas son:

- Código (.code): En esta sección del programa se almacenan el código del mismo, una vez traducido por el compilador.
- Datos (.data): En esta sección se almacena los datos de los valores literales y valores globales del programa.
- Pila de ejecución (execution stack o .stack): no existe hasta que el programa es ejecutado.
- Montículo (.heap): no existe hasta que el programa es ejecutado.

Cada una de estas secciones guarda distinto tipo de información perteneciente al programa.



En la sección **.code** se almacena todo el código fuente del programa. En la sección **.data** se almacenan las constantes globales, variables externas y valores literales del programa. En el **.stack** de ejecución se almacenan todas las variables locales de las funciones que se encuentran en ejecución. La última sección, **.heap** está reservada a la denominada **memoria dinámica**, la misma es memoria que puede estar disponible para ser utilizada por el programador en tiempo de ejecución. La memoria dinámica no es reservada por el compilador, sino que por lo contrario es manejada por el programador, en tiempo de ejecución del programa. Una de las implicaciones de esto es que no se sepa a priori la cantidad a ser reservada, sino hasta el momento en que es necesario reservarla.

Normalmente estas tres secciones (.code, .data. y .stack) son controladas exclusivamente por el compilador, el programador no tiene forma de alterar estas secciones. El compilador es el que estructura la información que debe ir en esas secciones sin que el programador pueda hacer ninguna otra modificación tras haber compilado el programa. Para que un programa compilado, que está almacenado en un disco, pueda ser ejecutado debe ser cargado directamente en la memoria RAM. De esta tarea se encarga el **sistema operativo**.

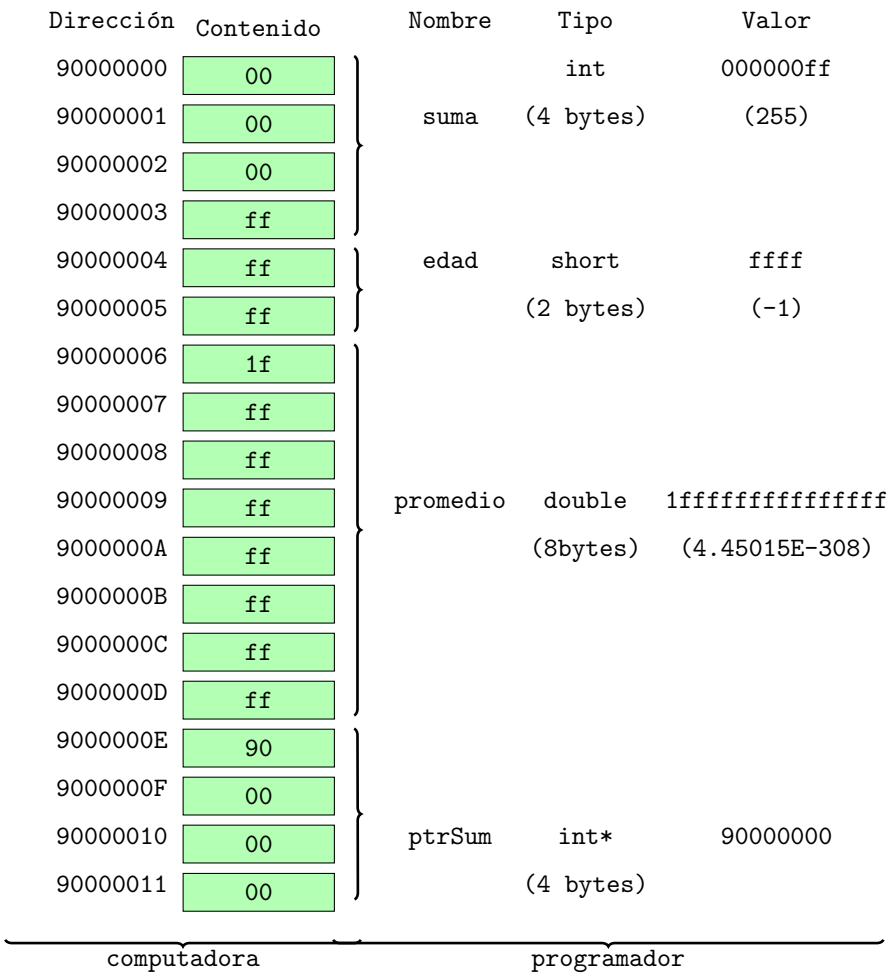
El mismo lee la estructura del programa compilado y teniendo en cuenta la memoria RAM, reserva espacio para las cuatro secciones (.code, .data, .stack y .heap).

2. Punteros en C

Uno de los aspectos que complican la utilización de punteros es su notación en C. Por ello se repasarán algunas equivalencias que suelen traer problemas.

2.1. La Memoria

La memoria es parte del hardware de la computadora, su funcionamiento es por demás complejo. Es importante poder entender cómo está estructurada y también cómo funciona. Para ello se hará uso de una herramienta que ya se ha utilizado, la abstracción. Cuyo objetivo es simplificar la compleja estructura real de la memoria física de la computadora y plasmarla en un concepto que simplifique su entendimiento. La memoria de la computadora puede ser vista como un arreglo de celdas, en la que cada una posee un nombre asociado. Este nombre es un número, la longitud del mismo depende de la arquitectura de la computadora, en este caso podría ser 32 bits o 64 bits. Cada una de estas celdas posee un número consecutivo a su vecina, iniciando de 0 hasta la cantidad que la computadora posea.



