**Cosas sobre C**

* Incluir siempre al principio #include <stdio.h> para que el compilador utilice la librería estándar de entrada/salida de C.
* OJO! \\ sirve para escribir barra simple y que no te pille el inicio de un \t, \n… Carácter nulo \0, por el que termina internamente cualquier cadena.
* Escribir programa 🡪 guardar 🡪 compilar con gcc 🡪 ejecutar el .exe.
* Declarar las variables con sus tipos (no es de tipado dinámico; una variable entera siempre será entera). Inicializarlas es buena práctica (aunque sea a 0).
* Si pones un número entre comillas te da su código ASCII.
* La definición de funciones tiene como estructura función(parámetros) {proposiciones}.
* Para leer un carácter con getchar() 🡪 int c = getchar().
* Si una variable es declarada entera y multiplicas su valor por un punto flotante, C lo trunca. Por ejemplo, calcular una variable entera multiplicando por 5/7 sería lo mismo que multiplicarla por 0 porque 5/7 se trunca. Para solucionarlo, se multiplica primero por 5 y se divide por 7. Si la variable a calcular fuera float, no tendríamos problema: se podría usar 5.0/7.0 para indicar que es división de puntos flotantes.
* Si se resta un entero a un flotante, el entero se convertirá a flotante. BUENA PRÁCTICA: poner el punto ficticio para aclarar qué es flotante y qué es entero.
* Cadenas formateadas:
  + printf(“%d”, variable) 🡪 %d hace referencia a una variable entera.
  + printf(“%nd”, variable) 🡪 n es un número que indica los campos que puede ocupar el entero de la variable en anchura.
  + %f hace referencia a un punto flotante y %g hace referencia a double.
  + Si haces %.0f/g 🡪 suprimir punto decimal y parte fraccional de float/double.
  + A las especificaciones de anchura se le puede añadir .num. Esto indica la cantidad de decimales que queremos en la impresión (%3.0f imprimirá en 3 caracteres de ancho como mínimo el decimal; sin números decimales. Si ponemos 1 en vez de 0, imprimirá un decimal).
  + OJO! %s es para imprimir arrays de caracteres; %c es para caracteres.
  + Poner %lf hace referencia a **DOUBLE** (long float); y %ld es **long int**
  + %hd se refiere a un SHORT INT; y %lu es long UNSIGNED int.
  + %p se usa para una dirección de memoria de un puntero (imprimirla).
* Constantes simbólicas:
  + No son variables 🡪 constantes guardadas con un nombre asignado. Se mete su valor en una dirección de memoria para usarlo cuando se requiera.
  + Se definen antes del main, no llevan ;. # define variable valor
* Entrada y salida de teclado (input en Python):
  + Para leer por teclado, te defines una variable de algún tipo (por ejemplo int si queremos que se reserve suficiente espacio para la variable como para almacenar el carácter EOF 🡪 char no es tan grande porque son 1 byte normalmente).
  + Lectura de teclado 🡪 variable = getchar()
  + Devolver estas variables por pantalla 🡪 putchar(variable)
  + EOF = end of file (Ctrl+z)
* Se permiten los operadores variable+=algo (igual que Python); y variable++ ó ++variable para aumentar en 1 el valor de la variable. Idem para restar.
* Estructura if … else if … else.
* Importante que para operadores AND (&&) y OR(||), especialmente para OR, las condiciones se evalúan de izquierda a la derecha. Si en un OR la primera ya se cumple, el OR ya es True sin necesidad de evaluar las otras.
* for (inicialización de la variable de control; condición; aumento de la variable)

**Arrays**

* Inicializar en declaraciones con int array[tamaño]. El tipo dependerá del tipo de elementos que contenga 🡪 **¿qué ocurre si guarda elementos de distinto tipo? 🡪 void**
* Luego hay que inicializar cada posición (por ejemplo darle a todas valor 0 🡪 for i=0, i<10, i++ 🡪 array[i] = 0).
* Inicializar array de caracteres (cadena en Python) 🡪 char array[tamaño]. Se permite hacer referencia a array[-1].
* Si queremos definir uno manualmente, declaramos la variable del array:
  + char array[len] = {‘a’, ‘b’, …}

Funciones:

* Se definen fuera del main de forma:
  + int función(parámetros){…}
  + void función(parámetros){…} 🡪 Función que no devuelve nada (procedimiento).
* Dentro tienen su propia parte de declaración de variables locales (argumentos).
* Se definirán según el tipo de dato que devuelva la función.
* Se pueden cerrar con un return vacío para devolver el flujo de ejecución al programa principal.
* Main es la función principal del programa. Se suele definir como int y se hace que devuelva 0 si la ejecución es correcta.
* Pueden declararse funciones prototipo antes del main, que son funciones vacías que reciben los mismos parámetros y tienen el mismo nombre que la que se definirá más adelante.
* Cualquier cosa que se le haga a una variable global dentro de una función no tendrá efecto sobre el programa principal. Los arrays sí son modificados.

Puede hacerse casting 🡪 poner entre paréntesis nuevo tipo para una variable.

