Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

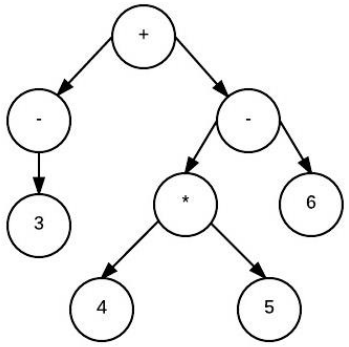
Explicación del código

Pregunta #1

```
void pregunta1() { // Aritmética básica
    int x;
    x = - 3 + 4 * 5 - 6; PRINTX;
    x = 3 + 4 % 5 - 6; PRINTX;
    x = - 3 * 4 % - 6 / 5; PRINTX;
    x = ( 7 + 6 ) % 5 / 2; PRINTX;

}
```

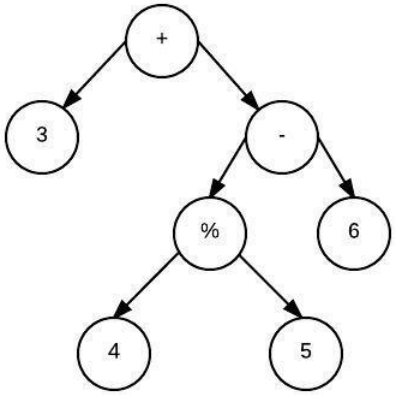
Para la primera asignación se da que x = 11 esto por la precedencia de operadores:



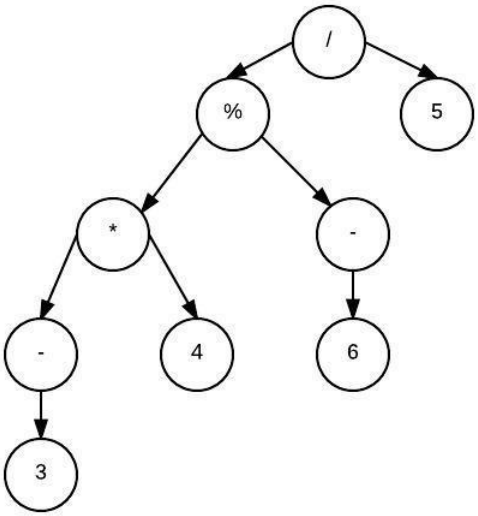
Por lo que el árbol se va evaluando de abajo hacia arriba y esto concluye en 11.

La segunda asignación x = 1 esto se da porque:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

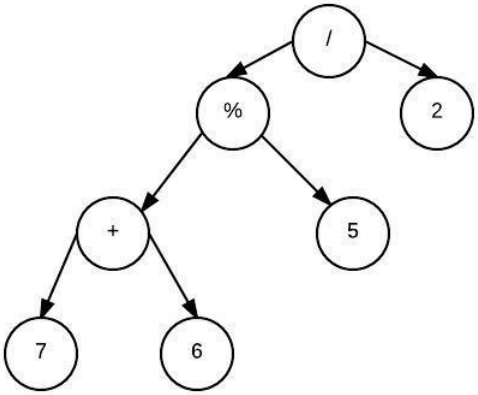


La tercera asignación de x es 0 ya que el módulo con -6 es 0 y la división entre 5 sigue siendo 0.



La última asignación da 1 esto es divido a que la división entre números enteros devuelve un entero si fuera con un flotante devuelve el número exacto de la división:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

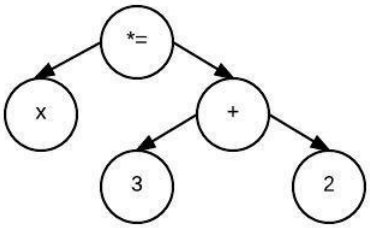


Pregunta #2

```
void pregunta2() { // Asignación
    int x = 2, y, z;
    x *= 3 + 2; PRINTX;
    x *= y = z = 4; PRINTX;
    x = y == z; PRINTX;
    x == (y = z); PRINTX;
}
```

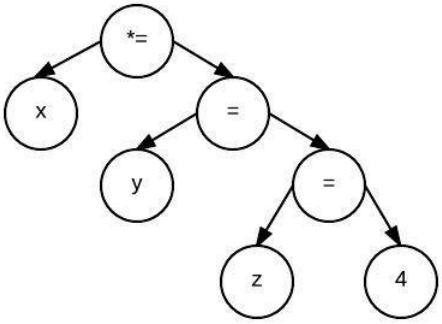
Se inicia con **“x”, “y”** y **“z”** igualados a 2.

