Memoria dinámica

PUNTEROS

Contenido

División de la memoria

Punteros

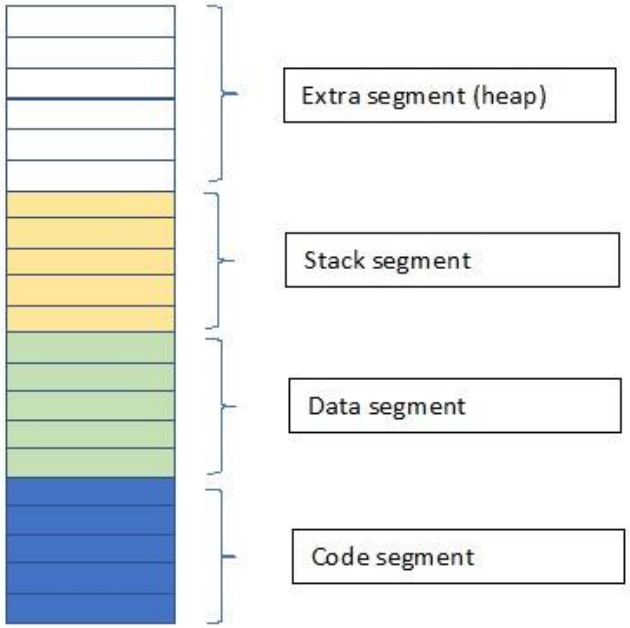
Punteros y vectores

Pedidos de memoria y liberación

Ejemplos

División de la memoria

A la memoria del ordenador la podemos considerar como una tira de celdas dividida en 4 segmentos lógicos que pueden variar en tamaño.



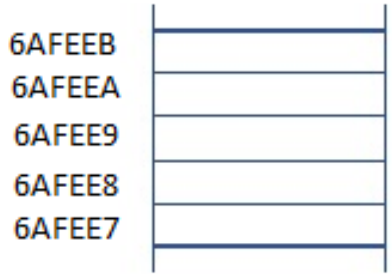
División de la memoria

- El *Code Segment,* o segmento de código, es donde se localiza el código resultante de compilar nuestra aplicación.
- El *Data Segment*, o segmento de datos, almacena el contenido de las variables definidas como externas (variables globales) y las estáticas .
- El *Stack Segment*, o pila, almacena el contenido de las variables locales en la invocación de cada función, incluyendo las de la función main.
- El *Extra Segment,* o heap, es la zona de la memoria dinámica.

División de la memoria

Cada celda corresponde a un *byte*, la unidad mínima de memoria. Recordemos que un byte es la conformación de 8 bits. En cada bit podemos almacenar dos intensidades distintas de corriente, en donde la intensidad más alta se interpreta como el valor 1, y la más baja como el valor 0.

Por este motivo, en cada byte se pueden almacenar hasta $2^8 = 256$ valores diferentes. Cada celda se identifica con un número que llamamos *dirección*, este número es correlativo. En general se representa en formato hexadecimal por ser múltiplo de 8.



División de la memoria

Cada variable ocupa una o más de estas celdas, dependiendo del tipo de variable y la plataforma de la máquina. Por ejemplo, si una variable de tipo *int* ocupa 4 bytes, se reservan cuatro celdas contiguas, siendo la dirección de la primera celda la dirección correspondiente a esa variable. Una variable de tipo *bool* ocupa un byte entero, aunque con un bit alcanza para guardar todos sus valores (0 o 1). Los otros 7 bits los rellena con ceros.