Existen programas que permiten ver el contenido de la memoria o de un archivo en un formato bastante parecido al que se describe en la figura anterior. En linux por ejemplo uno de esos programas se llama xxd, para más información *man xxd* :

```
Dirección                 Contenido                 Representación

00000000:  2247 6f6f 6420 6465 7369 676e 2061 6464  "Good design add
00000010:  7320 7661 6c75 6520 6661 7374 6572 2074  s value faster t
00000020:  6861 6e20 6974 2061 6464 7320 636f 7374  han it adds cost
00000030:  2e22 202d 2054 686f 6d61 7320 432e 2047  . Thomas C. G
00000040:  616c 6520 0a22 4920 696e 7665 6e74 6564  ale ."I invented
00000050:  2074 6865 2074 6572 6d20 274f 626a 6563  the term 'Objec
00000060:  742d 4f72 6965 6e74 6564 272c 2061 6e64  t-Oriented', and
00000070:  2049 2063 616e 2074 656c 6c20 796f 7520  I can tell you
00000080:  4920 6469 6420 6e6f 7420 6861 7665 2043  I did not have C
00000090:  2b2b 2069 6e20 6d69 6e64 2e22 202d 2041  ++ in mind. A
000000a0:  6c61 6e20 4b61 790a 4f6e 2061 6464 696e  lan Kay.On addin
000000b0:  6720 7072 6f67 7261 6d6d 6572 7320 746f  g programmers to
000000c0:  2061 206c 6174 652d 7275 6e6e 696e 6720  a late-running
000000d0:  7072 6f6a 6563 743a 2022 4e69 6e65 2070  project: "Nine p
000000e0:  656f 706c 6520 6361 6ee2 8099 7420 6d61  eople can...t ma
000000f0:  6b65 2061 2062 6162 7920 696e 2061 206d  ke a baby in a m
00000100:  6f6e 7468 2e22 202d 2046 7265 6420 4272  onth. Fred Br
00000110:  6f6f 6b73 200a 2244 656c 6574 6564 2063  ooks ."Deleted c
00000120:  6f64 6520 6973 2064 6562 7567 6765 6420  ode is debugged
00000130:  636f 6465 2e22 202d 204a 6566 6620 5369  code. Jeff Si
00000140:  636b 656c 200a 2254 6865 2063 6f6d 7075  ckel ."The compu
00000150:  7469 6e67 2073 6369 656e 7469 7374 2773  ting scientist's
00000160:  206d 6169 6e20 6368 616c 6c65 6e67 6520  main challenge
00000170:  6973 206e 6f74 2074 6f20 6765 7420 636f  is not to get co
```

2.2. Punteros y Notación de arreglos

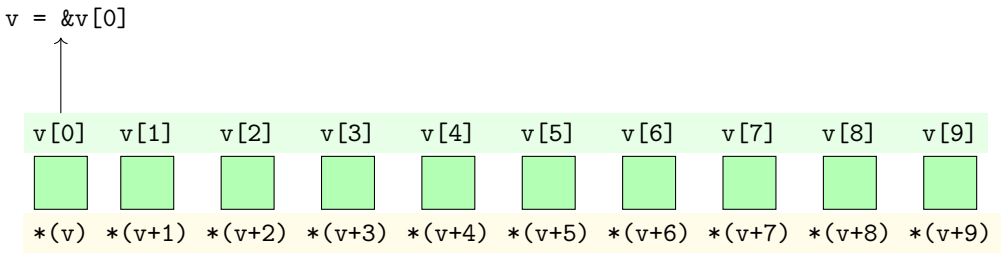
En el lenguaje C existe una equivalencia entre la notación para acceder a un elemento de un arreglo y la aritmética de punteros. Cabe recordar que un arreglo es un tipo de dato estructurado en el cual los elementos del mismo se

encuentran físicamente uno después del otro en la memoria física de la computadora, el nombre de un arreglo es justamente un puntero al primer elemento del mismo, sea *array* un arreglo, las siguientes expresiones son equivalentes, ver la Figura 1

Notación Arreglo	Notación de Puntero
array[0]	*array
array[1]	*(array+1)
array[2]	*(array+2)
array[3]	*(array+3)
array[n]	*(array+n)

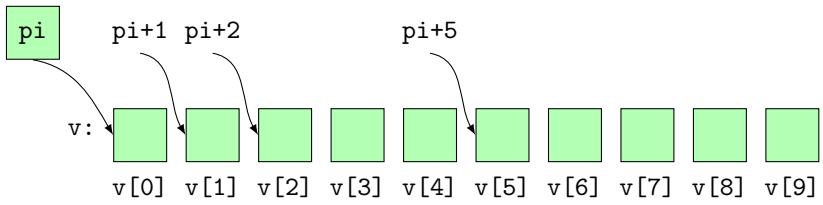
Figura 1: equivalencia notación arrego-puntrero

Teniendo en cuenta que array = &array[0], en la figura se muestra esta equivalencia en la siguiente declaración int v[10] :



Teniendo en cuenta que si **int v[10]** esta equivalencia v=&v[0], entonces :

```
int v[10];
int *pi;
pi=v;
```

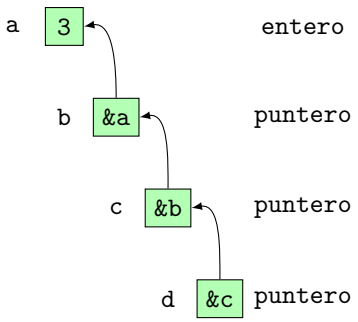


¿Qué pasa con un arreglo de dos dimensiones? Lo mismo pero un poco más complicado.

