Ejercicios resueltos de la relación 1: punteros

1.4

```
int v[100];
for (int i=0; i<100; ++i)
        *(v+i) = 1;
```

- alternativamente:

```
int v[100];
int *p=v;
while (p!= v+100) {
        *p = 1;
        ++p;
}
```

1.5

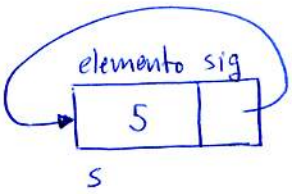
```
int numeros[1000];
for (int i=0; i<1000; ++i)
    if (*(v+i) < 0)
        *(v+i) = -*(v+i);
```

1.6

```
int numeros[1.000];
int *p = numeros;
int *final = numeros + 1000;
....
while (p != final){
        if (*p < 0)
            *p = -*p;
        ++p;
}
```

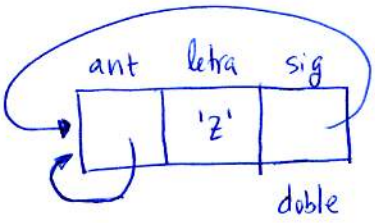
1.7

[a]



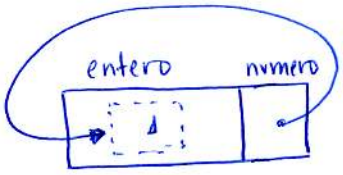

```
struct Ciclo {
        int elemento;
        Ciclo * sig;
};
Ciclo s;
s.elemento = 5;
s.sig = & s;
```

[b]



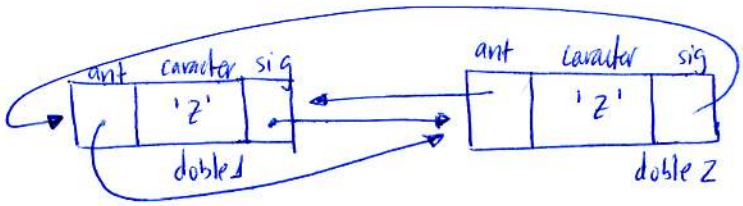

```
struct Ciclo_Doble {
        char letra;
        Ciclo_Doble * sig, * ant;
};
Ciclo_Doble doble;
doble.letra = 'z';
doble.sig = doble.ant = & doble;
```

[c]



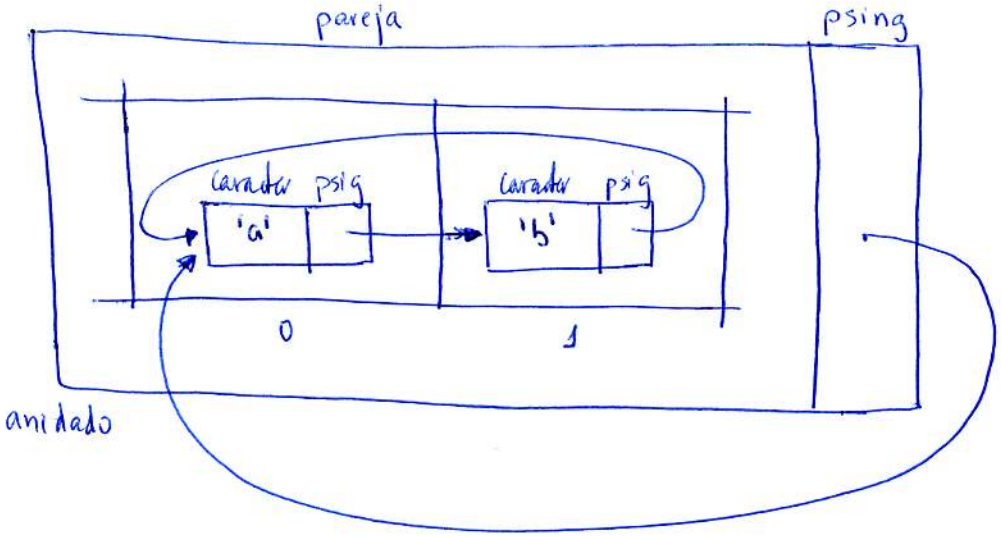

```
struct Ciclo_Interior {
        int entero;
        int * numero;
};
Ciclo_Interior s;
s.entero = 1;
s.numero = & (s.entero)
```