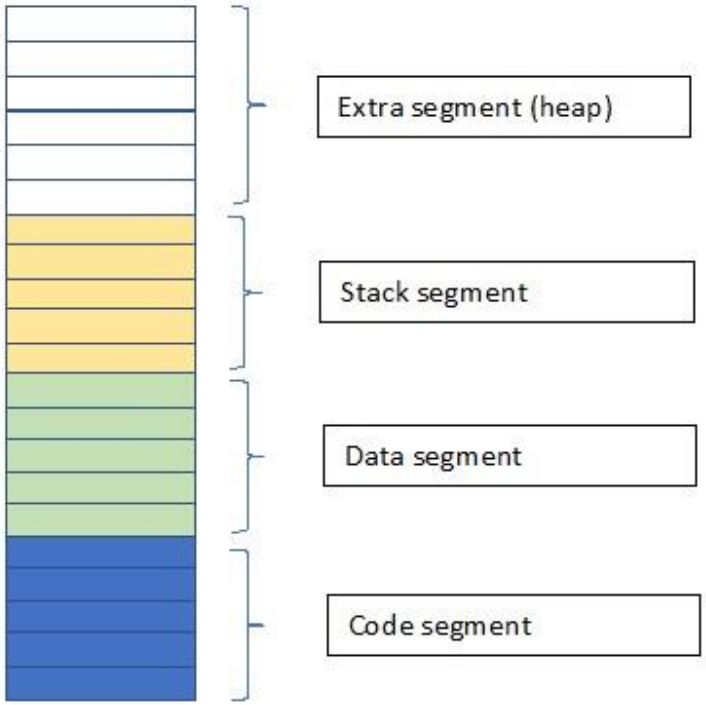
Si queremos saber en qué dirección está determinada variable debemos usar el operador &, pero para imprimir su valor hay que indicar que lo tome como un entero sin signo. Por ejemplo:

```
1.  int a=8;
2.  cout << (unsigned)&a<<endl;
```

División de la memoria

De la gestión de los tres primeros bloques de memoria: el segmento de código, el de datos y el de pila, se encarga el *compilador,* quien reserva y liberará los bloques de memoria de manera automática.

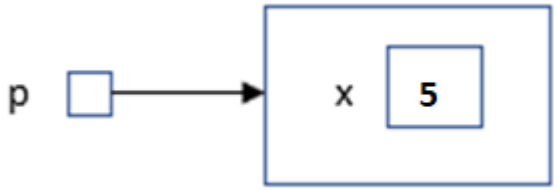
La gestión del bloque identificado como heap será responsabilidad del *desarrollador*. Es decir, los bloques de memoria que se soliciten al heap deben hacerse con una determinada instrucción en el programa, y cuando la variable no se necesite más debemos liberar la memoria ocupada con otra instrucción, en caso de no hacerlo esa porción de memoria queda inutilizada hasta que apaguemos la máquina.



Punteros

Un *puntero* es un tipo especial de variable que almacena direcciones de memoria. Es decir, en lugar de guardar datos, almacena direcciones de memoria de la propia máquina.

Cuando una variable de tipo puntero guarda una dirección de memoria decimos que *apunta* a esa dirección.



Punteros

Un puntero tiene que conocer, salvo alguna excepción que veremos pronto, de qué tipo será la variable que esté en la dirección de memoria que almacenará su valor. Dicho de otra forma, qué tipo de dato está guardado o se guardará en la dirección adonde apunta.

Por ejemplo, si un puntero guarda la dirección de memoria *AB*0012, debe saber qué tipo de dato se almacenará en dicha dirección, si será un *int*, un *float* o cualquier otro.

Punteros

Esto se debe a dos razones: en primer lugar debe saber qué cantidad de celdas ocupará el dato que está en dicha dirección.

En segundo lugar debe saber cómo interpretar esos datos. Tenemos que recordar que cada celda está formada por bits pero esos bits pueden ser interpretados de distinta manera, por ejemplo, como enteros, como flotantes, si representan un carácter en código ASCII, etc.

Punteros

Para llevar a cabo su función de almacenamiento debe ocupar el tamaño que ocupa una dirección, es decir una palabra de la arquitectura de la máquina.

En el caso de procesadores de 16 bits, son 2 bytes, para los de 32, 4 bytes y, para los de 64, 8 bytes.

Punteros

Al igual que cualquier otra variable, se declaran indicando su tipo y un identificador de la variable, con la salvedad que luego de indicar el tipo se debe colocar el signo asterisco “*”.

Ejemplos:

```
int * pi;

char * pc;

float * pf;
```

El símbolo asterisco puede ir inmediatamente después del tipo:

```
int* pi;
```

O inmediatamente antes del identificador de la variable:

```
int *pi;
```

Punteros

Hay que tener ciertos cuidados, por ejemplo:

```
int* pi, pi2, pi3;
```

No declara tres variables de tipo punteros a enteros, sino solo una, la primera.

Tanto *pi2* como *pi3* son variables de tipo enteras.

Punteros

Las direcciones de memoria no podemos asignarlas de forma explícita.

```
int * p = 11206656;
```

NO!

