Punteros

- Una variable de tipo puntero almacena una dirección de memoria.
- Dicha dirección de memoria puede ser la que almacena una variable o bien donde se encuentra el código de una función.
- Al declarar el puntero se indica el tipo de aquello a lo que apunta, de modo que pueda "interpretarse" lo que se encuentra en esa dirección.

Punteros

- Para declarar una variable de tipo puntero lo hacemos agregando el operador * entre el nombre de la variable y el tipo al que apunta

```
long l = 3000; /* variable tipo long a la que
                  asignamos el valor 3000 */
long k;    // otra variable long
long *pl;  // pl es de tipo "puntero a long"
```

- Para que una variable de tipo puntero "apunte" a una variable del tipo a la que apunta, usamos el operador de dirección &

```
pl = &l; /* ahora pl apunta a l, ya que con &l lo
            que obtenemos es la dirección de l */
```

Punteros

- Para obtener el valor de la variable a la cuál apunta una variable de tipo puntero usamos el operador de dirección *

```
k = *pl; /* ahora k vale 3000 ya que *pl (lo
            apuntado por pl) equivale a l, es
            decir, esta última línea equivale a
            k = l */
```

Punteros Genéricos

- Podemos declarar un puntero “genérico” dicendo que es un puntero a “vacío”

```
void *p; // p es un puntero genérico

p = pl; /* puedo asignar cualquier tipo de puntero
           a uno generico y también puedo asignar
           un puntero genérico a uno "concreto"
           (hace un cast automático) */

k = *(long*)p; /* pero no puedo tomar lo apuntado
                  por un puntero genérico, debo
                  indicar mediante un cast como
                  debo interpretar el puntero */
```

Ejemplo de intercambio

- Como C usa parámetros por valor, el siguiente ejemplo no funciona

```c
void
intercambio(int i, int j)
{
        int aux;

        aux = i;
        i = j;
        j = aux;
}
```

```c
int main()
{
        int i,j;

        i = 5;
        j = 7;
        intercambio(i, j);

        printf("i vale %d y j "
                "vale %d", i, j);
        /* imprime: i vale 5 y j
                vale 7 */
}
```

- No funciona por que los valores de i y de j dentro de la función intercambio son copias locales de los valores que le pasaron a la función.

Para que haga lo que pretendemos

- Debemos usar punteros del siguiente modo

```c
void
intercambio(int *i, int *j)
{
        int aux;

        aux = *i;
        *i = *j;
        *j = aux;
}
```

```c
int main()
{
        int i,j;

        i = 5;
        j = 7;
        intercambio(&i, &j);

        printf("i vale %d y j "
                "vale %d", i, j);
        /* imprime: i vale 7 y j
                vale 5 */
}
```

Asignación de memoria

- En stdlib.h se define la fución malloc, que significa “memory allocation”, es decir asignación de memoria (literalmente: alojamiento de memoria)
- La definición de malloc es

```
void * malloc (size_t size );
/* size_t está declarada en stddef.h y típicamente es
   unsigned long int */
```

- Dado un tamaño size devuelve un **puntero genérico** (void *) a una zona de memoria del tamaño pedido o NULL si no hay memoria suficiente
- NULL es una constante definida en stddef.h que indica un puntero nulo o vacío (en la práctica, cero)

Asignación de memoria

- Dado un puntero, en lugar de “apuntarlo” a una variable puedo pedir memoria

```
double *dp = malloc(sizeof(double));
/* sizeof devuelve el tamaño en bytes del tipo o variable que
se pase como argumento. Se calcula en tiempo compilación */
```

- Es una mejor práctica

```
double *dp = malloc(sizeof(*dp));
/* si cambio el tipo de dp no hace falta cambiar el argumento
de sizeof, lo que es muy útil si hay varios malloc con dp */
```

- Finalmente debo liberar la memoria pedida con malloc mediante la función free

```
free(dp);
```

- Recordar: *p hace que “reviente” el programa si p vale NULL

Arreglos

- Un vector (arreglo de una dimensión) se declara en C usando [ ]

```
int v[10]; /* declara un vector de enteros, de 10
              elementos */
```

- Los subíndices en C comienzan en 0 por lo tanto si quiero mostrar los 10 valores del vector (supongamos que en algún momento ya lo cargamos) podemos hacerlo así

```
for (i = 0; i < 10 ; i++)
        printf("v[%d] = %d\n", i, v[i]);
```