[d]



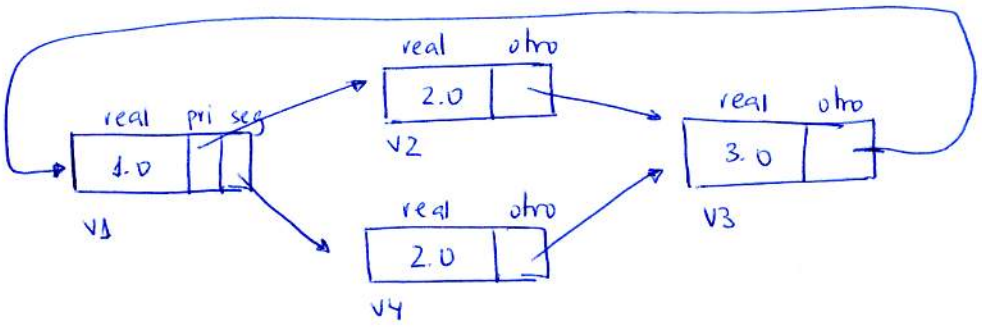

```
struct Doble {
        char caracter;
        Doble *sig, *ant;
};
Doble doble1, doble2;
doble1.caracter = doble2.caracter = 'z';
doble1.ant = doble1.sig = &doble2;
doble2.ant = doble2.sig = &doble1;
```

[e]



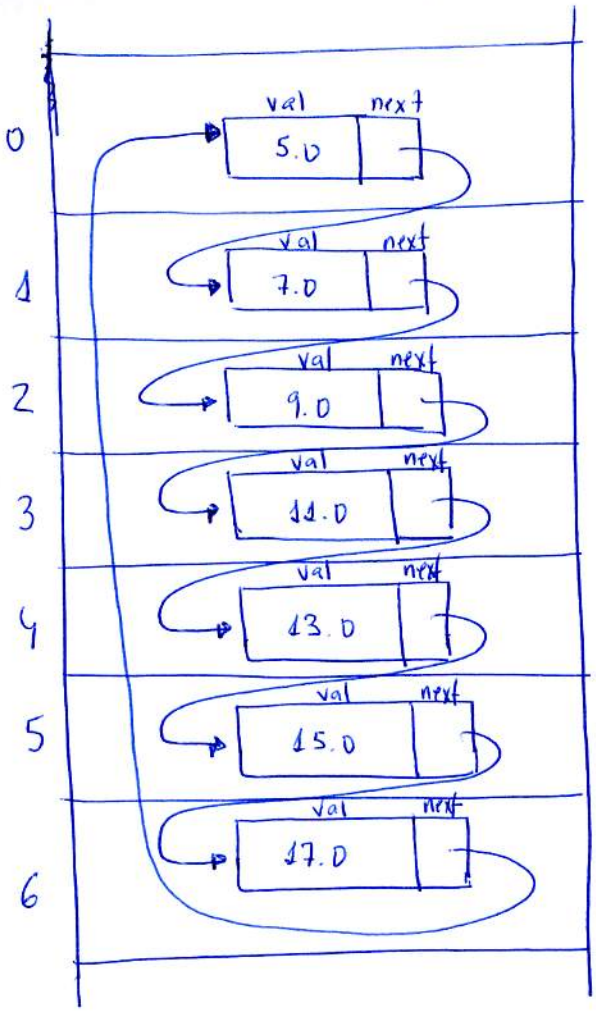

```
struct Simple {
        Char caracter;
        Simple *psig;
};
struct Compleja {
        Simple pareja[2];
        Simple *psing;
};
```

```
Compleja anidado;
anidado.pareja[0].caracter = 'a';
anidado.pareja[1].caracter = 'b';
anidado.pareja[0].psig = &(anidado.pareja[1]);
anidado.pareja[1].psig = &(anidado.pareja[0]);
anidado.psing = &(anidado.pareja[0]);
```