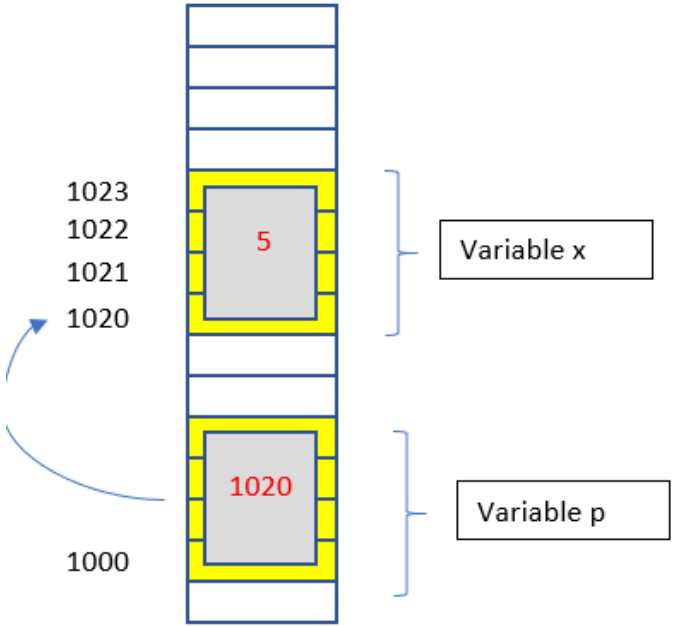
Podemos asignar la dirección de otra variable utilizando el operador “&”. Ejemplo:

```
1.  int main()
2.  {
3.      int * p;
4.      int x = 5;
5.      p = &x;
6.  }
```

Punteros

Supongamos que la variable *p* está ubicada en la dirección 1000 y la variable *x* en la 1020. Entonces, la variable *p* contendrá como dato el valor 1020. Si un entero ocupa 4 bytes, la memoria quedará:

```
1.   int main()
2.   {
3.        int * p;
4.        int x = 5;
5.        p = &x;
6.   }
```



Punteros

Siguiendo con este ejemplo, si quisiéramos modificar el valor de *x* a 8 podríamos hacerlo indirectamente de la siguiente forma:

```
*p = 8;
```

La sentencia anterior equivale a:

```
x = 8;
```

El operador * tiene distinta funcionalidad según el caso, decimos que está *sobrecargado*.

En un caso sirve para *declarar una variable* de tipo puntero. Es decir, su significado es decir que la variable que sigue es un puntero.

En cambio, cuando la variable ya está declarada se llama *operador de desreferenciación*. En este caso el significado es el siguiente “el contenido de la dirección a la que apuntas” o “dirígete a la dirección a la que apuntas”.

Punteros y vectores

Cuando declaramos un vector el compilador reserva la memoria solicitada y guarda solo la dirección del primer elemento. Si declaramos un vector de 10 enteros:

```
int vec[10];
```

De estos 10 lugares solo guarda la dirección del primero en la variable que llamamos *vec*. Luego, es responsabilidad del programador no excederse de los 10 lugares pedidos.

Supongamos que en dicho vector guardamos los números 100, 200, 300, etc. La siguiente línea:

```
cout << vec[5] << endl;
```

imprime 600, la sexta posición del vector.

En cambio, la siguiente instrucción:

```
cout << vec << endl;
```

debería imprimir la dirección en donde está el número 100.

Punteros y vectores

Por lo tanto, como en la variable *vec* estamos guardando una dirección.
La podemos pensar como un puntero constante, ya que a una variable de tipo puntero podemos cambiarle su contenido, lo cual la haría apuntar a otro lugar. En cambio, a una variable de tipo vector, no.
Con esto en mente, la siguiente instrucción es válida:

```
int * p = vec;
```

O también como parámetro de una función:

```
void cargar (int * p, int n);
```

Entonces, si manejamos un vector a través de punteros, ¿cómo accedemos a sus posiciones? Debemos mantener la lógica que vimos de punteros con el operador de desreferenciación:

```
*p // accede a la primera posición
*(p+1) // accede a la segunda posición
```

Punteros y vectores

Sin embargo, tenemos otra forma más natural de acceder que es utilizando los corchetes con índices como con cualquier variable de tipo vector:

```
p[0] // accede a la primera posición
p[1] // accede a la segunda posición
```

Por lo visto, los vectores y los punteros no presentan diferencias, salvo que los primeros son constantes.

Cuidado: los valores de los vectores pueden modificarse pero siempre en la misma porción de memoria.