La primera asignación de x es una multiplicación entre el valor de x con la suma esto da como resultado 10:



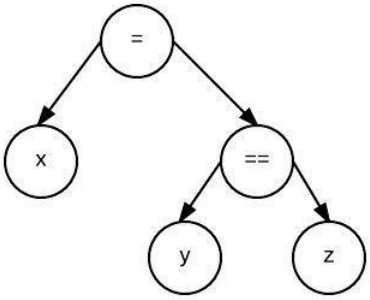
La segunda asignación sigue la misma idea de la primera solo que x = 10 y ya no a 2 como en un inicio, esto es igual a 40 por lo siguiente:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

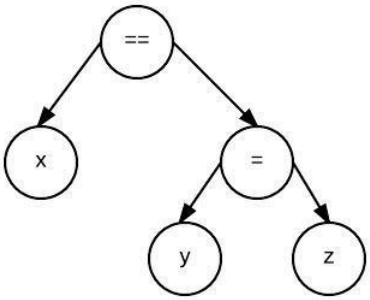


Además **“z”** y **“y”** tienen el valor de 4 asignado.

La tercera a x se le asigna 1 pero este 1 representa True en C y se da porque a x se le está asignando el resultado de una comparación, representado de la siguiente forma:



Con respecto a la última la verdad es que nunca se hace una asignación a x e imprime el resultado anterior ya que es lo que tiene almacenado. Sin embargo, se ilustrará la línea:



Pregunta #3

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

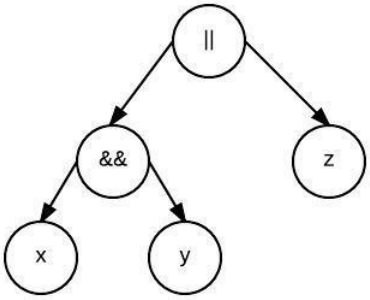
```
void pregunta3() { // Lógica y operadores de incremento
    int x, y, z;

    x = 2; y = 1; z = 0;
    x = x && y || z; PRINT(x);

    x = y = 1;
    z = x ++ - 1; PRINT(x), PRINT(z);
    z += - x ++ + ++ y; PRINT(x); PRINT(z);
    z = x / ++x; PRINT(z);
}
```

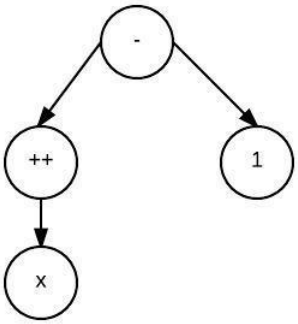
La siguiente pregunta hace referencia a lógica y operaciones de incremento. Se declaran las variables y se les asigna 2, 1 y 0 a "**x", "y"** y **"z"** respectivamente.

Para la primera línea se imprime que x es igual a 1 esto se debe a lo siguiente:



Donde && equivale a un and y || equivale a un or lógico; dado que x && y da True (o sea 1) y 1 || 0 da 1 la asignación para x va a ser 1.

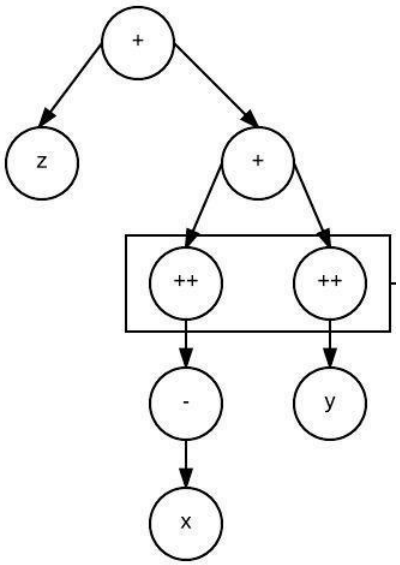
Para la siguiente pregunta se asigna **"x"** y **"y"** igual a 1; e imprime x y z con resultado de 2 y 0 respectivamente esto se da por lo siguiente:



El incremento de x se da hasta después de la asignación de z.

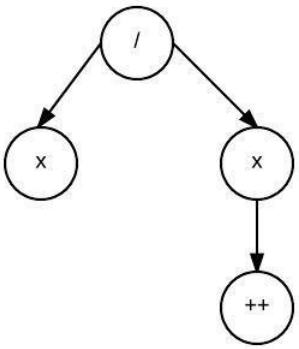
Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

Para la siguiente pregunta los resultados son x = 3 y z = 0 esto sucede por lo siguiente:



En donde se encuentra el rectángulo suceden dos cosas como se dijo anteriormente x se incrementa hasta después de la asignación (x= 3), pero cuando precede a una variable este primero se incrementa haciendo que y = 2 y concluya z = 0.

Para la última pregunta se imprime z el cual equivale a 1 y esto se da por lo siguiente:



De la misma manera ++ x incrementa antes de que se haga la asignación por lo que se hace una división de enteros 3 / 4 lo cual da como resultado 1.