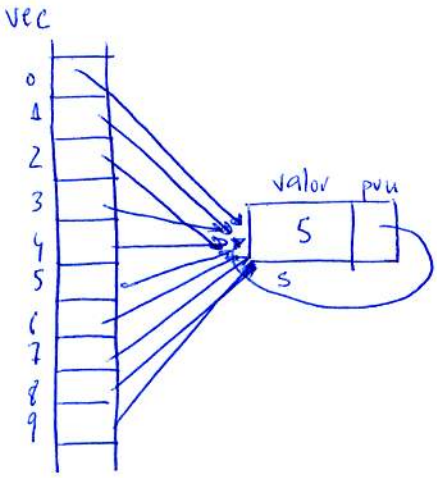

```
Struct Final;   //Hay que adelantar esta definicion

struct Simple{
        float real;
        Final *otro;
}
struct Inicial{
        float real;
        Simple *pri, *seg;
}

struct Final{
        float real;
        Inicial *otro;
}

Inicial V1;
Simple V2, V4;
Final V3;
V1.real = 1.0;
V1.pri = &V2;
V1.seg = &V4;
V2.real = V4.real = 2.0;
V2.otro = V4.otro = &V3;
V3.real = 3.0;
V3.otro = &V1;
```

[g]



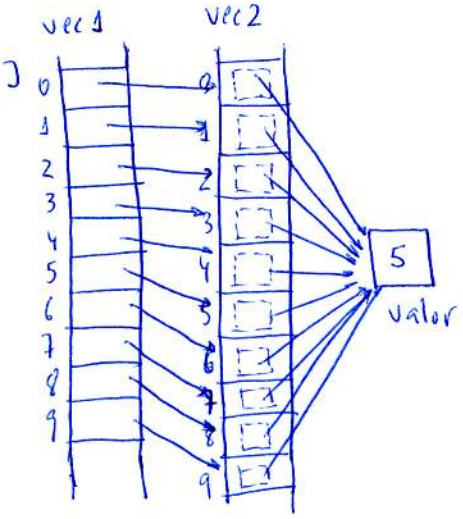

```
struct Enlazada {
        float val;
        Enlazada *next;
};
Enlazada encadenado[7];
for (int i=0; i<6; ++i) {
        encadenado[i].val = 5.0 + 2*i;
        encadenado[i].next = encadenado + i + 1;
}
encadenado[6].val = 17.0;
encadenado[6].next = encadenado;
```

[h]



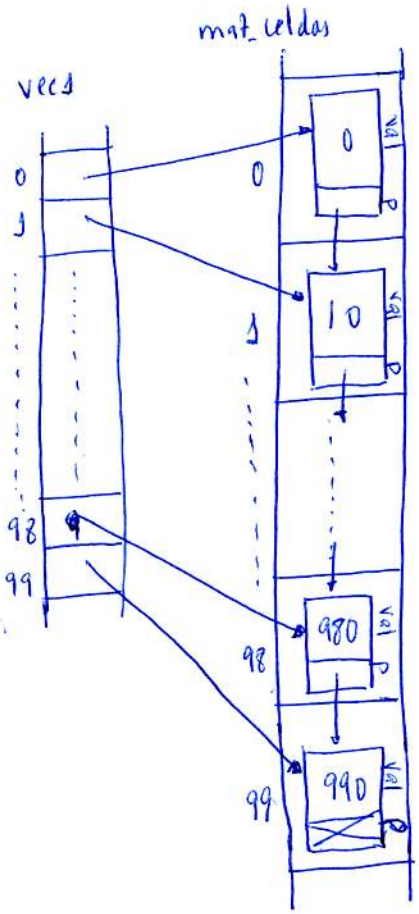

```
struct Simple {
        int valor;
        Simple * pun;
};
Simple s;
Simple * vec[10];
for (int i=0; i<10; i++)
     vec[i] = &s;
```