2.3. Múltiples Indirecciones

En el lenguaje de programación C se pueden declarar punteros a punteros, lo cual se denomina, múltiple indirección. Considérese, el siguiente ejemplo de código C:

```
1 int      a = 3;
2 int     *b = &a;
3 int    **c = &b;
4 int   ***d = &c
```



Teniendo en cuenta las definiciones anteriores y el esquema de cómo están conformadas las variables:

```
*d == c;
**d == *c == b;
***d == **c == *b == a == 3

```

3. Memoria Dinámica

La memoria dinámica es aquella que será reservada y utilizada únicamente en tiempo de ejecución. La memoria dinámica es reservada en la sección **.heap** o en el heap del programa. Para utilizar este tipo de memoria el programador debe hacerlo de forma explícita solicitándoselo al sistema operativo. Para ello en el lenguaje de programación C se utilizan dos funciones de la biblioteca estándar llamadas:

- **malloc()**
- **free()**

La primera es utilizada para decirle al sistema operativo que se desea utilizar memoria del heap (dinámica) y la segunda es utilizada par liberar dicha memoria una vez utilizada. Estas funciones se encuentran en stdlib.h, pueden encontrarse consultando la documentación de linux (`$ man malloc`)

3.1. malloc()

La función **malloc()** reserva *size* bytes y devuelve un puntero a la memoria reservada. La memoria no está inicializada. Si el valor de size es 0 ésta puede devolver, NULL o un valor a un puntero único que posteriormente podrá ser pasado a la función free() para ser liberado.

```
#include <stdlib.h>
void *malloc(size_t size);
```

Los pasos para utilizar memoria dinámica son dos:

1. en primer lugar se reserva utilizando *malloc()* y;
2. cuando se termina de utilizar y no se necesita más tener ese espacio de memoria reservado, se libera usando *free()*.

3.1.1. Ejemplo

El siguiente ejemplo va a mostrar el funcionamiento práctico de ambas funciones (malloc() y free()) a partir de un sencillo ejemplo de código. La variable *ptr_entero*, se declara y vive durante la ejecución del programa en el **stack** del mismo :

`ptr_entero;`

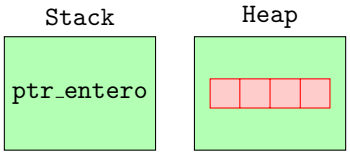
Stack

A continuación se utilizará la función **malloc()** para reservar *memoria dinámica* para un entero. Para ello, malloc() reserva la cantidad de bytes que se le pasa como parámetro. Para que el código fuente sea agnóstico de una arquitectura determinada (a que los tamaños de los tipos de datos varían de arquitectura en arquitectura) se utiliza una función que dado un tipo de dato devuelve la cantidad de bytes que éste ocupa según la arquitectura en la cual se esté ejecutando el programa, esta función se llama **sizeof()**

```
ptr_entero = malloc( sizeof( int ) );
```

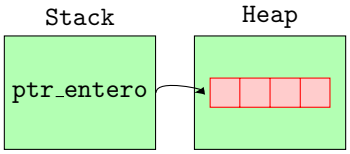
Como primer paso, malloc() reserva la cantidad de bytes de memoria devuelta por sizeof(int). Esta memoria pertenece al **Heap**. Una vez reservada la cantidad de bytes, la misma función, devuelve la dirección del primer byte de memoria:

```
ptr_entero = malloc( sizeof( int ) );
```



Una vez que esta operación se realizó, la asignación se encarga de asignar a la variable puntero la dirección de memoria que malloc() se encargó de reservar en el heap.

```
ptr_entero = malloc( sizeof( int ) );
```



Hay que tener en cuenta que si esta relación entre la memoria reservada en el heap y su dirección conocida desde el stack se pierde, la memoria pasa a ser memoria **no alcanzable** o **perdida** pues nadie sabe como llegar a ella.

En este punto vale la pena explicar un concepto interesante sobre la memoria que se encuentra en el *stack* y en el *heap*. Cuando uno crea una **variable local** en una función, la misma es creada en la sección llamada **.stack** del programa en ejecución, dentro del ámbito de memoria de esa función. Cuando la misma termina su ejecución, el ámbito se destruye y por ende todas las variables locales del mismo.

```c
int main (){

        int valor_entero;
        int ptr_entero;


        valor_entero = 5;
        ptr_entero = malloc( sizeof(int) );

        (* ptr_entero ) = 10;

        free( ptr_entero );

        return 0;
}

```

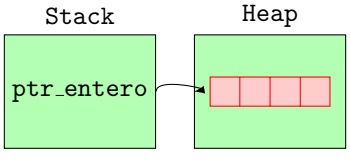
link al código

3.2. free()

La funcion **free()** libera espacio de memoria apuntado por el puntero (*ptr*) que debe haber sido devuelto previamente por una llamada a la función *malloc()*, *calloc()* o *realloc()*. De otra forma o si free(ptr) ha sido ya ejecutado anteriormente, **ocurrirá un comportamiento no definido**. Si ptr es NULL, la operación no se realiza.

Básicamente si anteriormente se reservó memoria dinámica para un entero, la situación, si todo fue bien, debería ser:

```
ptr_entero = malloc( sizeof( int ) );
```

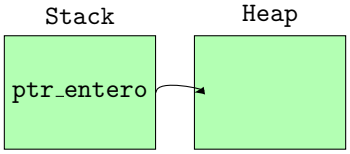


para liberar esta memoria previamente reservada con **malloc()** se debe utilizar **free()**

```
#include <stdlib.h>
void free(void *ptr);
```

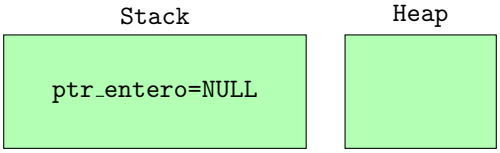
Para ello free(), en primer lugar libera la porción de memoria dinámica, reservada con malloc():

```
ptr_entero = malloc( sizeof( int ) );
```



Posteriormente inicializa al puntero que apuntaba a esa dirección con NULL:

```
ptr_entero = malloc( sizeof( int ) );
```



3.3. Ejemplos Sencillos

A continuación se mostrará a modo de ejemplo un pequeño programa escrito en lenguaje C que utiliza memoria dinámica:

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

int main(){
        int* pi;

        pi = (int*) malloc( sizeof(int) );

        if(pi == NULL){
                fprintf(stderr,"ocurrió un error: memoria agotada");
                return -1;
        }