Pregunta #4

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

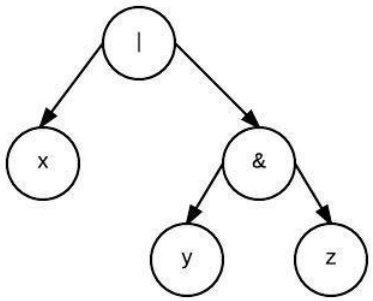
```c
void pregunta4() { // Operadores de bits
    int x, y, z;

    x = 03; y = 02; z = 01;
    PRINT( x | y & z );
    PRINT( x | y & - z );
    PRINT( x ^ y & - z );
    PRINT( x & y && z);

    x = 1; y = -1;
    PRINT( ! x | x );
    PRINT( - x | x );
    PRINT( x ^ x );
    x <<= 3; PRINT(x);
    y <<= 3; PRINT(y);
    y >>= 3; PRINT(y);
}
```

La representación bit a bit de las asignaciones es la siguiente (se expresará en forma binaria) x = 11, y = 10, z = 01.

La primera impresión da 3 esto porque:



Se evalúa primero el &:

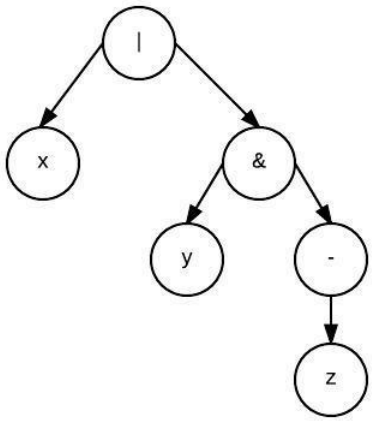
1	0
0	1
0	0

Seguidamente se evalúa el |:

1	1
0	0
1	1

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas
Por lo que se termina imprimiendo 3.

Para la siguiente impresión continúa dando 3 por lo siguiente:



Se evalúa primero el &: (C utiliza complemento-2 para números negativos)

1	0
1	1
1	0

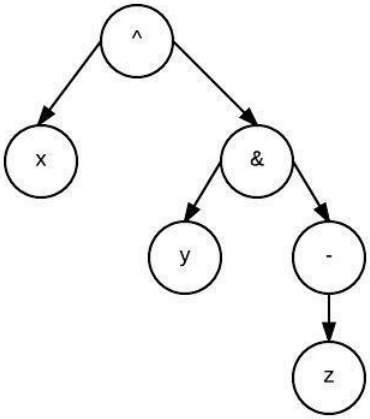
Seguidamente se evalúa el |:

1	1
1	0
1	1

Por lo que se termina imprimiendo 3.

Para la siguiente se imprime uno y esto sucede por lo siguiente:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas



Se evalúa primero el &:

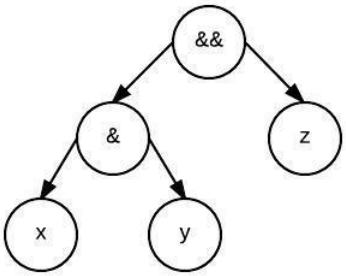
1	0
1	1
1	0

Seguidamente se evalúa el ^:

1	1
1	0
0	1

Por lo que se termina imprimiendo 1.

Después se tiene la impresión de x & y && z la cual imprime 1 al finalizar esto se evalúa de la siguiente manera:



Se evalúa primero el &:

1	1
1	0

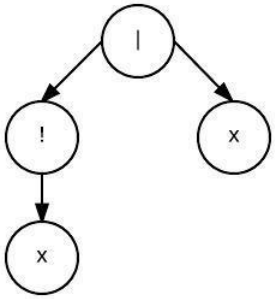
Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

1	0

Seguidamente se evalúa el && con 10 y 01 lo que se termina imprimiendo 1 ya que ambos evalúan a verdadero.

Se asigna x= 1 y y= -1.

Siguen impresiones que revisan los operandos !, -, ^ donde para el primero se tiene que la impresión fue de 1 esto porque:



El ! sirve como negación lógica por lo que si x es 1 (True) su negación va a ser 0 (False) y se evalúa de esta forma con |:

0	0
0	1
0	1

Concluyendo la impresión en 1.

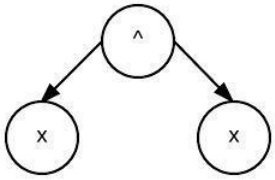
El siguiente es con el – que lo único que hace es hacer el número negativo, se puede ver de la siguiente forma para |:

1	1
0	1
1	1

Por lo que se termina imprimiendo -1 por el resultado anterior.

Para el siguiente que es ^ se obtiene 0 y esto porque:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

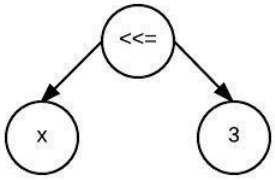


Esto se representa de la siguiente manera para ^:

0	1
0	1
0	0

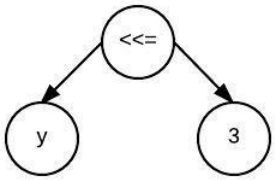
Ya que el ^ es un Xor por lo que termina imprimiendo el 0.