[i]



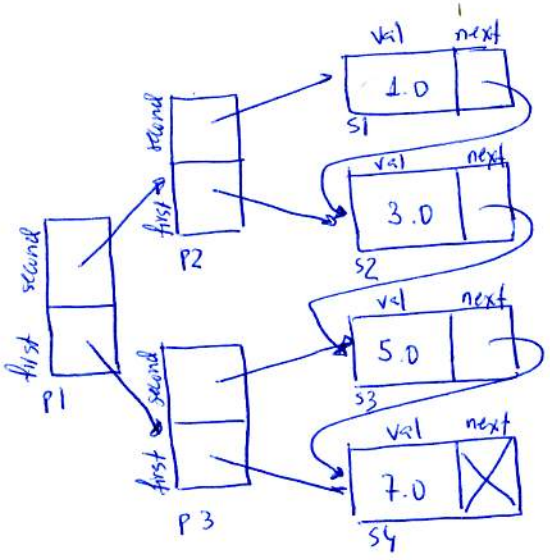

```
int * vec2[10];
int ** vec1[10];
int valor = 5;
for (int i=0; i<10; ++i) {
     vec1[i] = &vec2[i];
     vec2[i] = &valor;
}
```

[j]




```
struct Enlazada {
        int val;
        Enlazada * p;
};
Enlazada mat_celdas[100];
Enlazada *vec1[100];
for (int i=0; i<100; ++i) {
     mat_celdas[i].val = i * 10;
     mat_celdas[i].p = mat_celdas+i+1;
     vec1[i] = &mat_celdas[i];
}
mat_celdas[99].p = 0;
```

[K]




```
struct Enlazada{
    float val;
    Enlazada * next;
};
struct Mitad{
    Enlazada *first, * second;
};
struct Raiz{
    Mitad *first, * second;
};
```

Enlazada s1, s2, s3, s4;

Mitad p2, p3;

Raiz p1;

p1.first = &p3; p1.second = &p2;

p2.first = &s2; p2.second = &s1;

p3.first = &s4; p3.second = &s3;

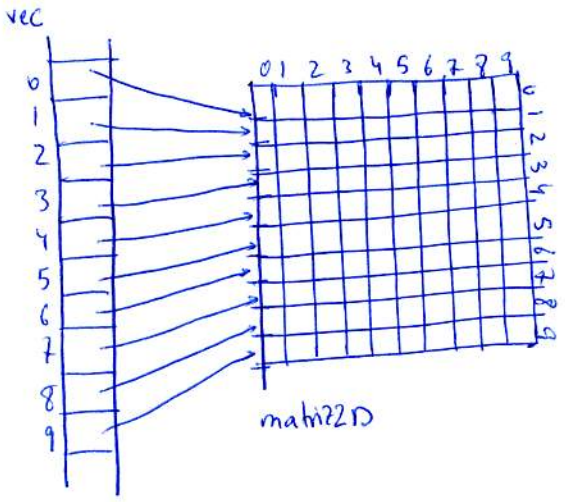
s1.val = 1.0; s1.next = &s2;

s2.val = 3.0; s2.next = &s3;

s3.val = 5.0; s3.next = &s4;

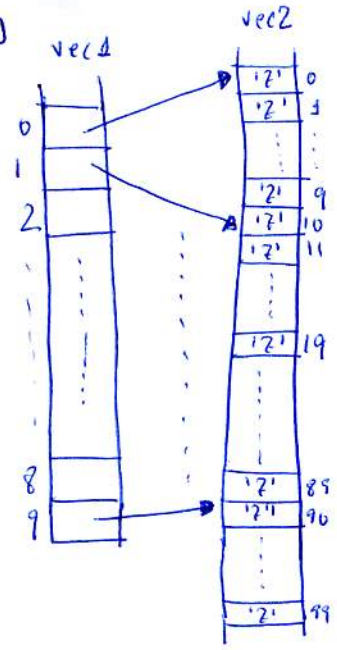
s4.val = 7.0; s4.next = 0;