        // procesamiento

        *pi=5;

        free(pi);
        return 0;
}
```

link a ctutor

Este programa declara un puntero a un entero, reserva memoria para él, valida que efectivamente la operación de reserva de memoria no haya fallado, realiza algún proceso y posteriormente libera el espacio de memoria reservado.

Otro ejemplo un poco más interesante es cuando se quiere reservar varias variables contiguas (un arreglo) en forma dinámica:

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

int main(){
    int* pi;
    int size=5;

    pi = (int*)malloc( size * sizeof(int) );                    // <-creación
```

```
9
10      if(pi == NULL){
11          fprintf(stderr,"ocurrió un error: memoria agotada");
12          return -1;
13      }
14
15      // procesamiento
16
17          for(int i=0;i<size;i++) pi[i]=0;
18
19
20      free(pi);                                                    // <-destrucción
21      return 0;
22  }
```

link a ctutor

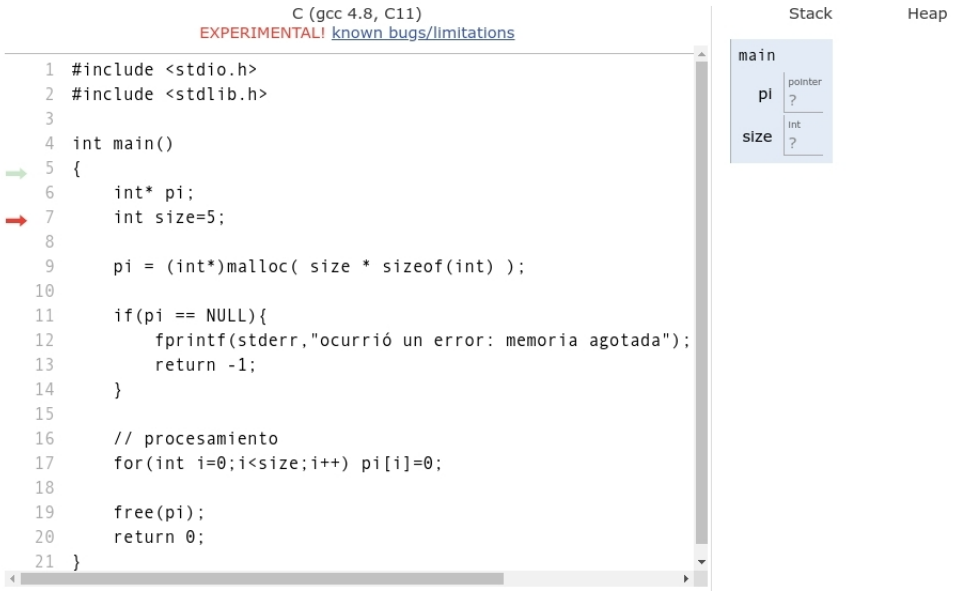


Figura 2

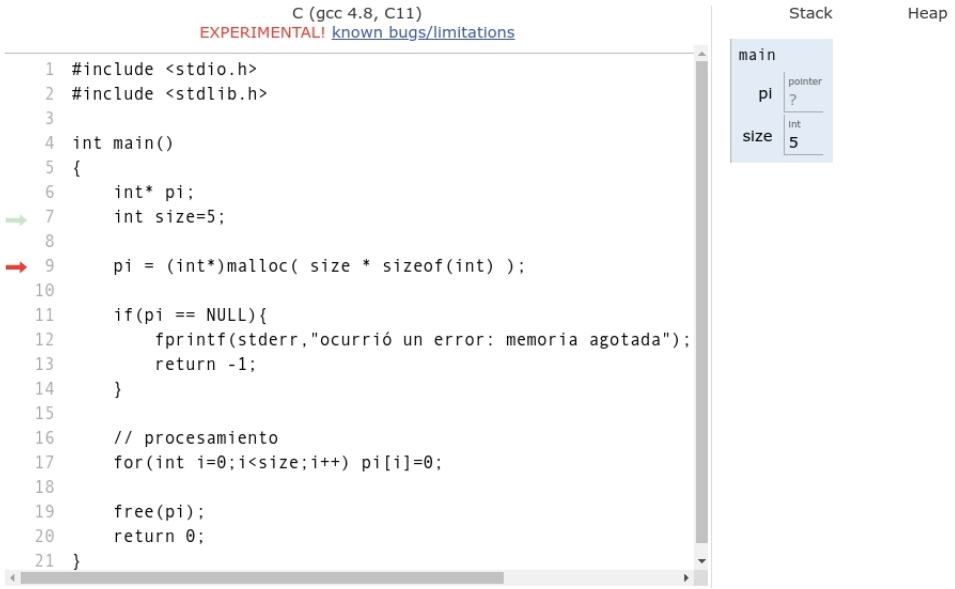


Figura 3

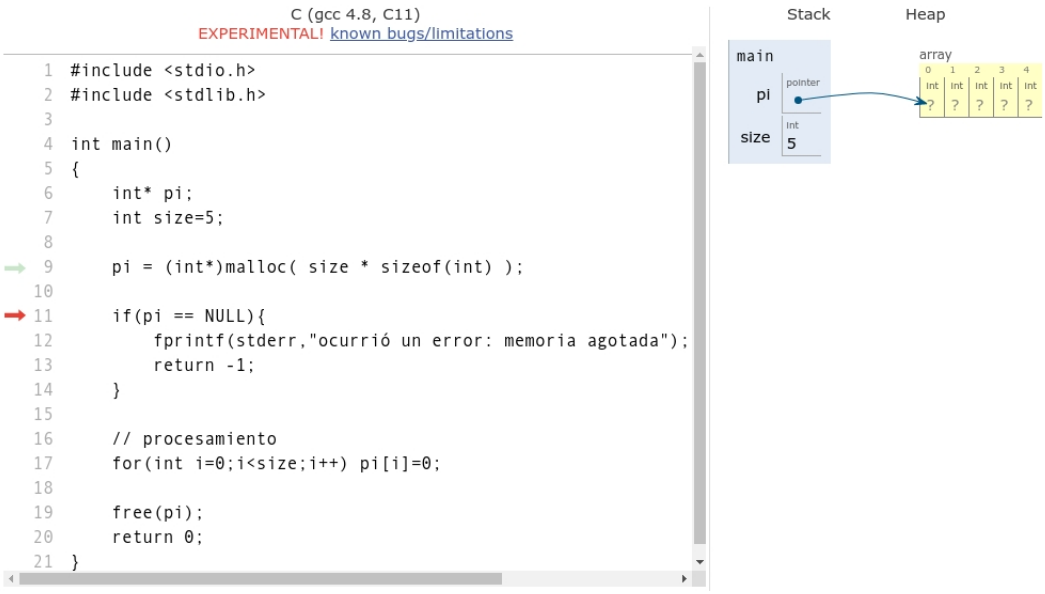


Figura 4

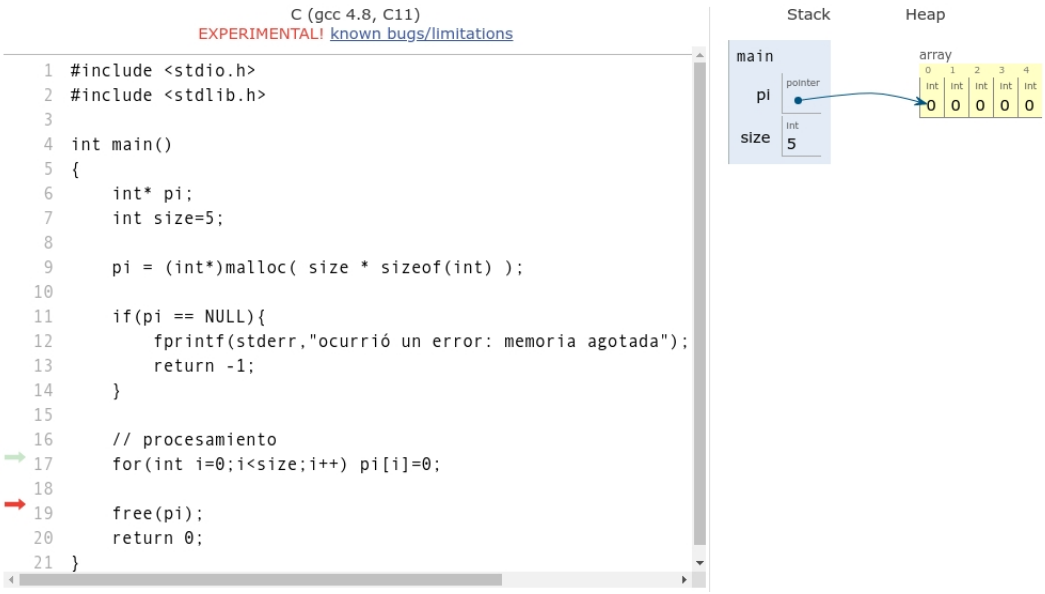


Figura 5

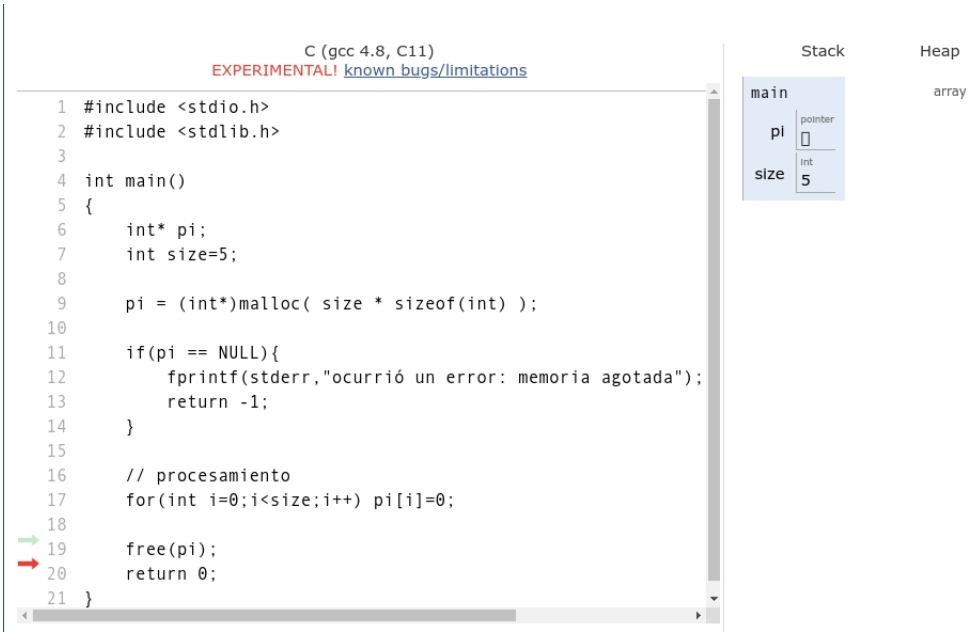


Figura 6

Ojo con este error clásico Un error muy frecuente es devolver en el *return* de la función un puntero a una variable local, por ejemplo:

```c
#include <stdio.h>

int* suma (int un_sumando, int otro_sumando){