Para las últimas 3 de estas impresiones se tienen los operadores shift, la primera es de la siguiente forma:



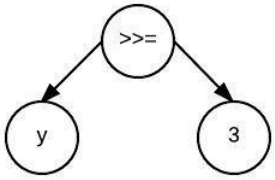
Equivalente a un left shift el cual lo que está haciendo realmente es x = x<<3 y esto significa 1<<3 => 1000 => 8 en decimal por lo que imprime 8.

Para el siguiente se tiene que y <<= 3 y esto significa y = y<<3 por lo que se tiene -1<<3 => -1000 => -8 en decimal por lo que se imprime -8 y se lee de la siguiente forma:



Para el último que es right shift se lee de la siguiente forma:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas



Esto significa y = y >> 3 lo cual implica que -0001 => -1 y eso es lo que imprime al final.

Pregunta #5

```
void pregunta5() { // Operadores relacionales y condicionales
    int x=1, y=1, z=1;

    x += y += z;
    PRINT( x < y ? y : x );

    PRINT( x < y ? x ++ : y ++ );
    PRINT(x); PRINT(y);

    PRINT( z += x < y ? x ++ : y ++ );
    PRINT(y); PRINT(z);

    x=3; y=z=4;
    PRINT( (z >= y >= x) ? 1 : 0);
    PRINT( z >= y && y >= x );
}
```

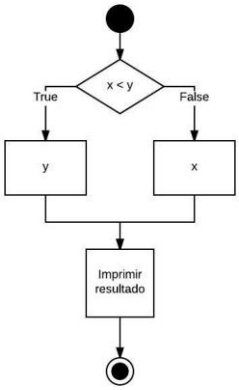
En la siguiente parte se asigna 1 a **“x”, “y”** y **“z”** y a continuación estás se les asigna x = 3, y = 2 y z = 1 (se queda igual).

Los operadores condicionales funcionan de la siguiente forma:

expresión ? expresión si da true : expresión si da false

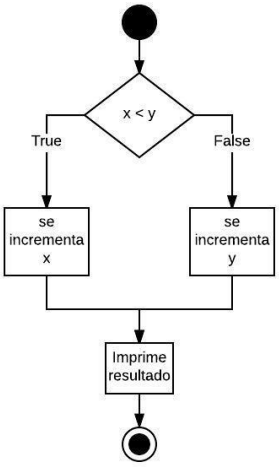
Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

Para la primera impresión se tiene lo siguiente:



Como x no es menor a y entonces se imprime x o sea 3.

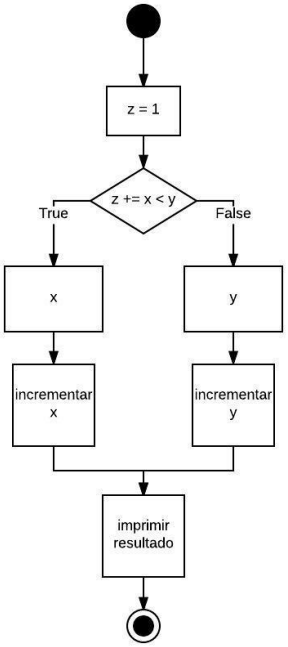
Para el siguiente se sigue el mismo proceso:



En este se evalúa si x<y lo cual es falso por lo que imprime y o sea 2 y después de la impresión incrementa, por lo tanto y = 3; x no incrementa ya que no entro al flujo y queda en 3.

Para el siguiente caso se imprime 4 esto es debido a lo siguiente:

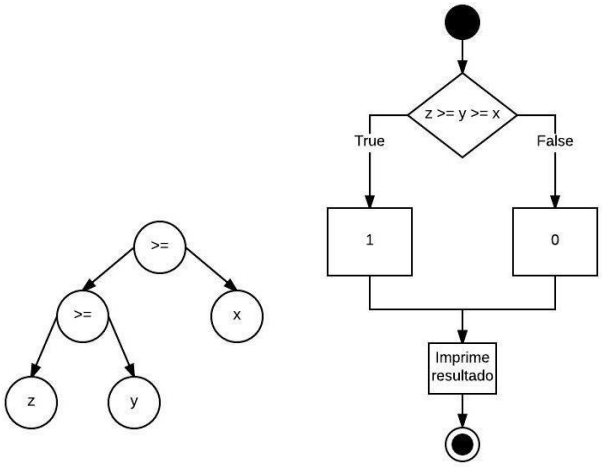
Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas



Aquí suceden 2 cosas, la primera es que se evaluá si x < y esto es falso ya que es igual por lo que se elige a **"y"** y este se le suma a z o sea z = z + 3 => z = 4 e imprime 4, además y = 4 por el incremento.

Los siguientes son relacionales, se asignan x = 3, y = 4 y z = 4.