Arreglos

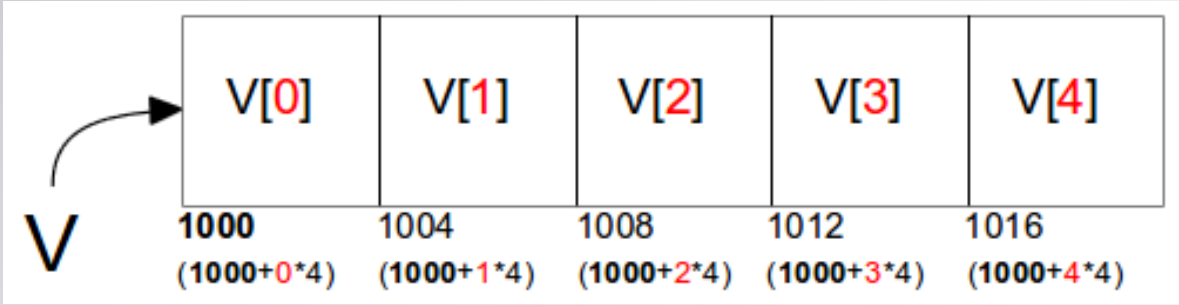
- Un vector también puede iniciarlizarse al momento de declararlo

```
int v[5] = {1, 6, 8, 3, 9};
int w[] = {3, 12, 7};
int x[10] = {0};
```

- Notar que en el caso de **w** no se indicó el tamaño explícitamente, pero al iniciarlo con 3 elementos el compilador lo genera de ese tamaño.
- También podemos indicar el tamaño e inicializar una cantidad de valores menor al tamaño, en cuyo caso los valores no inicializados explícitamente, se inicializan, implícitamente, en cero. Por ejemplo el vector **x** tendrá sus 10 elementos inicializados en cero.

Arreglos según C

- Para lenguaje C un arreglo no es más que un sector de memoria contigua con capacidad para guardar la cantidad de elementos declarados.
- Así si declaro int v[5] guarda lugar en memoria para almacenar 5 int consecutivos.



- Esto explica, porqué el primer subíndice es el cero, ya que el primer elemento es el que está **desplazado** cero "elementos" con respecto al inicio del vector.

Punteros y arreglos

- En lenguaje C un arreglo, es decir su identificador, se lo considera un puntero constante al al primer elemento del arreglo
  - La declaración

    ```
    int v[5];
    ```

    podemos verla como

    ```
    int * const v;
    ```

    salvo que la primera reserva espacio en memoria para los 5 int y la segunda no
- El operador [ ] sirve para obtener el contenido del elemento “desplazado” tantos elementos con respecto al principio del arreglo como indique el argumento

Punteros y arreglos

- Así v[0] es el contenido del elemento que está desplazado “cero” posiciones con respecto al incio del arreglo, o sea, el primer elemento
- En tanto que v[2] es el tercer elemento, que está desplazado dos elementos respecto al incio del arreglo
- **OJO**: desplazado n elementos, **NO** n bytes. La cantidad de bytes desplazados depende del tipo de elementos (conocido como “aritmética de punteros”)

Punteros y arreglos

- Si
  - v es un arreglo y p un puntero al mismo tipo de datos
  - En p se asignó la dirección de inicio de v
  - Y n es una expresión que da un valor entero
- Entonces se da la equivalencia

```
v[n] ≡ *(p+(n))
```

- Por eso, si hago p++ suma “1 desplazamiento” a p de modo de apuntar al siguiente elemento del tipo al que apunta. En otras palabras suma el sizeof del tipo al que apunta

Punteros y arreglos

```c
int main()
{
        int v[5];
        int *p;

        p = v; /* Notar que no hice &v
                  Hubiese sido equivalente poner &v[0] */
        v[0] = 5;
        printf("v[0] vale %d\n", *p);
        /* imprime: v[0] vale 5 */

        *(p+1) = 7;
        printf("v[1] vale %d\n", v[1]);
        /* imprime: v[1] vale 7 */
}
```

Ejemplo

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int v[5] = { 2, 4, 6, 8, 10};
        int *p;
        int i;

        p = v;
        for (i = 0; i < 5; i ++)
                printf("v[%d] = %d\n", i, v[i]);

        printf("\n----------\n\n");
        for (i = 0; i < 5; i ++)
                printf("*(p+%d) = %d\n", i, *(p+i));

        printf("\n----------\n\n");
        for (i = 0; i < 5; i ++)
                printf("p desplazado %d = %d\n", i, *p++);
        /* OJO p ya no apunta al inicio de v */
}
```

Salida

```
v[0] = 2
v[1] = 4
v[2] = 6
v[3] = 8
v[4] = 10

----------

*(p+0) = 2
*(p+1) = 4
*(p+2) = 6
*(p+3) = 8
*(p+4) = 10

----------

p desplazado 0 = 2
p desplazado 1 = 4
p desplazado 2 = 6
p desplazado 3 = 8
p desplazado 4 = 10
```

Arreglos dinámicos

- El estándar ANSI C requiere que se indique el tamaño del vector con una constante, solo a partir del estándar C99 se permite usar una variable (cuyo valor solo se conocerá en tiempo de ejecución) para dimensionar un vector.
- La restricción es que el vector sea automático, es decir, declarado adentro de una función y sin anteponer static.
- Ejemplo: supongamos que ya teníamos una variable n cargada mediante un scanf y queremos declarar un vector de double del tamaño que indique n

```
double vp[n]; // válido solamente si el compilador implementa C99 o C11
```

- Sigue siendo más seguro pedir memoria con malloc y usar el operador [ ] con el puntero devuelto

Licencia