[L]




```
int matriz2D[10][10];
int *vec[10];
for (int i=0; i<10; ++i)
    vec[i] = matriz2D[i];
```

[m]




```
char *vec1[10];
char *vec2[100];
for (int i=0; i<100; ++i)
    vec2[i] = 'z';
for (int i=0; i<10; ++i)
    vec1[i] = vec2 + i*10;
```

1.8

```
const int n=...;
float v[n];
float *izq, *der;
.....
izq = v;
der = v+n-1;
while (izq < der) {
    While (*izq < v[0] && izq < der)
        ++izq;
    While (*der > v[0] && der > izq)
        ++der;
    float aux = *izq;
    *izq = *der;
    *der = aux;
}
```

Ejercicios resueltos de la relación 2: funciones

2.1 La sentencia `int &temp = p;` es incorrecta porque los tipos son incompatibles: temp es una referencia a entero y p es un puntero a entero que no puede inicializar a temp (para hacerlo habríamos de usar un entero). En consecuencia la función daría un error de compilación

2.2 Teóricamente, la función parece pensada para intercambiar los valores de los punteros p y q (lo que aparentemente se hace según el código interno de la función), pero al pasar éstos por valor el efecto es nulo, de forma que p y q continuan tras la ejecución de la función con los mismos valores.

2.3

```
void swap_puntero (int *&p, int *&q)
  {
    int *temp;
    temp = p;
    p = q;
    q = temp;
  }
- - - - -
int *p1, *p2;
- - - -
swap_puntero (p1, p2);
```

} Llamada

2.4

```
void swap_puntero (int **p, int **q)
  {int *temp;
   temp = *p;
   *p = *q;
   *q = temp;
  }
- - - - -
- - - - -
int *p1, *p2;
- - - -
swap_puntero (&p1, &p2);
```

} Llamada

2.5 Las funciones dan lugar a errores de compilación porque no es posible su enlazado: La llamada a la función `par` desde la función `numero_pares` no es compatible con la cabecera declarada.

La llamada pasa como parámetro un puntero a entero constante: `(v+i)`, pero la función `par` espera un puntero a entero no constante y como sabemos no podemos asignar un `const int *` a un `int *`

Para solucionar el problema bastaría que la cabecera de `par` fuese:

```
bool par (const int *p)
```

2.10 La función intercambia recibe una estructura `Par` por valor, con lo que tal estructura se copiará al hacer la llamada y por tanto a la vuelta los valores de los campos quedarán como antes de la llamada.

Por otra parte, los campos de esa estructura que pasamos por valor son punteros, y por tanto la copia de la estructura tiene dos punteros que apuntan a la misma zona que la estructura original de forma que si cambiamos los objetos apuntados por estos ~~punteros~~, tal cambio se verá reflejado en la estructura original afectando a los objetos que se apuntan originariamente.

El cuerpo de la función hace 2 intercambios, el de los punteros y el de los enteros que apuntan. El primero no afecta en el código de llamada (al haberse pasado por valor), pero los enteros apuntados sí se intercambian (al no estar copiados).

Conclusión. Se escribirá que los punteros están igual, pero nada acerca de los enteros puesto que estos sí que han cambiado.

2.8

4. | int v | v | float v |

Correcto, porque aunque el tipo del parámetro actual y formal no coinciden, se pasa por valor un flotante obtenido a partir de la conversión implícita de un int.

2. | int m[] | m | int *mat |

Incorrecto. La declaración del vector es incorrecta, porque con esa notación se ha de especificar el tamaño.