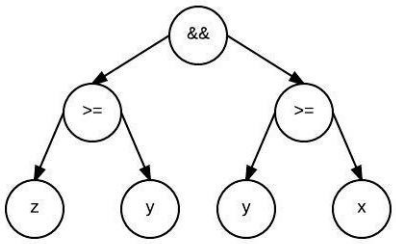
Para la siguiente impresión da resultado 0 esto es por lo siguiente:



Como z es igual a y entonces da True (1), pero 1 es > a x por lo que se va al false.

Para el último problema se presenta lo siguiente:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas



Lo siguiente se lee de la siguiente forma z = y por lo que da True (1) y x<y por lo que da True (1) por lo tanto 1 and 1 devuelve 1 o sea True.

Pregunta #6

```
void pregunta6() { // Precedencia de operadores y evaluación
    int x, y, z;

    x = y = z = 1; //Solo evalua ++x
    ++x || ++y && ++z; PRINT3(x,y,z);

    x = y = z = 1; //Evalua x como true y tiene que revisar b
    ++x && ++y || ++z; PRINT3(x,y,z);

    x = y = z = 1; //Evalua todos
    ++x && ++y && ++z; PRINT3(x,y,z);

    x = y = z = -1; //Evalua x, y no, z verifica que el b sea T/F
    ++x && ++y || ++z; PRINT3(x,y,z);

    x = y = z = -1; //Evalua a y b
    ++x || ++y && ++z; PRINT3(x,y,z);

    x = y = z = -1; //Evalua solo a
    ++x && ++y && ++z; PRINT3(x,y,z);
}
```

Los siguientes problemas lo que hacen es evaluar los operadores lógicos de la forma más optima, o sea en el caso de and evalúa la primera expresión si es True y después la segunda lo demás lo tira como False, en el caso del or se revisa la primera expresión que sea False y si es así, evalúa que la segunda también de lo contrario retorna True.

Se asigna a x = 1, y = 1 y z = 1

Para la primera impresión se da que x = 2, y = 1 y z = 1; esto se debe a que

++x || ++y && ++z al momento de leer x ya es true por lo que no importa lo demás y solo incrementa a x y por eso la impresión termina en 2.

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

Para la segunda impresión se da x = 2, y = 2 y z = 1; esto se debe a lo siguiente:

++x && ++y || ++z lo siguiente evalúa x el cual es True por lo que también evalúa la segunda expresión y para el or como su primera expresión dio True no importa lo que siga.

Seguidamente la tercera expresión tiene lo siguiente:

++x && ++y && ++z En este evalúa a todas las variables por lo que las 3 se incrementan en una unidad x = y = z = 2.

Para los siguientes casos **“x”, “y”** y **“z”** equivalen a -1.

La siguiente expresión tiene **++x && ++y || ++z** por lo que evalúa a x el cual es 0 por el pre-incremento y esto da false, no evalúa a y, pero dado que el or verifica la segunda expresión en caso de que sea True, incrementa z el cual da 0 (False) e imprime que x = 0, y = -1, z = 0.

La siguiente impresión da x = 0, y = 0 y z = -1 esto se debe a que la expresión **++x || ++y && ++z** incrementa a x y lo convierte en False (0) como es un or tiene que asegurar que la segunda expresión no sea True para devolver 1, evalúa y incrementado y este da False (0) debido a que el and recibe False (0) no evalúa a z.

Para la última expresión sucede lo siguiente **++x && ++y && ++z** evalúa únicamente a x ya que al incrementarlo da False (0) y no importa lo que sigue retorna False (0).

Pregunta #7

```
char input7[] = "SSSWILTECH1\1\11W\1WALLMP1";

void pregunta7() { // switch, break continue
    int i, c;
    for ( i=2; (c=input7[i])!='\0'; i++) {
        switch (c) {
            case 'a' : putchar('i'); continue;
            case '1' : break;
            case  1  : while( (c=input7[++i])!='\1' && c!='\0') ;
            case  9  : putchar('S');
            case 'E' : case 'L': continue;
            default: putchar(c); continue;
        }
        putchar(' ');
    }
    putchar('\n');
}
```

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas
Putchar es una función de C que guarda en el buffer para que en el momento que haya cambio de línea imprima lo que tiene guardado.

El código funciona de la siguiente forma, se tiene un array de caracteres que contiene **"SSSWILTECH1\1\11W\1WALLMP1"** la función comienza haciendo un for hasta el final del carácter y hace un switch que equivale a una lista de condiciones que siguen ciertas especificaciones.

El for inicia en la posición 2 del array como no cumple ninguna especificación pasa a default y agrega la ‘S’ al buffer y continúa evaluando al siguiente carácter, igual con ‘W’, con la ‘I’, mientras que la ‘L’ tiene un caso que si es ‘L’ o ‘E’ continua al siguiente no lo agrega al buffer.