Pedido de memoria y liberación

Lo que vimos hasta el momento sobre las variables de tipo puntero no tendría sentido si no tuviéramos la necesidad de utilizar memoria dinámica. Es decir, memoria que solicita el programador en el momento de ejecución. Esta zona de la memoria es la que llamamos *heap*.

¿Cómo solicitar memoria al heap? Mediante la instrucción *new tipo*.
Ejemplo:

```
new int;
```

Crea una nueva variable de tipo int en la zona del heap. Sin embargo, esta variable anónima la perderíamos si no tuviéramos la dirección en donde fue creada.

Pedido de memoria y liberación

El operador *new* hace algo más que crear la variable solicitada: devuelve su dirección, la cual debemos guardarla en una variable adecuada.
Una variable adecuada para guardar direcciones es un puntero, por lo tanto, la instrucción correcta es:

```
int * p = new int;
```

De esta forma *p* guarda la dirección de esa variable anónima y puede acceder mediante el operador * para guardar / recuperar valores.

Cuando trabajemos con memoria dinámica debemos tener cuidado porque la memoria solicitada hay que liberarla.
La instrucción para liberar la memoria es con el operador *delete:*

```
delete p;
```

Pedido de memoria y liberación

La potencia de los punteros se aprecia en las estructuras dinámicas. Si quisiéramos crear un vector de enteros en forma dinámica podemos solicitar todo un bloque al *heap*:

```
int * p = new int [100];
```

En la instrucción anterior estamos creando un bloque contiguo de 100 enteros del cual guardamos la dirección del primer entero en la variable *p*. Ya vimos cómo podemos acceder a cada uno de esos enteros, trabajando con la variable *p* como si fuera un vector común y corriente.

Al momento de liberar debemos indicarle al compilador que debe liberar todo un bloque colocando corchetes vacíos luego del operador *delete*:

```
delete [] p;
```

Ejemplo 1

```
1.  int main()
2.  {
3.      int x;
4.      int *px;
5.      cout<<"ingrese un valor para x:";
6.      cin>>x;
7.      px=&x;
8.      cout<<"el numero ingresado fue: "<<*px;
9.  }
```

Ejemplo 1

```
1.  int main()
2.  {
3.      int x;
4.      int *px;
5.      cout<<"ingrese un valor para x:";
6.      cin>>x;
7.      px=&x;
8.      cout<<"el numero ingresado fue: "<<*px;
9.  }
```

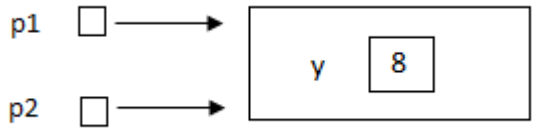
```
ingrese un valor para x:3
el numero ingresado fue: 3
```

Ejemplo 2

```
1.  int main()
2.  {
3.      int x=5, y=8;
4.      int *p1;
5.      int *p2;
6.      p1=x;
7.      p1=&x;
8.      p2=&y;
9.      cout<<p1<<endl;
10.     p1=p2;
11.     *p1=*p2;
12.     return 0;
13. }
```

Ejemplo 2

```
1.  int main()
2.  {
3.      int x=5, y=8;
4.      int *p1;
5.      int *p2;
6.      p1=x;                  //Linea 6: ERROR!
7.      p1=&x;
8.      p2=&y;
9.      cout<<p1<<endl;        //Linea 10:
10.     p1=p2;
11.     *p1=*p2;
12.     return 0;
13. }
```



Ejemplo 3

```
1.  typedef int* Pint;
2.  typedef char* Pchar;
3.  /*Datos:
4.  ´@´ es 64
5.  ´A´ es 65
6.  …*/
```