3. | float mat[5] | mat | float *f mat |

Incorrecto. Si pasamos un float * por referencia, el parámetro actual ha de ser un objeto de tipo float *, y el nombre de un vector no es un objeto de tipo float * (con independencia de que sepamos que puede convertirse a ese tipo).

4. | const int v[10] | v | int *mat |

Incorrecto. No puede pasarse un const int * a un int *

5 | int m[] | m | int mat[10] |

Incorrecto. Igual que en 2. La declaración no es válida

6 | int m[3][5][7] | m | int mat[][5][7] |

Correcto. Los tipos coinciden (puede además dejarse sin especificar la primera dimensión de una matriz.

7. | float v[5] | v+2 | const float mat[] |

Correcto. Puede pasarse un float * a un const float * (más restrictivo)

8. | int m[] | m | int mat[][5] |

Incorrecto. La declaración no es válida, pero aunque se hubiese puesto un valor entre [] seguiría siendo incorrecto al no coincidir los tipos.

9. `float f  f  double f`

Correcto. El float se convierte implícitamente en double

10. `float f  &f  double &f`

Incorrecto. Se espera la referencia de un objeto de tipo double y se le manda un float. Aquí no hay conversión implícita: estamos tratando con "punteros" de forma que una referencia a double (que ocupa p.ej. 8 bytes) no puede inicializarse mediante un float (que ocupa p.ej. 4 bytes)

11. `bool mat[5][7]  f mat[3][2]  const bool mat[]`

Correcto. La dirección de un elemento de la matriz es un bool *, que puede asignarse (convertirse) a un const bool *

12. `char mat[3][5]  mat[0]  char *mat`

Correcto. mat[0] es un vector de char que puede pasarse como un char *

13. `int *m[10]  m  int **mat`

Correcto. Un vector de punteros se puede convertir a un puntero al primer elemento, en este caso, un puntero a puntero.

14. `const double v  &v  double v[]`

Incorrecto. La dirección de v es un const double * y no puede asignarse a un double *

15. `int **m  m  int mat[][]`

Incorrecto. El parámetro no es válido. Se puede prescindir del tamaño de la primera dimensión pero no de las demás.

16. `int mat[5][7][9][11]  &mat[0][0][0][0]  int *p`

Correcto. Pasamos la dirección de un entero, es decir, un puntero a entero lo que coincide con el parámetro formal.

17. | int * p | & p | int * mat[] |

Correcto. Pasamos un int ** que es lo que indica el parámetro formal

18. | float * p | p | float *& p |

Correcto. Los tipos coinciden (estamos pasando por referencia un float *)

19. | int m[3][5][7] | m | int * mat[5][7] |

Incorrecto. Declaramos una matriz de enteros que se pasa a una matriz de punteros a enteros haciendose la "suposición" de que hay punteros almacenados.

20. | int mat[5][7][9][11] | mat[2][4][3] | const int * m |

Correcto. Declaramos una matriz, pasamos un vector que puede convertirse a int * que puede sin problemas pasarse a un const int *

21. | float * p | p+5 | float *& p |

Incorrecto. Se pasa un float * pero como resultado de una expresión y una referencia no acepta ser inicializada por una expresión, y espera un lugar donde se almacena un float *

22. | double * p | p+2 | float * p |

Incorrecto. Son distintos tipos punteros

23. | int m[5][10] | m | int * mat[] |

Incorrecto. Se declara una matriz de enteros (no existe ningún puntero almacenado), y espera un puntero o vector de punteros a entero.

24. | int * mat[5] | & mat[5] | const int **& p |

Incorrecto. Se pasa un int ** y espera un const int **. Son tipos incompatibles (2 punteros son distintos si apuntan a tipos distintos y int * ≠ const int *)

25. | const bool * mat | mat+4 | bool *& p |

Incorrecto. Un const bool * no puede pasarse a un bool *

26 | — | 5 | int & val |

Incorrecto. Espera un lugar donde se almacena un entero, no un literal.