Función scanf:

* Para pedir una entrada por teclado.
* Se define una variable del tipo que queramos leer, y se llama a:
  + Scanf(“%tipo”, &variable) 🡪 puntero a la variable.

Variables externas:

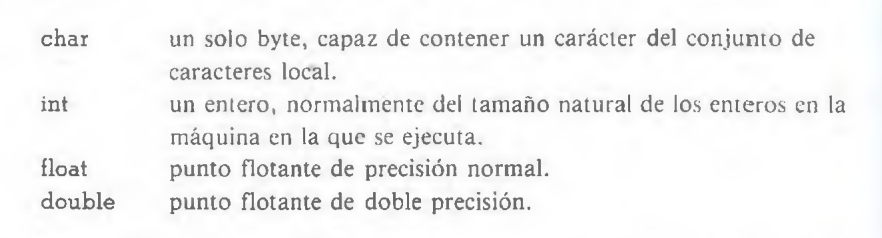
* Se definen fuera de cualquier función. Todas tienen acceso a ellas; y se mantienen permanentemente en memoria mientras el programa se ejecuta.
* Hay que DECLARARLAS dentro de todas las funciones donde vayan a usarse (extern tipo variable) si se usan antes de definirlas o están definidas en otro fichero.
* Pueden usarse para funciones en vez de pasarle argumentos:
  + Defines la función como tipo función(void)
  + Dentro de la función, declaras las variables externas con extern.
* OJO! Si las variables externas se definen al principio del programa, no hace falta declararlas con extern en las funciones, sino que no se declaran en la función. Esta es la mejor práctica cuando usamos solo un archivo como programa.
* Usar variables externas para funciones hace que perdamos generalidad 🡪 hacemos referencia a variables que tienen que existir en el programa (no pueden usarse para otros programas sin esas variables).

OJO! Las comillas dobles se usan para imprimir texto; las simples se usan para comparar variables con texto (no sé la explicación teórica detrás de esto).

BUENA PRACTICA 🡪 Puedes definir como constantes simbólicas las variables que describen estados (1 si pasa algo, 0 si no).

OJO! En los ejercicios en los que construyes una nueva cadena/texto a partir de otro, añadiendo o eliminando cosas, tienes que definir un índice para recorrer el texto inicial y otro para el final. Los índices no coinciden porque la posición j de la entrada no tiene por qué ser la siguiente de la salida.

**Tema 2 – variables, tipos de datos, operadores**



* Enteros int 🡪 short (int) como mínimo de 16 bits <= int <= long (int) como mínimo 32 bits.
* Calificador signed/unsigned 🡪 ­aplicable a char (es al final un entero pequeño) o cualquier int. Unsigned para enteros no negativos.
* Long double 🡪 punto flotante de precisión extendida.
* El valor de una asignación (=) es simplemente lo que se asigna. Precedencia de derecha a izquierda.

Constantes de carácter:

* Se ponen entre ‘APÓSTROFOS’.
* Se suelen usar para comparar (if (var == ‘letra’) …).
* Si se usan sobre ellos operadores como < o que requieran datos numéricos, se usan sus códigos ASCII.
  + Ejemplo: Si s es un array de caracteres que incluye char entre ‘0’ y ‘9’, entonces s[i]-‘0’ devuelve el resultado numérico de s[i].
  + Para transformar una letra codificada como int c mayúscula en minúscula, hay que hacer 🡪 **c + ‘a’ – ‘A’**. La lógica detrás de esto es que los códigos ASCII de las minúsculas son 97-122; y los de las mayúsculas son 65-90. La diferencia entre cualquier letra minúscula y su mayúscula es 32. Como si c es mayúscula, su ASCII es 32 unidades menor que el de su minúscula, le sumamos 32, expresado para saber de dónde viene como (‘a’ – ‘A’).

Secuencias de escape:

* Aunque sean dos caracteres representan solo uno.
* A parte de \n, \t, \0 (carácter vacío para fin de cadena):
  + \a (carácter de alarma o campana)
  + \b (retroceso)
  + \v (tabulador vertical) 🡪 vtab
  + \\ (diagonal \), \?, \’, \”
  + \(uno, dos o tres dígitos octales)
  + \x(uno o dos dígitos hexadecimales)

OJO! Una constante de cadena es un array de caracteres. Su representación interna ocupa en memoria tantos bytes como caracteres tenga, más uno para incluir el carácter nulo \0.

* Función strlen(cadena) de la biblioteca estándar (header **string.h**) es el equivalente al len de Python (no cuenta el carácter nulo).
* ¡¡¡MUCHO OJO!!! ‘x’ es un carácter (literalmente codificado como su número correspondiente en ASCII). “x” es un array de caracteres codificado carácter a carácter, con el carácter nulo para indicar el final.
* Para definir un array de caracteres con un valor dado, se puede definir como constante simbólica (# define FRASE “…”). Se puede definir también un array manualmente (char s[] = “…”).