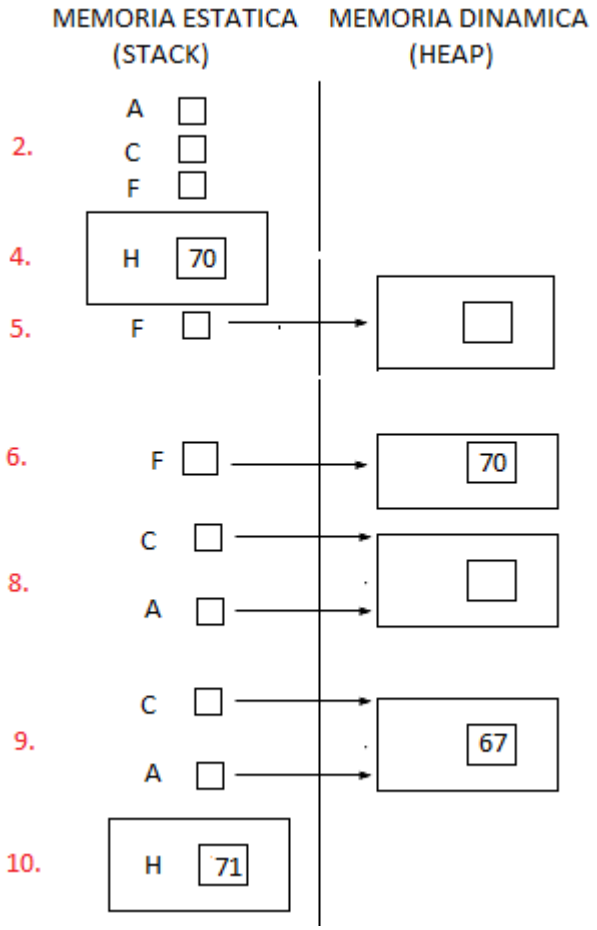
Ejemplo 3

```
1.  main() {
2.      Pint A, C, F;
3.      int H;
4.      H=70;
5.      F=new int;
6.      (*F)=H;
7.      C=new int;
8.      A=C;
9.      (*A)=67;
10.     H++;
11.     cout<<(*C)<<(*A)<<(*F)<<endl;
12. }
```

Ejemplo 3

```
1.  main() {
2.      Pint A, C, F;
3.      int H;
4.      H=70;
5.      F=new int;
6.      (*F)=H;
7.      C=new int;
8.      A=C;
9.      (*A)=67;
10.     H++;
11.     cout<<(*C)<<(*A)<<(*F)<<endl;
12. }
```

```
676770
```

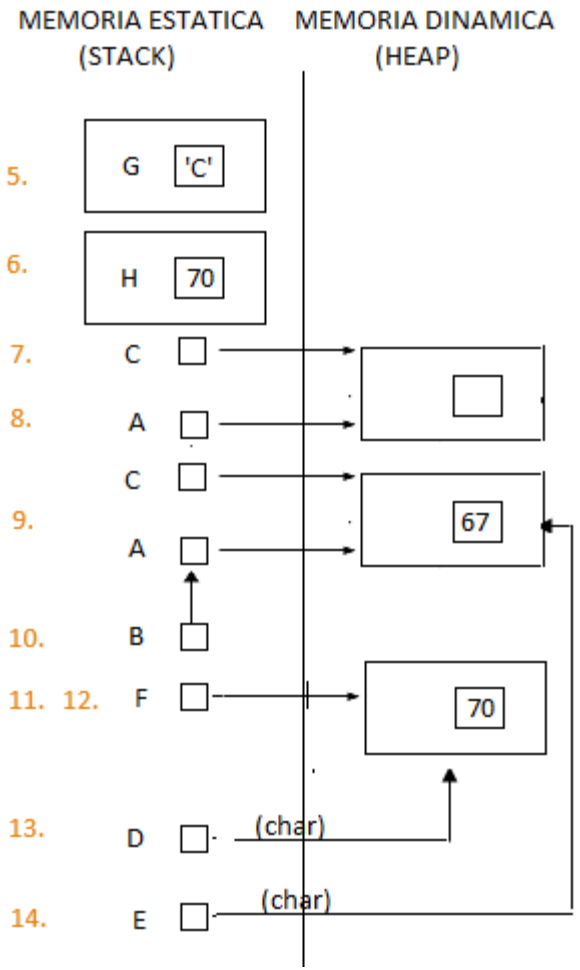


Ejemplo 4

```
1.  main() {
2.      Pint A,C,F;
3.      Pint*B;
4.      Pchar D,E;
5.      char G='C';
6.      int H=70;
7.      C=new int;
8.      A=C;
9.      (*A)=67;
10.     B=&A;
11.     F=new int;
12.     (*F)=H;
13.     D=(Pchar)F;
14.     E=(Pchar)(*B);
15.     (**B)=(*A)-63;
16.     if((*E) != G)
17.         cout<<(*A)<<(*D)<<(*C)<<endl;}
```