Eso continua hasta llegar al ‘1’, lo que se encuentra en el buffer es lo siguiente “SWITCH” cuando lee el ‘1’ tiene un break y prosigue hasta añadir el ‘ ’ en el buffer y continua con el siguiente que sería el ‘\1’ el cual entra en el while y como no es ‘\1’ sigue incrementando hasta llegar al último ‘\1’.

Seguidamente este ciclo se termina y como no tiene un continue cae en el siguiente caso agrega la ‘S’ al buffer y sigue en el siguiente caso y prosigue con lo que falta y así hasta terminar el arreglo de caracteres.

Termina imprimiendo en pantalla **“SWITCH SWAMP ”**.

Pregunta #8

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

```
int a8[] = {0,1,2,3,4};

void pregunta8() { // Vectores y punteros simples
    int i, *p;

    for( i=0; i <=4; i++ ) PR(d,a8[i]);
    NL;
    for( p= &a8[0]; p<=&a8[4]; p++)
        PR(d,*p);
    NL; NL;

    for ( p=&a8[0],i=1; i <=5; i++ )
        PR(d,p[i]);
    NL;
    for( p=a8,i=0; p+1<=a8+4; p++,i++ ){
        PR(d,*(p+i)); //*(p+i) es equivalente a es
    }
    NL; NL;

    for ( p=a8+4; p>=a8; p--) PR(d,*p);
    //Misma historia para atrás
    NL;
    for ( p=a8+4,i=0; i<=4; i++ ) PR(d,p[-i]);
    NL; //Para atras con -1
    for ( p=a8+4; p >=a8; p-- ) PR(d,a8[p-a8]);
    //p va en forma descendente mientras que a8 es
    NL;
}
```

Primero se crea una lista de 5 elementos [0,1,2,3,4] con variable a8, en esta función se imprime varios resultados el primero imprime el contenido de la lista a8:

```
a8[i] = 0 ---> 4210752
a8[i] = 1 ---> 4210756
a8[i] = 2 ---> 4210760
a8[i] = 3 ---> 4210764
a8[i] = 4 ---> 4210768
```

Nota: Si se incrementa un puntero este no puede ser restado o viceversa.

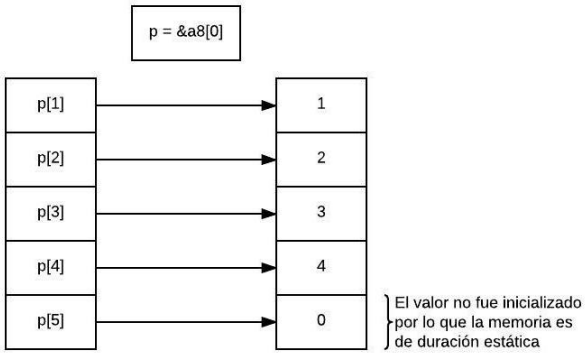
Se hizo una modificación para imprimir la posición en memoria donde se están guardando los datos que representa la columna derecha, además el i comienza en 0 hasta llegar a 4.

Para el siguiente se le asigna la posición en memoria a p que es un puntero este sin el operador * imprimiría la posición en memoria; sin embargo, el código lo que hace es ir incrementando la posición en memoria hasta llegar a 4.

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

```
a8[i] = 0       4210752     *p = 0
a8[i] = 1       4210756     *p = 1
a8[i] = 2       4210760     *p = 2
a8[i] = 3       4210764     *p = 3
a8[i] = 4       4210768     *p = 4
```

Seguidamente se da un ejemplo de lo que sucede si una lista se pasa de la cantidad total de elementos que es cuando se le asigna a p = &a8[5] este es equivalente a 0, ya que nunca fue inicializado ni se definió el tamaño de la lista.



El siguiente for es un equivalente de escribir p[i] = *(p+i), además este for tiene otra peculiaridad que incrementa p en cada iteración esto significa que el primer *(p+0) = 0, después p incrementa (p[1]) pero debido al i que también incrementa este corresponde a *(p+2) = 2 y así continua con ese patrón hasta llegar al a8+4.

EL siguiente for lo que hace es recorrer la lista de la cola hacia la cabeza decrementando la posición de memoria de p, igual sucede cuando se hace poniendo el índice [-i] por lo que imprime:

```
*p = 4  *p = 3  *p = 2  *p = 1  *p = 0
p[-i] = 4       p[-i] = 3       p[-i] = 2       p[-i] = 1       p[-i] = 0
```

El último for lo que hace es que p va en forma descendente mientras que a8 es la cabeza de la lista o sea 0 y se asigna a p = a8+4 o sea *p = 4 esto hace que la resta de ambos a8[p-a8] retorne el valor de la lista en esa posición la cual va desde 4 hasta 0.

