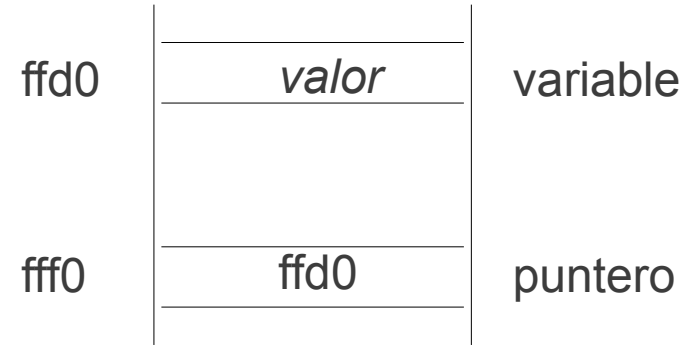
PUNTEROS

- **Punteros en C**
  - Un puntero es una variable que almacena direcciones de memoria

```
tipo *puntero;
tipo variable;
puntero = &variable;
```



A partir de la última sentencia `puntero` y `variable` permiten modificar el valor almacenado en la dirección de memoria `ffd0`

  - Un puntero está asociado a un **tipo** y contiene **como valor una dirección**

- **Punteros en C**
  - Como un puntero posee una dirección de memoria esta puede ser asignada a un puntero también

```
tipo **puntero2;
puntero2 = &puntero;
```

  - Operaciones
    - Asignarle un valor

```
puntero=direccion
puntero=(cast)puntero
```

el operador **& (operador de dirección)** retorna la dirección de una variable

```
puntero=&variable
```

- **Operador de desreferencia (*)**
  - Es posible modificar el valor de la variable que “apunta” el puntero

```
int a=10;
int *ptra;
ptra = &a;
*ptra=50;

printf(“%d”, a);      ----->  50
a=100;
printf(“%d”, *ptra); -----> 100
```

- **Pasaje de argumentos por valor y referencia**
  - En C, por defecto, los parámetros a una función se pasan por valor
  - Las variables pasadas a una función no son modificadas, en la función se crean *variables automáticas* con los valores de las variables empleadas en la invocación
  - Las variables automáticas se destruyen al retornar la función

Pasaje por valor

```
int suma_dobles(int a, int b) {
   //a posee una copia del valor de x, b de y, vars. automáticas
    a = a*2;          // variable local a cambia de valor
    b = b*2;          // variable local b cambia de valor
    return a+b;
 }

 int main() {
    int x=10, y=20;
    printf("Variables x=%d y=%d\n", x, y);  // variables x=10 y=20
    suma_dobles(x, y);
    printf("Variables x=%d y=%d\n", x, y);  // variables x=10 y=20
    // x e y no cambian luego de invocar a suma_dobles
    return 0;
 }
```

- **Pasaje de argumentos por valor y referencia**
  - Si se emplean direcciones en la invocación a una función, las variables automáticas reciben direcciones (referencias), no valores
  - Toda modificación dentro de la función que altere los valores almacenados en dichas direcciones serán visibles al retornar dicha función

Pasaje por referencia

```
int suma_dobles(int *a, int *b) {
   //a y b son punteros, reciben direcciones
    *a = (*a)*2;          // cambia de valor el lugar de memoria de a
    *b = (*b)*2;          // cambia de valor el lugar de memoria de b
    return (*a)+(*b);
 }

 int main() {
    int x=10, y=20;
    printf("Variables x=%d y=%d\n", x, y); // variables x=10 y=20
    suma_dobles(&x, &y);                   // se pasan direcciones
    printf("Variables x=%d y=%d\n", x, y); // variables x=20 y=40
    //x e y cambiaron luego de invocar a suma_dobles
    return 0;
 }
```

- **Aritmética de punteros**
  - Los punteros pueden oficiar de operandos para algunos operadores aritméticos

  - Es posible **sumar o restar una constante** numérica a un puntero

```
tipo a=valor;
tipo *ptra=&a;
tipo *ptrb=NULL;
ptrb=ptra+Constante;
          //esto es en realidad ptra+Constante*sizeof(tipo);
```

También valen

```
ptra = ptra + Constante
ptra += Constante   (+=)
ptra++              (++)
```

Mismas operaciones con (-)

En el contexto de un array o un espacio de memoria dinámicamente obtenido la **resta de punteros** retorna la cantidad de elementos de tamaño del tipo asociado entre las direcciones que apuntan sendos punteros. Ejemplo.