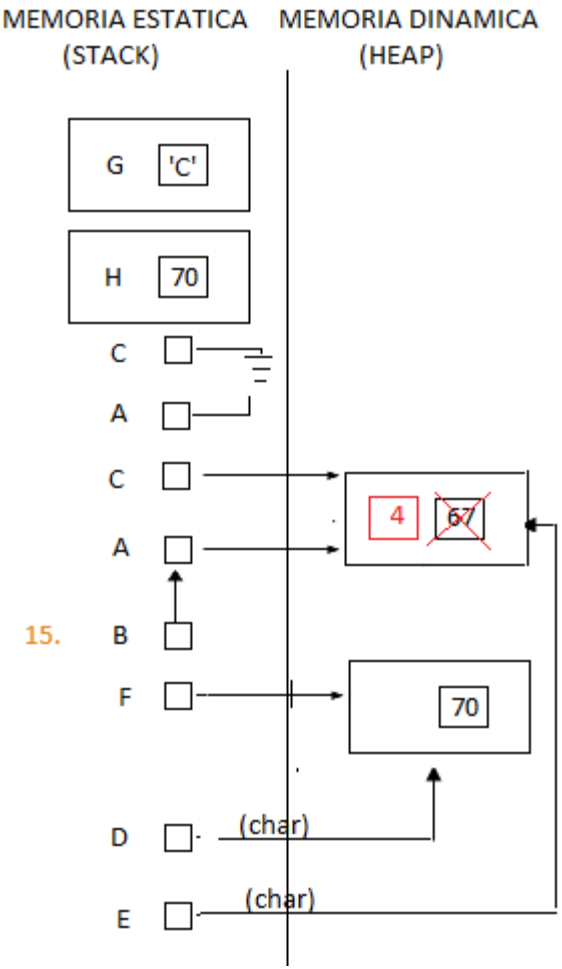
Ejemplo 4

```
1.   main() {
2.       Pint A,C,F;
3.       Pint*B;
4.       Pchar D,E;
5.       char G='C';
6.       int H=70;
7.       C=new int;
8.       A=C;
9.       (*A)=67;
10.      B=&A;
11.      F=new int;
12.      (*F)=H;
13.      D=(Pchar)F;
14.      E=(Pchar)(*B);
15.      (**B)=(*A)-63;
16.      if((*E) != G)
17.          cout<<(*A)<<(*D)<<(*C)<<endl;}
```



Ejemplo 4

```
1.   main() {
2.       Pint A,C,F;
3.       Pint*B;
4.       Pchar D,E;
5.       char G='C';
6.       int H=70;
7.       C=new int;
8.       A=C;
9.       (*A)=67;
10.      B=&A;
11.      F=new int;
12.      (*F)=H;
13.      D=(Pchar)F;
14.      E=(Pchar)(*B);
15.      (**B)=(*A)-63;
16.      if((*E) != G)
17.          cout<<(*A)<<(*D)<<(*C)<<endl;}
```



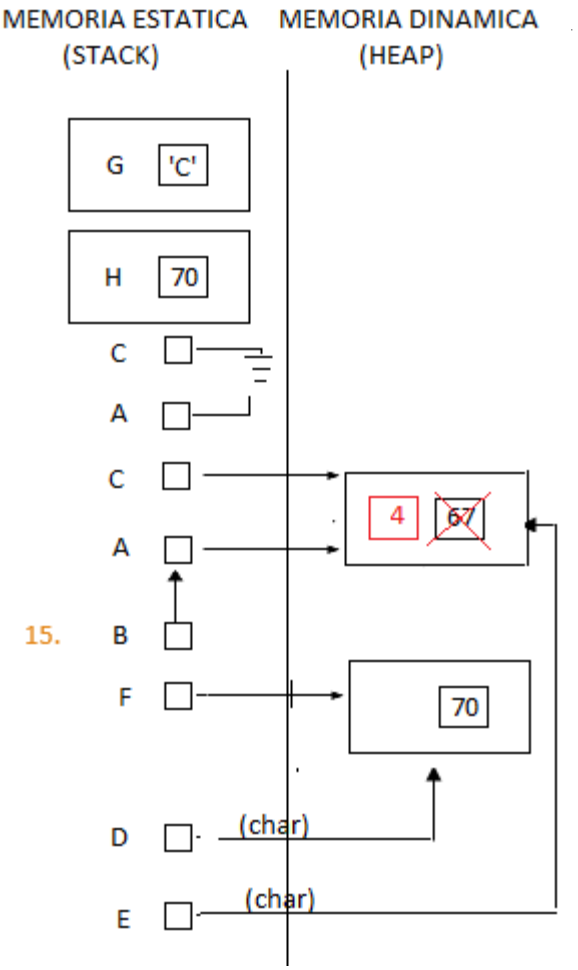
4F4

Ejemplo 4

Que pasa si hacemos:

```
16.  (*A)=(*A)-(*C)+66;

17.  cout<<(*E)<<(*F)<<G<<endl;
```