        int resultado;

        resultado= un_sumando + otro_sumando;

        return &resultado;                                    // <------incorrecto!

}


int main (){

        int operacion,un_valor,otro_valor;
        &operacion=suma(un_valor,otro_valor);
        return 0;
}
```

link a ctutor

Claramente la dirección de la variable **resultado** únicamente está vigente dentro del ámbito de la función **suma**, cuando termina la ejecución de la función **suma**, todas las variables locales dejan de existir. ¿Cuál sería la forma correcta de implementar tal función? Con memoria del heap.

```c
#include <stdio.h>

int* suma (int un_sumando, int otro_sumando){
        int* resultado;

        resultado=malloc( sizeof(int) );

        *resultado=un_sumando + otro_sumando;

        return resultado;                                     // <------correcto!

}

int main (){

        int     *operacion;
```

```
        int un_valor=1,otro_valor=1;
        operacion=suma(un_valor,otro_valor);
        free(operacion)
        return 0;
}
```

link a ctutor

Más complejos . Existe una función de la biblioteca de strings que, dado un string, devuelve una copia del mismo. Es claro que para ello se debe utilizar memoria dinámica. La función en cuestión es **strdup()** para hacer un poco más complejo no se podrá utilizar ninguna función de *string.h*. A continuación se realizará una versión de la misma:

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

char* strdup(const char* un_string){
        int longitud;
        int i;
        char* str_copia;

        i = 0;
        longitud = 0;
        while ( un_string[longitud] != '\0' )
                longitud++;

        str_copia = malloc( (longitud+1) * sizeof(char) );

        if( str_copia == NULL){
                return NULL;
        }

        i = 0;
        while( i< longitud ){
                str_copia[i] = un_string[i];
                i++;
        }
        str_copia[longitud] = '\0';
        return str_copia;
}

int main(){
        char* str;

        str = strdup("Hola Mundo");
        free(str);
        return 0;
}
```

link a ctutor

3.4. calloc()

La función **calloc()** reserva memoria para cualquier arreglo de *nmemp* elementos de tamaño *size bytes* cada uno y devuelve un puntero al inicio de la memoria reservada. Toda la memoria se setea con ceros. Si el valor de size es 0, ésta puede devolver NULL o un valor a un puntero único que posteriormente podrá ser pasado a la función free() para ser liberado. El producto de *nmem* y *size* determinan el tamaño del bloque de memoria reservado.

```
#include <stdlib.h>
void* calloc (size_t n, size_t size);
```

3.5. realloc()

La función **realloc()** modifica el tamaño del bloque de memoria apuntado por *ptr* en *size* bytes. El contenido del bloque de memoria permanecerá sin cambios desde el inicio del mismo hasta el mínino entre el viejo y nuevo tamaño. Si el nuevo tamaño del bloque es mayor que el tamaño anterior, la memoria añadida no se encuentra inicializada en ningún valor. Si *ptr* es NULL, entonces la llamada es equivalente a malloc(size) para cualquier valor de *size*. Si *size* es cero y *ptr* no es NULL entonces la llamada es equivalente a free(ptr).

```c
#include <stdlib.h>
void* realloc (void* ptr, size_t size);
```

3.6. reallocarray()

La función **realloacarray** cambia el tamaño del bloque de memoria apuntado por *ptr* para que tenga el tamaño suficiente para un arreglo de *nmemb* elementos de tamaño *size* bytes. Es equivalente a hacer la llamada:

```c
realloc(ptr, nmemb * size);
```

De todas formas reallocarray falla de forma segura en el caso de que la multiplicación de overflow. Ya que si esto sucede realloccarray() devuelve NULL, seteando errno en ENOMEM y deja el bloque original sin cambios.

```c
#include <stdlib.h>
void* reallocarray(void* ptr, size_t nmemb, size_t size);
```

3.7. Teorema Fundamental de la Memoria Dinámica

Es muy importante entender que la memoria dinámica es **controlada pura y exclusivamente por el programador**, él decide cuando pedirla y él debe decidir cuando liberarla. El hecho de no liberar memoria dinámica implica **SERIOS PROBLEMAS** en el **COMPORTAMIENTO del programa**. Para ello existe un teorema similar al teorema de conservación de la energía en física, el teorema de Mendez de la memoria dinámica:

"**La memoria dinámica no se conserva! Siempre que se crea debe destruirse!**"

Es imprescindible recordar que el manejo de la memoria dinámica es responsabilidad del programador y el mismo debe crearla y destruirla, haciendo que siempre haya la misma cantidad de memoria dinámica reservada entre el inicio y el fin del programa... esta cantidad es **0 bytes** de memoria dinámica. La regla es que la memoria dinámica reservada debe liberarse cuando ya no se necesite.

Existe una herramienta que permite controlar que un programa no esté perdiendo memoria dinámica, e infringiendo el teorema anterior. Ésta herramienta se llama Valgrind. A continuación se mostrará su funcionamiento.

3.7.1. Ejemplo sin Pérdida

El primero es un ejemplo sencillo en el cual sería imposible tener pérdida de memoria dinámica, pues no se hace uso de la misma:

```c
#include <stdio.h>

int main(){

        int una_variable;

        una_variable=2;
        printf("el valor de x es = %i\n", una_variable);
        return 0;

}
```

La línea de compilación debe incluir la opción de **-g** para que se mantenga la información para debugging:

```
gcc -g -O0 sin_perdida.c -o executable
```

A continuación se ejecuta el comando correspondiente a detectar pérdida de memoria dinámica (memory leaks):

```
darthmendez@universe: valgrind:$valgrind --leak-check=full executable
==27427== Memcheck, a memory error detector
==27427== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==27427== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==27427== Command: executable
==27427==
el valor de x es = 2
==27427==
==27427== HEAP SUMMARY:
==27427==     in use at exit: 0 bytes in 0 blocks
==27427==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==27427==
==27427== All heap blocks were freed -- no leaks are possible
==27427==
==27427== For counts of detected and suppressed errors, rerun with: -v
==27427== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

En este caso se ve cómo no hay memoria que se pierde.