- **Puntero NULL**
  - NULL es una constante simbólica que representa un valor especial de dirección de memoria, un valor nulo
  - NULL está definido en <stdio.h>
  - Un “puntero NULL” es un puntero con este valor
  - NULL es útil para protegerse del acceso no controlado a memoria

```
int a=10;
int *ptra;     //Inicializado “apuntando” a alguna dirección
*ptra = 5000; //Error: modificación de una dirección de memoria desconocida
```

Forma correcta

```
int a=10;
int *ptra=NULL;
 Intentar modificar el valor de lo que “apunta” ptra es un error
```

  - Es un error desreferenciar un “puntero NULL”
    Se puede comprobar antes si un puntuero es NULL mediante
    ptra == NULL ptra != NULL

- **Puntero void**
  - Es posible asignar directamente un puntero a otro si ambos poseen el mismo tipo
  - Si los tipos son diferentes debe realizarse una conversión explícita de tipos mediante cast. Sintaxis: *(tipo)puntero*
  - El puntero void es un "puntero genérico" capaz de representar cualquier tipo de puntero
    Todos los tipos de puntero pueden asignarse a un puntero void sin cast, pero un puntero void solo puede asignarse a otro de otro tipo si se emplea cast
  - Se recibe un error si se desreferencia un puntero void

```
void *ptrv;
int i=5;                    ptrv=ptri;            //OK
double d=10.7;              ptrv=ptrd;            //OK
int *ptri=&i;               ptrd=ptrv;            //Error
double *ptrd=&d;            ptrd=(double*)ptrv;   //OK
```

Punteros a string (cadena)

- En C no existe el tipo string ni se cuenta con operadores para su tratamiento, como un operador de contatenación de cadenas, por ejemplo.

- Las strings son representadas como arrays de caracteres finalizados en '\0' Una cadena de n caracteres en realidad emplea el espacio de (n+1) caracteres

- El acceso al contenido de una string se hace a través de un puntero a su primer elemento. Ej: printf para imprimir una cadena

```
int printf(const char *format, arg_1, arg_2, arg_3, ... , arg_n);
```

- Definiciones

```
char msg1[]="texto de ejemplo";   // msg1 es un array de 17 elementos
                                  // msg1 siempre apunta a la misma dir
                                  // se pueden modificar sus componentes

char *msg2="texto de ejemplo";    // msg2 es un puntero a char
                                  // apunta a una cadena constante
```

- **Punteros a string (cadena)**

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t){
    int i=0;

    while ((s[i] = t[i]) != '\0')
        i++;
}


/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t) {
    int i=0;

        while ((*s = *t) != '\0') {          while (*s++ = *t++) ;
            s++;
            t++;
        }
}
```

- **Punteros y arrays**
  - En C los arrays y punteros están muy relacionados
  - Toda operación que puede realizarse con arrays e indexación puede lograrse con punteros (arimética y desreferencia)

```
int a[10];  // a[i] es el i-ésimo elemento de a, con i=0..9
int *pa;
pa = &a[0]; // puntero al primer elemento del array a
// *pa y a[0] acceden al mismo lugar de memoria
// (pa+1) apunta al 1-ésimo elemento del array a
// (pa+i) apunta al i-ésimo elemento del array a
   *(pa+i) = a[i]
```

  - El **nombre de un array** es un sinónimo (variable constante, no puede cambiar de valor) de la dirección del elemento inicial, es **un puntero al primer elemento de si mismo**

```
*(a+i) = a[i] = *(pa+i)
 (a+i) = &a[i] = (pa+i)
(pa+i) = &pa[i]
       // pa[i] es el i-ésimo elemento de a, con i=0..9
```

- **Punteros a funciones**
  - El nombre de una funcion por si solo no es una invocación
    El operador de invocación de funciones es el ()

```
int f(int i, char c) { body }
```

La expresión `f(3,'h')` posee tipo `int`

¿y la expresión `f`?
`f` es una dirección (constante) de memoria relacionada con el principio del segmento de memoria reservado para las variables automáticas y locales de dicha función.

Esta dirección puede almacenarse en un puntero del siguiente tipo