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

Pregunta #9

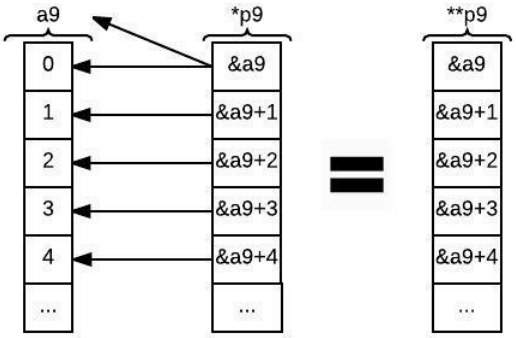
```
int a9[]={0,1,2,3,4};
int *p9[] = {a9,a9+1,a9+2,a9+3,a9+4};
int **pp9=p9;

void pregunta9() { // vectores de punteros
    PR2(d,a9,*a9);
    PR3(d,p9,*p9,**p9);
    PR3(d,pp9,*pp9,**pp9);
    NL;

    pp9++; PR3(d,pp9-p9,*pp9-a9,**pp9);
    *pp9++; PR3(d,pp9-p9,*pp9-a9,**pp9);
    *++pp9; PR3(d,pp9-p9,*pp9-a9,**pp9);
    ++*pp9; PR3(d,pp9-p9,*pp9-a9,**pp9);
    //Como es doble puntero pp9 lo único qu
    //en las diferentes formas que puede

    pp9=p9;
    **pp9++; PR3(d,pp9-p9,*pp9-a9,**pp9);
    *++*pp9; PR3(d,pp9-p9,*pp9-a9,**pp9);
    ++**pp9; PR3(d,pp9-p9,*pp9-a9,**pp9);
}
```

Lo primero es que se crean las siguientes cosas:



Después la función lo que hace es imprimir los valores:

```
a9 = 4210784    *a9 = 0
p9 = 4210816    *p9 = 4210784    **p9 = 0
pp9 = 4210816   *pp9 = 4210784   **pp9 = 0
```

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

Donde a9 es la ubicación en memoria y su puntero contiene el valor, *p9 tiene una lista de las direcciones de a9 con cada valor y **pp9 es el puntero de *p9.

Las siguientes impresiones se dan debido a las diferentes formas de incrementar el **pp9 que puede ser pp9++, *pp9++, *++pp9, ++*pp9 la primera aumenta a la dirección del siguiente elemento, la segunda pasa al tercer elemento por incremento y así sucesivamente, por lo que se da lo siguiente:

```
pp9-p9 = 1      *pp9-a9 = 1     **pp9 = 1
pp9-p9 = 2      *pp9-a9 = 2     **pp9 = 2
pp9-p9 = 3      *pp9-a9 = 3     **pp9 = 3
pp9-p9 = 3      *pp9-a9 = 4     **pp9 = 4
```

Las siguiente 3 impresiones son algo diferente ya que primero **pp9++ incrementa al siguiente elemento de la lista desde los valores, el siguiente lo *++*pp9 lo que hace es incrementar la posición del puntero *pp9 pero la dirección de pp9 y p9 incrementan de igual forma por lo que su diferencia da 1 y finalmente ++**pp9 aquí solo se incrementa el valor del puntero lo demás queda como antes.

```
pp9-p9 = 1      *pp9-a9 = 1     **pp9 = 1
pp9-p9 = 1      *pp9-a9 = 2     **pp9 = 2
pp9-p9 = 1      *pp9-a9 = 2     **pp9 = 3
```

Pregunta #10

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

```
int a10[3][3] = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
};

int *pa[3] = { a10[0], a10[1], a10[2] };
int *p10 = a10[0];
void pregunta10() { // multiples dimensiones
    int i;
    for( i=0; i<3; i++ )
        PR3(d, a10[i][2-i], *a10[i], *(*(a10+i)+i) );
    //el primero recorre la matriz diagonal de arriba izq a abajo der
    //segundo el puntero esta aspciado a la direccion a10[i][0]
    //el tercero es equivalente a a10[i][i]
    NL;
    for ( i=0; i<3; i++ )
        PR2(d, *pa[i], p10[i]);
    //*pa[i] tiene asociado el &a10[i] <=> &pa[1][0]
    //p10[i] es un arreglo de la primer fila de *pa

}
```

La siguiente función es como funciona una matriz, como se aprecia en la imagen de arriba se hace un for el cual imprime lo siguiente:

```
a10[i][2-i] = 3 *a10[i] = 1     *(*(a10+i)+i) = 1
a10[i][2-i] = 5 *a10[i] = 4     *(*(a10+i)+i) = 5
a10[i][2-i] = 7 *a10[i] = 7     *(*(a10+i)+i) = 9
```

La primera fila de a10[i][2-i] lo que hace es recorrer la matriz de forma diagonal desde el último elemento de la primera fila, el elemento del medio de la segunda y el primer elemento de la tercera fila.

El siguiente es el puntero *a10[i] que toma el primer elemento de cada fila esto se da ya que a10[i] = a10[i][0]. Mientras que la última fila es una representación de que *(*(a10+i)+i) = a10[i][i] esta imprime de forma diagonal, pero del primer elemento de la primera fila, segundo elemento de la segunda fila y tercer elemento de la tercera fila.