3.7.2. Ejemplo con Pérdida

En este ejemplo sencillo se incurre en dos errores comunes, el código:

```c
#include <stdlib.h>

void f(void){
        int* x = malloc(10 * sizeof(int));
        x[10] = 0;        // problem 1
}                       // problem 2

int main(void){
        f();
        return 0;
}
```

La línea de compilación debe incluir la opción de -g para que se mantenga la información para debugging:

```
gcc -g -O0 con_perdida.c -o executable
```

A continuación se ejecuta el comando correspondiente a detectar pérdida de memoria dinámica (memory leaks):

```
darthmendez@universe:$valgrind --leak-check=full ./executable
==31920== Memcheck, a memory error detector
==31920== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31920== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==31920== Command: ./executable
==31920==
==31920== Invalid write of size 4
==31920==    at 0x108668: f (con_perdida.c:22)
==31920==    by 0x108679: main (con_perdida.c:26)
==31920==  Address 0x522d068 is 0 bytes after a block of size 40 alloc'd
==31920==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==31920==    by 0x10865B: f (con_perdida.c:21)
==31920==    by 0x108679: main (con_perdida.c:26)
==31920==
==31920==
==31920== HEAP SUMMARY:
==31920==     in use at exit: 40 bytes in 1 blocks
==31920==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==31920==
==31920== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==31920==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==31920==    by 0x10865B: f (con_perdida.c:21)
==31920==    by 0x108679: main (con_perdida.c:26)
==31920==
==31920== LEAK SUMMARY:
==31920==    definitely lost: 40 bytes in 1 blocks
==31920==    indirectly lost: 0 bytes in 0 blocks
==31920==      possibly lost: 0 bytes in 0 blocks
==31920==    still reachable: 0 bytes in 0 blocks
==31920==         suppressed: 0 bytes in 0 blocks
==31920==
==31920== For counts of detected and suppressed errors, rerun with: -v
==31920== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

En este caso se ve cómo si se reportan 2 errores. El primero es una escritura fuera del rango del vector, x[10] = 0, ver definición de x[]:

```
==31920== Invalid write of size 4
==31920==    at 0x108668: f (con_perdida.c:22)
==31920==    by 0x108679: main (con_perdida.c:26)
==31920==  Address 0x522d068 is 0 bytes after a block of size 40 alloc'd
==31920==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==31920==    by 0x10865B: f (con_perdida.c:21)
==31920==    by 0x108679: main (con_perdida.c:26)
==31920==
```

y la segunda es la pérdida de 40 bytes de memoria del vector reservado con malloc() que no se libera con free() infringiendo el teorema fundamental de la memoria dinámica.

Recordar que para ejecutar valgrind en modo chequeo de memoria, se escribe por línea de comando:

```
$valgrind --leak-check=full ./executable
```

3.8. El Valor Nulo

Cuando se declara un puntero, se dice que éste toma un valor cualquiera que no está inicializado. Es necesario garantizar la comparación entre dos punteros de cualquier tipo en la igualdad. Por ello la macro **NULL** define la constante *puntero a un valor nulo*. El estándar de C11 en el apartado 6.3.2.3, dice:

> *An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.*

Es decir que el valor **NULL** corresponde, ya sea a *(void *)0* o el valor *0*, dependiendo de la implementación que haga el compilador. Se utiliza un puntero con valor nulo cuando la variable puntero debe apuntar a diversas direcciones de memoria a lo largo de un programa, el valor nulo indica que el mismo no se encuentra apuntando a una dirección válida.

4. C Avanzado

A continuación se desarrollaran algunos conceptos avanzados del lenguaje de programación C, necesarios para poder implementar correctamente la mayoría de las estructuras de datos.

4.1. Aritmética de Punteros

Los punteros son variables que, al igual que otros tipo de variables, tienen definidos operadores sobre ellas y sus valores. Los operadores más utilizados con punteros son:

- **Operador de Dirección (&)**: Éste permite acceder a la dirección de memoria de una variable.
- **Operador de Indirección (*)**: Además de que permite declarar un tipo de dato puntero, también permite ver el VALOR que está en la dirección asignada.
- **Incrementos (++)** y **Decrementos (–)**: Se puede usar un puntero como si de un array se tratase, por eso mismo permite estos operadores.

```c
//Dadas:
int x[100], b, *pa, *pb;

x[50]=10;               // se asigna el valor de 10, al array #50
pa=&x[50];              // se asigna al puntero pa, la direccion de memoria que tiene x[50]



b = *pa+1;              // esto es igual a: b=x[50]+1; => Su valor seria igual a 11.


b = *(pa+1);    // Esto primero pasa a la siguiente direccion de memoria
                // y luego lo referencia
                //El resultado es: b = x[51];

pb = &x[10];    //al puntero pb se le asigna la direccion de x[10]

*pb = 0;        // Al valor que tiene el puntero se le asigna 0
                // Esto es igual que decir: x[10] = 0

*pb += 2;       // El valor del puntero se incrementa en dos unidades,
                // es decir x[10] += 2

(*pb)--;        // El valor del puntero se decrementa en una unidad.

x[0] = *pb--;   // A x[0] se le pasa el valor de x[10] y el puntero pb,
                // pasa a apuntar a x[9]
                //recordar, que -- es post-decremento, primero asignará y luego restará.
```

4.2. Punteros a Funciones

Un puntero a una función no es más que la dirección de memoria donde reside una determinada función en un programa en C. Además permite ejecutar a la función desde cualquier parte del programa. Los punteros a funciones en C pueden:

1. ser elementos de un vector.
2. ser pasados como parámetros a una función.
3. ser devueltos por una función.

Lo más difícil de los punteros a funciones es entender su declaración: *tipo_de_retorno (* nombre_función) (tipo_argumento_uno, tipo_argumento_2,..., tipo_argumento_n)*

Por ejemplo si se quiere definir una función que no devuelve ningún valor pero recibe dos parámetros enteros eso puede hacerse de la siguiente forma:

```
void (* mi_funcion) (int , int)
```

Es muy importante no confundirse con:

```
void * mi_funcion (int , int);
```

Esta declaración es válida en C, lo que esta definiendo es el prototipo de una función que devuelve un puntero a void.

4.2.1. Un Ejemplo

Para ello vamos a ver un ejemplo en concreto:

```
#include <stdio.h>
// declaracion de una funcion normal con un parámetro entero
// y que devuelve un valor de tipo void.
void fun(int a) {
    printf("Value of a is %d\n", a);
}

int main(){
    // fun_ptr es un puntero a una funcion fun()
    void (*fun_ptr)(int) = &fun;

    /* la línea de arriva es equivalente a las siguientes dos líneas
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invocando fun() usando fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