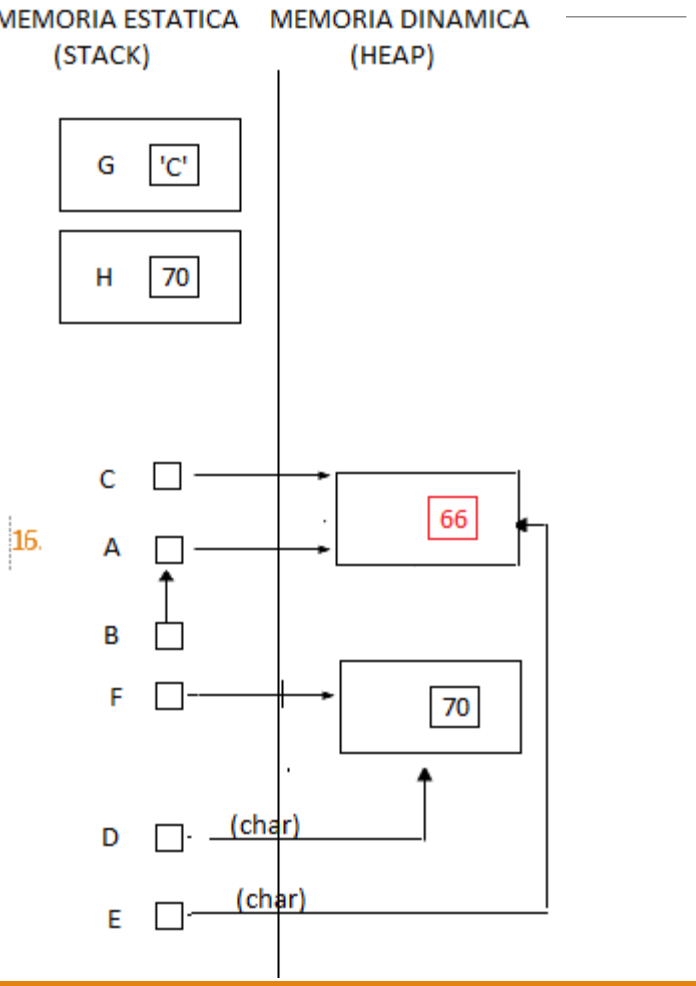


Ejemplo 4

Que pasa si agregamos:

```
16.  (*A)=(*A)-(*C)+66;
17.  cout<<(*E)<<(*F)<<G<<endl;
```

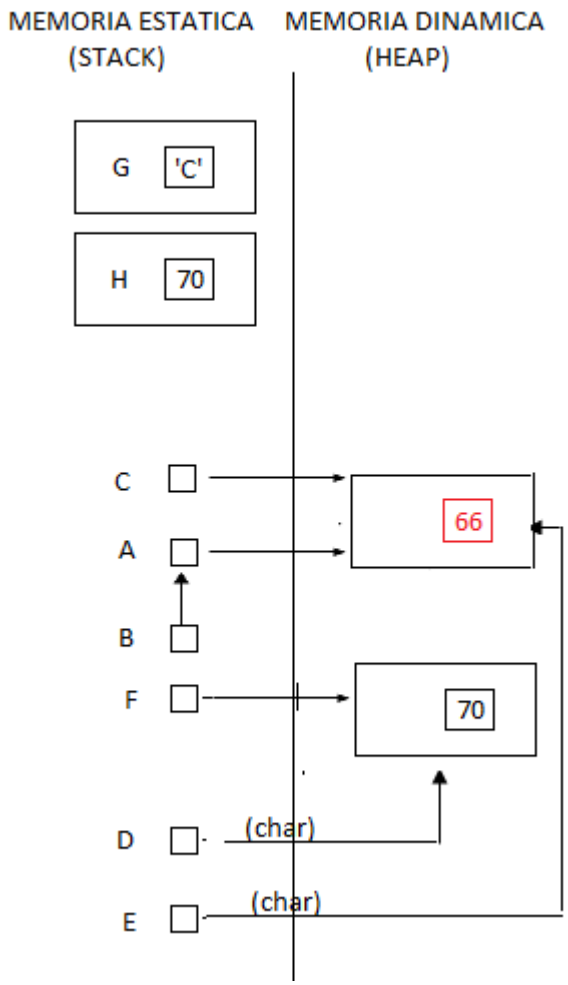
B70C



Ejemplo 4

Y si ahora agregamos:

```
18.  while ((*C)>0){
19.     (*E)='E';
20.     (*A)=(*F)-(*C);
21.     cout<<(**B)<<endl;
22.     (*C)--;}
```