```
int (*)(int,char)  // un puntero a función que recibe parámetros int y char
                   // y retorna int
int (*g)(int,char);
g = f;
g(3, 'h') == f(3, 'h')
```

- **Punteros a funciones**

  `tipo nombre_funcion (tipo1 arg1, tipo2 arg2, tipo3 arg3)`

  `nombre_funcion` es una dirección compatible con el puntero a función de tipo

  `tipo (*)(tipo1, tipo2, tipo3)`

Ejemplos

```
int (*f) (int, int)
// puntero a una función que recibe dos int y retorna int
void (*mensaje)()
// puntero a una función que no recibe argumentos y retorna void
double *(*g)(char) // puntero a una función que recibe char y retorna double*
void qsort(void*, size_t, size_t, int (*cmp)(const void*, const void*))
void* bsearch(const void*, const void*, size_t, size_t,
                                        int (*cmp)(const void*, const void*))
//el último argumento de ambas funcione es un puntero a una fución que
//recibe dos constantes a puntero void y retorna un int
char **argv             // puntero a puntero char
int dias[31]            // array de 31 punteros a int
int (*dias)[31]         // puntero a un array de 31 int
char (*(*x())[])()      // función que retorna
                        // un puntero a un array[] de punteros a
                        // función que retorna char y no recibe argumentos
```

- **typedef**
  - Permite introducir nuevos nombres de tipos
  - Útil para
    - documentar y comprender el código: léxico del dominio
    - lidiar con declaraciones complicadas (punteros a función)

  - Ejemplos

```
typedef unsigned long int dni;
typedef long double monto;

dni d1, d2;       // dos variables tipo dni
dni familia[6];   // array de dni
int (*)(dni);     // puntero a función que recibe un dni y retorna int (check_dni)
// Es más simple pensar en dni's que en enteros largos sin signo
```

Punteros – Errores comunes

- **Ausencia de inicialización**
  Modificar un area de memoria no conocida

```
int *ip;
*ip=5000; // donde se está escribiendo? Es seguro?
```

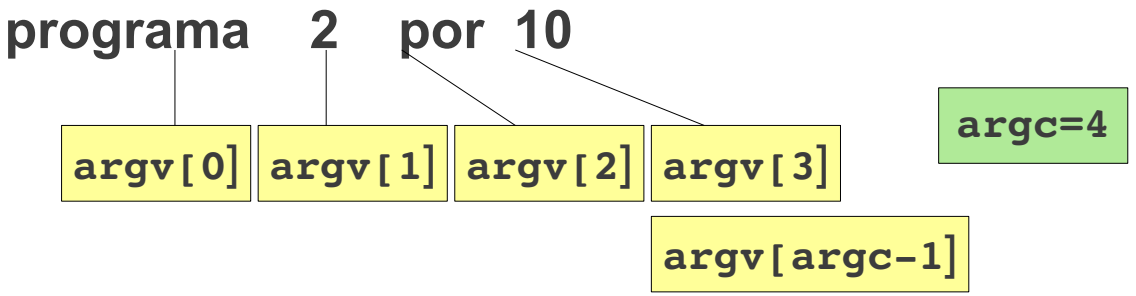
- **Problemas de alias**
  Puede dejar punteros "colgados"

```
int *ip, *jp;
ip=solicitud_memoria_dinamica();
jp=ip;
liberar_memoria_dinamica(ip);
// Adonde apunta ahora jp?
```

- **Pérdida de memoria**

```
int *ip=solicitud_memoria_dinamica(); //pedido 1
int *jp=solicitud_memoria_dinamica(); //pedido 2
jp = ip;
// Se pierde lo pedido en segunda instancia
```

- **Argumentos en línea de comandos**
  - Al invocar a main se pueden emplear dos argumentos
    - **argc** (argument count) **int**
    - **argv** (argument vector) **char *[]** puntero a un arreglo de cadenas una cadena == un argumento




```
#include <stdio.h>

int main (int argc, char *argv[]) {
   int i;
   for (i = 0; i<argc; i++)
      printf("argv[%d]=%s\n", i, argv[i]);
   return 0;
}
```

```
$ ./args 4 + cadena
argv[0]=./args
argv[1]=4
argv[2]=+
argv[3]=cadena
```