Punteros

- Una variable de tipo puntero almacena una dirección de memoria.
- Dicha dirección de memoria puede ser la que almacena un objeto (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 un * entre el nombre de la variable y el tipo al que apunta

```
long l = 375; /* variable tipo long a la que
                 asignamos el valor 375 */
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 unario de indirección *

```
k = *pl; /* ahora k vale 375 ya que *pl (lo apuntado
            por pl) equivale a l, es decir, esta última
            línea equivale a k = l */
```

- Casting: Al hacer casting entre tipos de punteros, el mismo no cambia, es decir la dirección almacenada permanece inalterada. Lo que cambia es el modo de interpretar aquello a lo que apunta

```
float f = 12.345;
float *p = &f;
printf("float* = %g\tint* = %d\n", *p, *(int *)p);
//float* = 12.345 int* = 1095075103
```

Punteros en lugar de referencias

- En lenguaje C no hay pasaje de parámetros por referencia. En los casos que se quiere modificar el argumento actual se lo hace a través de un puntero

C++ por referencia

```
intercambio(i, j);

void
intercambio(int& i, int& j)
{
        int aux;

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

C mediante punteros

```
intercambio(&i, &j);

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

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

sizeof

- El operador `sizeof` toma como operando o bien un tipo de datos (completo), encerrado entre paréntesis, o bien una expresión, la que puede o no estar entre paréntesis.
- El resultado es el tamaño en bytes que ocupa el tipo de datos del operando. El tipo de entero que devuelve es `size_t`
- El operador `sizeof` se calcula en tiempo de compilación
  - Salvo para arreglos de largo variable

Punteros Genéricos

- Lo que en C++ se logra con sobrecarga y/o templates en C se suele resolver con punteros “genéricos”, es decir con punteros a void.
- Guardan una dirección de memoria pero no pueden interpretar lo que hay en dicha dirección, es necesario hacer un casting para desrreferenciar lo apuntado
- No es necesario hacer casting para asignar un puntero con tipo a uno void o al revés.
  - En C++ para asignar un puntero void a uno con tipo **debo** hacer casting.

```
float f = 12.345, *pf1 = &f;
float *pf2 = pf1; //mismo tipo, no hace falta casting
int *pi = (int *)pf1; //distinto tipo, necesito casting
void *pv = pf1; //a void no hace falta casting
int c = *pi; //c valdrá, como vimos, 1095075103
c = *(int *)pv; //el casting es necesario para desrreferenciar
```

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 );
```

- 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));
```

- 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

Aritmética de punteros

- Dado un puntero a un tipo completo (lo que elimina punteros a `void`) y un tipo entero puede hacer la suma o resta.
- La semántica es, si sumo un entero n, hacer que el puntero apunte n elementos adelante o atrás según sume o reste.
- Para ello al valor del puntero se le suma o resta `n * sizeof(tipo-dato)`
- Por ejemplo si tengo un puntero double que guarda la dirección 1000 y le sumo al puntero un 3, apuntará a:

```
  1000 + (3 * sizeof(double))
= 1000 + (3 * 8) = 1024
```

Arreglos (repaso)

- Los arreglos en C son similares a los de C++

```
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]);
```

- Los arreglos deben dimensionarse de la menos un elemento para ser un tipo completo

Arreglos

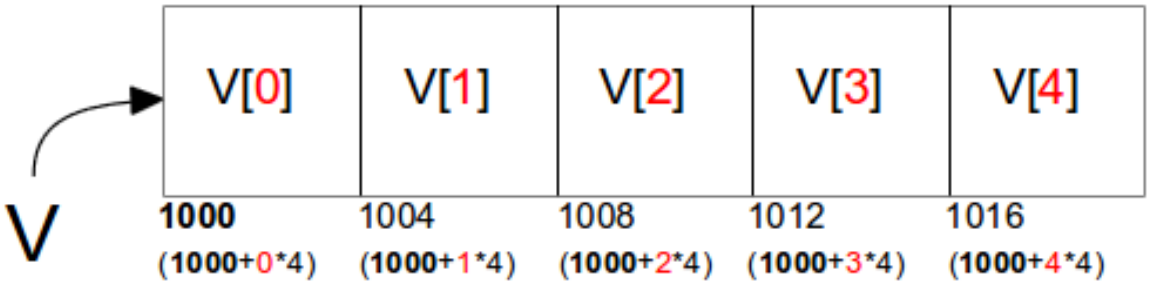
- Un vector también puede iniciarlizarse al momento de declararlo

```
int v[5] = {1, 6, 8, 3, 9};
int w[] = {3, 12, 7};
int x[10] = {0}; //en C debo poner al menos un 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, cuando se lo usa en una expresión “decae” a 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.
- Sus operandos son un arreglo o un puntero y un entero