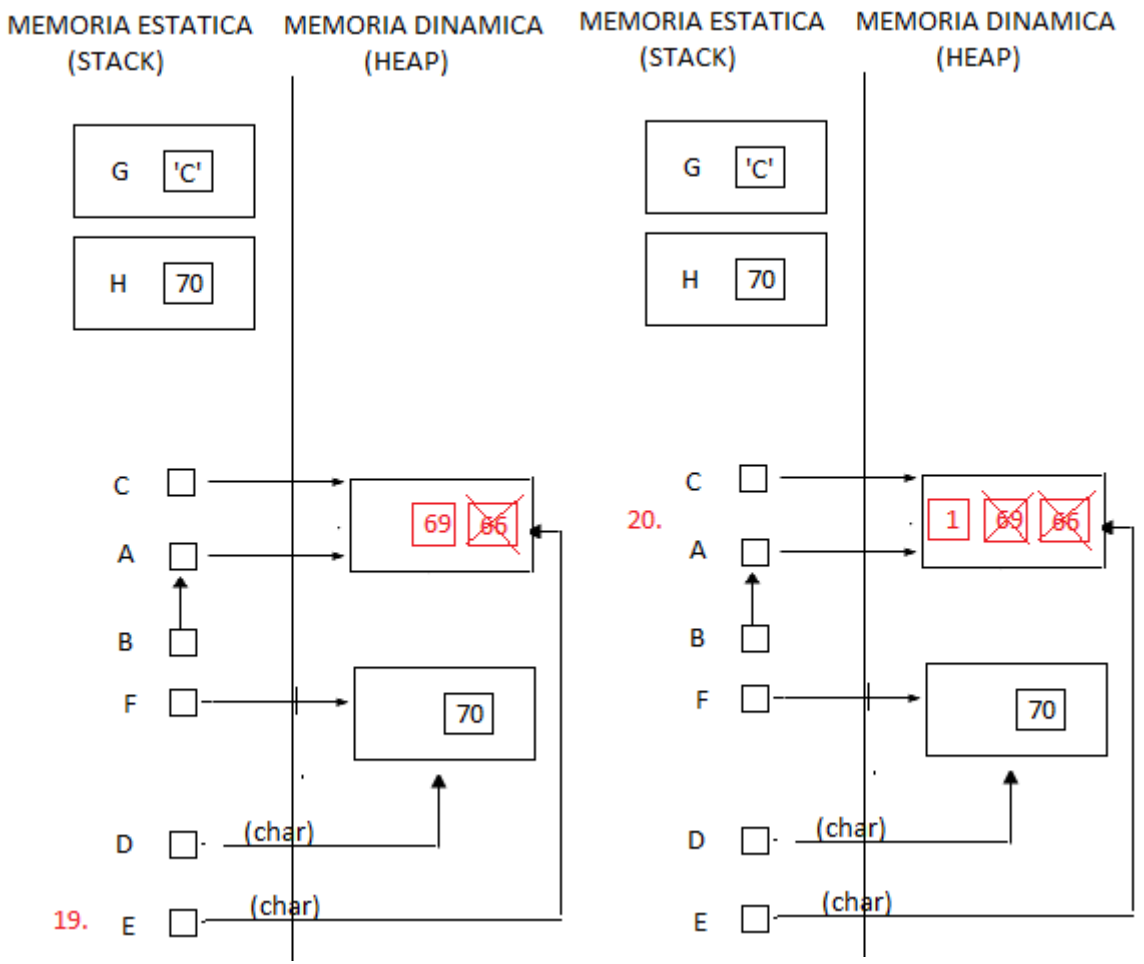


Ejemplo 4

Y si ahora agregamos:

```
18.  while ((*C)>0){
19.     (*E)='E';
20.     (*A)=(*F)-(*C);
21.     cout<<(**B)<<endl;
22.     (*C)--;}
```

1



Fin