Finalmente, el siguiente for lo que hace es imprimir la lista pa en comparación a p10:

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

```
*pa[i] = 1      p10[i] = 1
*pa[i] = 4      p10[i] = 2
*pa[i] = 7      p10[i] = 3
```

En el caso de *pa[i] este tiene una lista de 3 elementos donde cada elemento es una lista de las 3 filas diferentes; sin embargo, en el caso del i este se va moviendo por la lista pa y devolviendo el primer elemento, ya que cuando se referencia a una lista este funciona como la cabeza de esa lista. En el caso de p10 es un arreglo pero únicamente de la primera fila por eso al incrementar i devuelve los valores de la primer fila de a10.

Pregunta #11

```
char *c[] = {
    "ENTER",
    "NEW",
    "POINT",
    "FIRST"
};
char **cp[] = { c+3, c+2, c+1, c };
char ***cpp = cp;

void pregunta11() { // sopa de punteros
    int i;

    printf("%s", **++cpp ); //POINT | ag
    printf("%s ", *--*++cpp+3 ); //ER
    printf("%s", *cpp[-2]+3 );//ST | Se
    printf("%s\n", cpp[-1][-1]+1 ); //EW
    //lo que significa que al moverse pa
}
```

En esta función se crea una lista con los strings los cuales tienen los siguiente:

```
c[i] = ENTER    &c[i] = Qa@
c[i] = NEW      &c[i] = Wa@
c[i] = POINT    &c[i] = [a@
c[i] = FIRST    &c[i] = aa@
```

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

En la imagen se muestra la lista c y su respectiva dirección en memoria. Luego la lista cp contiene los elementos, pero de atrás para adelante y cpp es igual a cp.

```
cpp[0] = aa@    cpp[1] = [a@    cpp[2] = Wa@    cpp[3] = Qa@

*cpp[0] = FIRST *cpp[1] = POINT *cpp[2] = NEW   *cpp[3] = ENTER
```

Ahora bien, lo que hace la función es imprimir el siguiente elemento después de “FIRST” en este caso sería “POINT”, después notamos que el *cpp[0] pasa a ser “POINT” y los demás cambian de número.

```
POINT
*cpp[i] = POINT         &*cpp[i] = 4215088
*cpp[i] = NEW           &*cpp[i] = 4215080
*cpp[i] = ENTER         &*cpp[i] = 4215072
```

Seguidamente se hace lo siguiente se resta para irse a “NEW” y suma puntero para ir a “ENTER” y agarra del 3 en adelante.

```
ER
*cpp[-i] = ENTER                &*cpp[-i] = 4215072
*cpp[-i] = POINT                &*cpp[-i] = 4215088
*cpp[-i] = FIRST                &*cpp[-i] = 4215096
```

Para la siguiente impresión se pide que *cpp[-2]+3 esto significa que se devuelve en el array incrementado y se pierde new del array con punteros por lo que termina en “FIRST” tomando solo el “ST”.

Para finalizar se encuentra cpp en la posición de la cabeza que es “POINT” pero el arreglo esta ordenado lo que significa que al moverse para atrás consigue “NEW” y para adelante consigue a “FIRST” como se mueve para atrás y toma el segundo elemento -1 y a partir del +1 esto concluye en “EW” y termina imprimiendo:

```
POINTER STEW
```

Integrantes:
Valeria Garro
Luis Pablo Monge
Josué Vargas

Tabla de precedencia y asociatividad

Nivel	Operadores	Descripción	Asoci.
1	`() [] -> .`	Acceso a un elemento de un vector y paréntesis	Izquierdas

2	`+ - ! ~`
	`* &`
	`++ --`
	`(cast) sizeof`
		Signo (unario), negación lógica, negación bit a bit
		Acceso a un elemento (unarios): puntero y dirección
		Incremento y decremento (pre y post)
		Conversión de tipo (*casting*) y tamaño de un elemento
			Derechas

3	`* / %`	Producto, división, módulo (resto)	Izquierdas
4	`+ -`	Suma y resta	Izquierdas
5	`>> <<`	Desplazamientos	Izquierdas
6	`< <= >= >`	Comparaciones de superioridad e inferioridad	Izquierdas
7	`== !=`	Comparaciones de igualdad	Izquierdas
8	`&`	Y (*And*) bit a bit (binario)	Izquierdas
9	`^`	O-exclusivo (*Exclusive-Or*) (binario)	Izquierdas
10	`|`	O (*Or*) bit a bit (binario)	Izquierdas
11	`&&`	Y (*And*) lógico	Izquierdas
12	`||`	O (*Or*) lógico	Izquierdas
13	`?:`	Condicional	**Derechas**
14	`= *= /= %= += -= >>= <<= &= ^= |=`	Asignaciones	**Derechas**
15	`,`	Coma	Izquierdas