**Nombre:** Ismael Iván López Murillo.

Instrucciones:

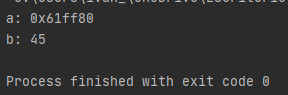
Programar cada uno de los segmentos de código mostrados en cada ejercicio; para cada uno de ellos hacer una ejecución y realizar una captura de pantalla del resultado además de explicar el resultado obtenido.

1. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a;

int b = 45;

cout <<"a: " <<a << endl <<"b: " <<b <<endl;



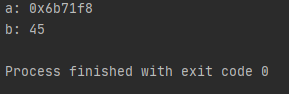
Imprime la dirección a la cual apunta “a”, la cual es una dirección al azar, al no estar inicializada aún, puede ser memoria el programa, del sistema operativo o cualquier otro, no lo sabemos, enseguida imprime el valor de b (45). El status 0 nos dice que no se violó ningún acceso a memoria, todo fue válido, aunque si quisiéramos acceder a \*a ya tendríamos problemas.

2. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

int b = 45;

cout <<"a: " <<a << endl <<"b: " <<b <<endl;



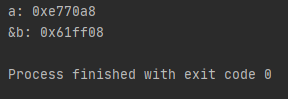
Se imprime la dirección a la cual apunta “a” y el valor de “b”, sin embargo, aunque a y b tengan en mismo valor, no son la misma variable, “a” no apunta a la dirección de memoria en la cual “b” está almacenada, sino a una propia, al haberse declarado en tiempo de ejecución, dicha variable de hecho está en el heap y no en el stack (como “b”). El status 0 nos dice que no se violó ningún acceso a memoria, todo fue válido.

3. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

int b = 45;

cout <<"a: " <<a << endl <<"&b: " <<&b <<endl;



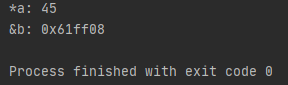
Como se mencionó en el ejercicio anterior, “a” y “b” tienen distinto espacio en la memoria reservada, aquí imprimimos la dirección en la cual están almacenadas nuestras variables tipo int con valor de 45 “a” y “b”, no es necesario usar & en “a” ya que “a” es un puntero que precisamente apunta a un espacio de memoria, podemos ver que tienen distinta dirección. El status 0 nos dice que no se violó ningún acceso a memoria, todo fue válido.

4. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

int b = 45;

cout <<"\*a: " <<\*a << endl <<"&b: " <<&b <<endl;



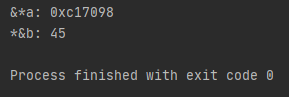
Caso contrario a su naturaleza, ahora imprimimos el valor de la variable a la cual apuntamos con el puntero “a” y la dirección de memoria en la cual “b” está almacenada. Para ello hacemos uso de \* y & respectivamente. No son la misma variable ni tienen el mismo espacio de memoria. No se viola ningún acceso a memoria.

5. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

int b = 45;

cout <<"&\*a: " <<&\*a << endl <<"\*&b: " <<\*&b <<endl;



En este caso imprimimos la dirección de memoria a la cual apunta el valor de contenido en el puntero “a”, es decir, es lo mismo que imprimir solamente “a”, es como si se cancelaran dichos operadores, y en el caso de “b”, imprimimos el valor de lo que hay en la dirección de “b”, es decir, “b”. Estos operadores no sirven si los usamos al revés. No hay violación de acceso a memoria.

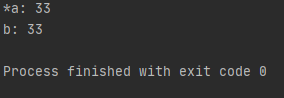
6. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

int b = 33;

a = &b;

cout <<"\*a: " <<\*a << endl <<"b: " <<b <<endl;



En este caso imprimimos tanto el valor de la variable a la que apunta “a” y el valor de “b”, para este código, 33, de hecho, el mismo 33, “a” y “b” están relacionadas, ya que “a” apunta a la dirección de memoria en “b”, por ello al imprimir el valor de lo que apunta y el valor mismo, obtenemos la misma respuesta. En este caso dejamos una referencia colgante al new int que creamos. Aún así, no hay violación de acceso a memoria.

7. ¿Qué resulta de ejecutar el siguiente segmento de código?

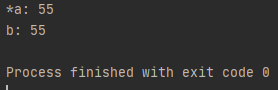
int\* a = new int(45);

int b = 33;

a = &b;

b = 55;

cout <<"\*a: " <<\*a << endl <<"b: " <<b <<endl;



En este caso, igual que en el ejercicio anterior, “a” termina apuntado a la dirección de “b”, por ello, al cambiar el valor de “b”, también cambia el valor del contenido de la dirección de “a”, el cual imprimimos, tanto mediante la variable entera como por valor en puntero. Podemos decir que “a” es una referencia a “b”. Dejamos una referencia colgante, pero aún no hay violación de memoria.

8. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

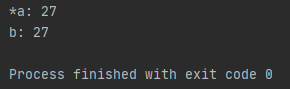
int b = 33;

a = &b;

b = 55;

\*a = 27;

cout <<"\*a: " <<\*a << endl <<"b: " <<b <<endl;



Al igual que en el ejercicio anterior, “a” es una referencia a “b”, por lo tanto, si modificamos el valor de lo que hay en la dirección a la cual apunta “a”, también cambia “b” (al menos mientras esté referenciada). Por ello, al imprimir el valor almacenado en la dirección a la cual apunta “a” y el valor de “b” tenemos la misma salida, porque están relacionadas a nivel de memoria. Lo que cambia en “b” y en “\*a” va a estar ligado. No hay violación de acceso de memoria.

9. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

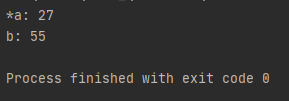
int b = 33;

a = &b;

b = 55;

a = new int(27);

cout <<"\*a: " <<\*a << endl <<"b: " <<b <<endl;



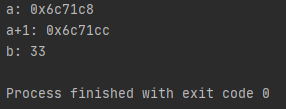
En este caso, aunque “a” era referencia de “b”, fue reasignado con el operador new, por lo tanto, ahora “a” no tiene nada que ver con “b” (aún). Se imprime el valor contenido en la dirección apuntada de “a”, el cual es 27, el valor creado con el operador new, y el valor de b, el cual fue reasignado a 55. No hay violación de acceso de memoria.

10. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

int b = 33;

cout <<"a: " <<a << endl <<"a+1: " <<a+1 <<endl <<"b: " <<b <<endl;



En este caso, “a” y “b” no tienen relación, al imprimir a, obtenemos la dirección en la cual la variable de valor 27 creada con new está almacenada, y al imprimir a+1 accedemos al siguiente bloque de memoria (adyacente), tomando en cuenta que un int ocupa 4 espacios, 0x6c71c8 + 4 = 0x6c71cc (Se toma en cuenta el tipo de dato del puntero para hacer el “recorrido” con el operador de +). Se imprime el valor de b, el cual es una variable de toda la vida. No hay violación de acceso de memoria porque no accedemos a \*(a+1), el espacio que no podemos asegurar que sea de nuestro control.

10.1 ¿Qué resultará de evaluar a+2?

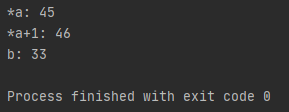
0x6c71c8 + 2(4) = 0x6c71d0.

11. ¿Qué resulta de ejecutar el siguiente segmento de código?

int\* a = new int(45);

int b = 33;

cout <<"\*a: " <<\*a << endl <<"\*a+1: " <<\*a+1 <<endl <<"b: " <<b <<endl;



Se imprime el valor almacenado en la dirección a la cual apunta “a”, luego ese valor + 1, que es como imprimir cualquier valor +1 (ej: (‘a’) + 1 = ‘b’ <casteado a char>), esto no modifica el valor de \*a, y finalmente el valor de b. No hay violación en acceso de memoria.

12. ¿Qué resulta de ejecutar el siguiente segmento de código?

void swap(int &a, int &b)

{

int temporal;

temporal = a;

a = b;

b = temporal;

}

int main()

{

int c = 5;

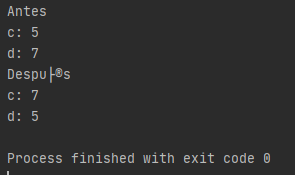
int d = 7;

cout <<"Antes" <<endl <<"c: " <<c <<endl <<"d: " <<d <<endl; swap(c, d);

cout <<"Después" <<endl <<"c: " <<c <<endl <<"d: " <<d <<endl;

return 0;

}



Se imprime el valor de los variables enteras creadas, c = 5, d = 7. Luego se mandan a una función como tal (no su dirección, sino la variable). Sin embargo, dentro del prototipo de la función podemos notar que son recibidas como referencia constante, ojo, no se recibe su dirección como tal, para ello usaríamos un puntero, sino una referencia a ellas (mediante &) y este operador no indica que estamos obteniendo su memoria, ya que no es una operación como tal, sino que podemos verlo como un tipo de dato “referencia entera”, es por ello que al modificar dentro de la función el valor de estas variables, esto hace un cambio en main, es por ello que al imprimir de vuelta los valores vemos que están intercambiados. No hay violación de acceso a memoria.

13. ¿Qué resulta de ejecutar el siguiente segmento de código?

void incrementa(int \*a)

{

(\*a)++;

}

int main()

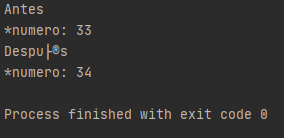
{

int\* numero = new int(33);

cout <<"Antes" <<endl <<"\*numero: " <<\*numero <<endl; incrementa(numero);

cout <<"Después" <<endl <<"\*numero: " <<\*numero <<endl; return 0;

}



En este caso primero imprimimos el valor que tiene el espacio de memoria al cual apunta nuestro puntero “numero” (33), este después es mandado como tal a una función que recibe un puntero, dicha función incrementa el valor almacenado en el espacio al cual apunta el puntero en 1, es por ello que al imprimir el valor después de la llamada tenemos de salida 34, ya que se cambió el valor de lo que hay dentro de la dirección a la cual apunta nuestro puntero “numero”. No hay violación de acceso a memoria.