Arreglos y sizeof

- Donde defino un arreglo el sizeof me da el tamaño en bytes reservado para el arreglo.
- Si lo paso como parámetro de un función “decae” un puntero, por lo tanto su sizeof dará el tamaño de puntero.
  - Por eso pasamos la dimensión como un parámetro adicional.
- Como parámetro de un función un arreglo incompleto y un puntero son equivalentes, si bien declarar un arreglo da un código más claro

```
void f1(int vec[], ...); //lo aconsejado
void f1(int *vec, ...); // equivalente pero menos claro
```

- En ambos casos puedo usar el operador `[]`

Operador [ ]

- 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 (usando aritmética de punteros)

```
v[n] ≡ *(p+(n))
```

- Como el + es comuntativo es válido escribir (aunque desaconsejado por claridad)

```
v[i] ≡ i[v]        v[5] ≡ 5[v]
```

- Los operadores & y * se cancelan entre ellos por lo tanto

```
&*p ≡ p
&v[i] ≡ &*(v+i) ≡ v+i
```

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

Ejemplo

```c
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
```

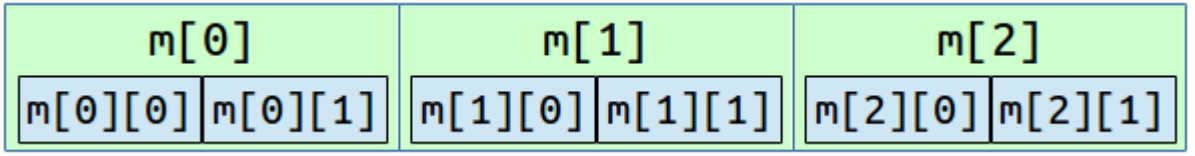
Aritmética de punteros Diferencia

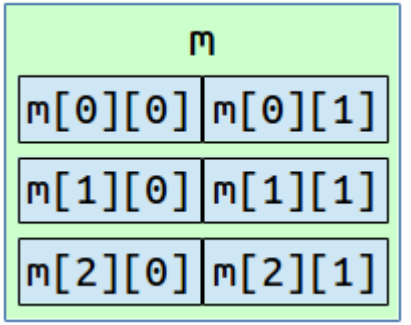
- Puedo restar dos punteros del mismo tipo (completo). El resultado es la diferencia medida en elementos, no bytes. El tipo de datos es un entero de tipo `ptrdiff_t`.
- Esto permite calcular el índice en un arreglo a partir de un puntero:

```
int comp_f(const void *i, const void *j);
int *pelem;
int clave = 15;
int vec[dim] = {2, 4, 9, 15, 22, 37};
pelem = bsearch(&clave, vec, dim, sizeof vec[0], comp_f);
if(pelem) {
    int i = pelem - vec; //vec sin [] es un int*
    printf("%d está en la posición %d\n", vec[i], i);
} else {
    printf("%d no está entre los datos\n", clave);
}
```

Matrices

- No hay matrices como tales, solo vector de vector. Por ejemplo si declaramos
  - `double mat[3][2];`
- Entonces `mat` es un vector de 3 elementos, donde cada uno de ellos es a su vez un vector de 2 doubles.
- Como cada elemento es un vector de 2 doubles implica que en memoria se guardará la “matriz” por filas (cada fila es un elemento del “vector” `mat`)





Matrices como parámetros

- Si tengo un **vector de tipo T** al pasarlo a una función decae a **puntero a tipo T**. Por tanto si paso mat a una función decae a: puntero a vector de 2 doubles: `(double(*)[2])`
  - Nota: decae solo el primer vector, no es recursivo por eso se mantiene el segundo y no cambia a `double **`
- Una función con parámetro: `T param[]` se decae a **puntero a T** y con eso le basta para saber como pasar de un elemento al siguiente (sizeof **T**).
- Con una matriz puedo hacer `T param[][DIM]` con `DIM` una expresión entera conocida en tiempo de compilación, ya que es un vector de vector. Esto le basta para saber como pasar al elemento siguiente.
- En resumen, solo la dimensión "más interna" puede no especificarse, el resto debe ser conocida en tiempo de compilación

Arreglos dinámicos

- El estándar ANSI C y C++ requieren que se indique el tamaño del vector con una constante. El estándar C99 agregó los VLA (variable length array) que pueden establecer su dimensión en tiempo de ejecución.
- La restricción es que el vector sea automático, es decir, que será alojado en el stack, lo que lo hace peligroso (stack overflow) si la dimensión es grande. Por eso es una característica deprecada.
- Sigue siendo más seguro pedir memoria con malloc y usar el operador [ ] con el puntero devuelto.

Arreglos dinámicos

- Los VLA permite usar un “modo dinámico” en las funciones para hacerlas genéricas con respecto a las dimensiones

```
void mostrar(int n, int m, double a[n][m]);
void invertir(int n, double a[n][n]);
void mult(int n, int m, int k
          , float a[n][k], float a[k][m]
          , float r[n][m]);
```

- Estas características son optativas y están en revisión

Licencia