1. A diferencia de los punteros normales, los *punteros a funciones* apuntan a código y no a datos, normalmente apuntan a la primera instrucción ejecutable de la función.
2. A diferencia que los punteros normales, con los *punteros a funciones* no se debe reservar-liberar espacio de memoria.
3. El nombre de la función también puede ser utilizado para obtener la dirección de la misma. A su vez también puede ser desreferenciada sin la utilización del *.

```
void (*fun_ptr)(int) = fun;  // sin  &

fun_ptr(10);  // sin *

```

4. Al igual que los punteros normales, se puede tener un arreglo de punteros a funciones.

4.2.2. Otro Ejemplo

A continuación se muestra otro ejemplo de uso de punteros a funciones:

```
void intercambiar_enteros (int * un_puntero_entero, int * otro_puntero_entero){
        int aux=*un_puntero_entero;
        *un_puntero_entero=*otro_puntero_entero;
        *otro_puntero_entero=aux;
}
```

Esta función intercambia dos valores enteros. ¿Cómo se define una variable que sea un puntero a esa función?. En C existe una forma de hacerlo, y es la siguiente:

```
#include<stdio.h>

void intercambiar_enteros (int* un_puntero_entero, int* otro_puntero_entero){
        int aux=*un_puntero_entero;
        *un_puntero_entero=*otro_puntero_entero;
        *otro_puntero_entero=aux;
}

int main (){

        int x,y;
        void (*swap)(int *,int *) ;
        x=1;
        y=0;
        swap= intercambiar;
        printf("x= %i , y= %i \n",x,y);
        swap(&x,&y);
        printf("x= %i , y= %i \n",x,y);
}
```

link a ctutor

4.3. Conversión Explícita de Datos: Casteo

El lenguaje de programación C tiene un mecanismo por el cual un programador puede convertir una expresión, que al ser evaluada corresponde a un tipo de dato determinado, como si perteneciera a otro tipo de dato. Para ello se utiliza el tipo de dato al cual quiere convertirse la expresión entre paréntesis delante de la expresión a ser convertida. Por ejemplo:

```
#include<stdio.h>

 int main(){
     int division,dividendo, divisor;
     double valor;

     dividendo=5;
     divisor=2;

     printf("division entera=%i \n",dividendo/divisor);
     printf("division decimal=%f \n",(float)dividendo/divisor);

     division=dividendo/divisor;
     // conversion explicita o casteo de int a double
     valor=(double) dividendo/divisor;
     return 0;
 }
```

link a ctutor

4.4. El Operador sizeof()

El lenguaje de programación C proporciona un operardor unario llamado *sizeof* que se emplea para calcular el tamaño de cualquier objeto. Las expresiones que contienen a sizeof dan como resultado un valor entero sin signo del tipo *size_t*. Cuando se dice **un objeto** se refiere a que el mismo puede ser: una variable, un arreglo, una estructura o un tipo de dato.

4.5. Puntero Comodín: void*

En C, existe una clase de puntero que se denomina *puntero comodín* o *genérico*, es decir, a este tipo de puntero se le puede asignar la dirección de cualquier puntero de cualquier tipo de dato del lenguaje, ya que no está asociado a

ningun tipo de dato en especial. En otras palabras, un puntero genérico puede apuntar a cualquier cosa. La restricción sobre el puntero comodín es que no puede ser desreferenciado (ver man 3 malloc). Además no pueden aplicarse las reglas de aritmética de punteros a un puntero genérico, y cualquier puntero genérico puede ser convertido a cualquier tipo de dato sin una conversión explicita.

Por ejemplo:

```
1
2  int main(){
3          int a = 10;
4          char b = 'x';
5
6          void *p = &a;   // El puntero a void guarda la dirección del entero  'a'
7          p = &b;                  //  El puntero a void guarda la dirección del  caracter 'b'
8          return 0;
9  }
```

link a ctutor Aspectos interesantes de los punteros genéricos:

1. Las funciones malloc() y calloc() devuelven *void **, ver man 3 malloc.
2. Son utilizados para crear funciones genéricas int (*comparator)(const void*,const void*).

Cosas a tener en cuenta:

1. Los punteros genéricos *no pueden ser desreferenciados directamente*:

```
1  #include<stdio.h>
2  int main()
3  {
4      int a = 10;
5      void *ptr = &a;
6      printf("%d", *ptr);
7      return 0;
8  }
```

para desreferenciar siempre hay que castearlos:

```
1                  #include<stdio.h>
2  int main()
3  {
4      int a = 10;
5      void *ptr = &a;
6      printf("%d", *(int *)ptr);
7      return 0;
8  }
```

2. El estándar de C no permite aritmética de punteros en C. GNU si lo permite pero considera su longitud 1.

4.5.1. Ejemplos

```
1  #include<stdio.h>
2  //To use realloc in our program
3  #include<stdlib.h>
4
5
6
7  int main ()
8  {
9          char* elem[]={"a","b","c","d","e","f","g","h","i","j"};
10         void** elem_generico=NULL;
11
12
13         elem_generico=malloc(sizeof(void*)*10);
14
15         for(int i =0; i<10; i++)
16         elem_generico[i]=(void *) elem[i];
17
18         free(elem_generico);
19         return 0;
20
21
22
23 }
```

link a ctutor
Más ejemplos sobre la utilización de punteros genéricos:

```
void *vp;

int a = 100, *ip;
float f = 12.2, *fp;
char ch = 'a';
```

vp es un puntero genérico:

```
vp = &a; // ok
vp = ip; // ok
vp = fp; // ok

ip = &f; // mal
fp = ip; // mal
```

4.5.2. Un Ejemplo

A continuación se muestra un ejemplo del uso de los punteros genéricos:

```
#include<stdio.h>

int main(){
        Struct student{
                char *name;
                int roll;
        };


        Struct student *stu;
        Stu=(struct student *)malloc(sizeof(struct student));
        return 0;
}
```

Referencias