Constantes de enumeración:

* Se definen con enum nombre { … }.
* En los corchetes meteremos nombres de constantes, y sus respectivos valores. Por ejemplo:
  + enum escapes { BELL = ‘\a’, RETROCESO = ‘\b’, … }
* Si se ponen solo los nombres, la primera constante tomará valor 0, la segunda 1…
* Si se pone que la primera tome un valor numérico, no hará falta especificar las siguientes ya que se continuará asignando los siguientes valores enteros.
* Podemos definir variables enum nombre id = (una de las constantes definidas).
* Podemos meter este tipo de constantes como parámetros de una función (enum nombre id).
* Podemos hacer typedef enum day day para encapsular el tipo de dato constante bajo el nombre “day”.
* Para pasar de una variable de enumeración a su nombre, se puede hacer con un switch-case o definiendo un array en el mismo orden e indexándolo.

Declaraciones:

* Notar que se pueden inicializar variables al declararlas: int i = 0;
* Las variables estáticas y externas se inicializan en 0 por omisión. Las automáticas contienen un valor indefinido si no se inicializan (basura). Estas son las comunes. Las estáticas se definen con static, y las externas son las de fuera del programa.
* Pueden declararse variables con el prefijo “const” que no cambiarán durante la ejecución (const int, const char…). También aplicable a arrays y a argumentos de funciones (la función no cambia la variable argumento). Si se trata de cambiar un const, el programa no compilará.

Operadores lógicos 🡪 && (AND lógico) y || (OR lógico):

* Se evalúan de izquierda a derecha:
  + En AND, en cuanto hay una condición que no se cumple, deja de evaluarse la condición y se concluye False. Se consigue más eficiencia poniendo primero las condiciones más improbables. Conviene también poner primero las condiciones menos costosas de comprobar.
  + En OR, en cuanto una de las condiciones se cumple, se deja de evaluar y se concluye True. Más eficiente colocar primero las condiciones más probables.
* Precedencia 🡪 asignación (=) < || < && < igualdad < desigualdad < aritméticos < ++

IMPORTANTE: La biblioteca ctype.h incluye funciones para manejar tipos de datos, como tolower() para regresar el valor en minúscula de c si es mayúscula; o isdigit(), que regresa 0 si es falso o cualquier otro valor si es verdadero. Esto es distinto a los operadores de comparación, que regresan 1 si es verdadero.

Conversión de tipos en operaciones aritméticas: cuando hay dos tipos distintos, se convierte el “menor” al “mayor”, devolviéndose el resultado en el tipo mayor 🡪 long double > double > float > int > char y short. Además, si alguno es long, el resto se convierten a long. Se complican si hay unsigned.

Las funciones de la biblioteca math.h utilizan doble precisión por defecto. Si aplicamos funciones como sqrt, los argumentos que le pasemos tienen que ser double. Si ponemos directamente un número (como por ejemplo, sqrt(2)), la propia llamada obliga al entero 2 a ser el valor double 2.0.

En argumentos de funciones, char y short pasan a ser int; y float pasa a ser double. Por eso declaramos argumentos a funciones como int aunque le pasemos un char por ejemplo; ya que la conversión se encarga de que funcione.

**Casting:**

* Se trata de forzar la conversión de una variable definida de un tipo a otro.
* Se usa la variable para algo, poniendo delante de ella entre paréntesis el nuevo tipo. Por ejemplo: sqrt((double) n).

Operadores de incremento:

* Los operadores ++ y -- pueden usarse como operadores prefijos y postfijos, pero son diferentes.
* Si n = 5:
* Prefijos 🡪 incrementa el valor antes de usarse: x = ++n 🡪 x toma valor 6.
* Sufijos 🡪 incrementa el valor después de usarse: x = n++ 🡪 x toma valor 5.
* Si no se va a asignar a una variable, ambos usos son equivalentes.

Ejemplo: La sentencia s[j++] = ‘algo’; es igual a asignar el carácter a la posición j y luego incrementarla.

En la biblioteca estándar (string.h) tenemos las funciones **strcat** para concatenar, **strlen** para longitud, **strbrk** para regresar un puntero a la primera posición de una cadena que esté contenida en otra. En ctype.h tenemos funciones como **isdigit**(), **isspace()** o **tolower**(). Notar que isspace detecta espacios en blanco, tabulaciones, fin de línea…

MANEJO DE BITS: Si tienes un número en una variable x y haces x&1 🡪 regresa 1 si el bit menos significativo de x es 1; 0 si no. La operación x &= (x-1) elimina el primer bit menos significativo (de más a la derecha) que sea 1.

**Sentencias if-else:**

z = (a > b) ? a : b; 🡪 if (a > b), z = a ; else z = b.

Estructura 🡪 condición ? arg1 : arg2.

Si las condiciones hacen que una variable tome valores de distinto tipo según se dé una condición u otra, la variable se convertirá al “mayor” tipo tome el valor que tome (por ejemplo, si puede tomar un int o un float, la variable será float pase lo que pase).



Dependiendo del compilador, n se incrementará antes de realizar la llamada de la función o después. Ante la incertidumbre, es mejor incrementar la variable fuera del print.

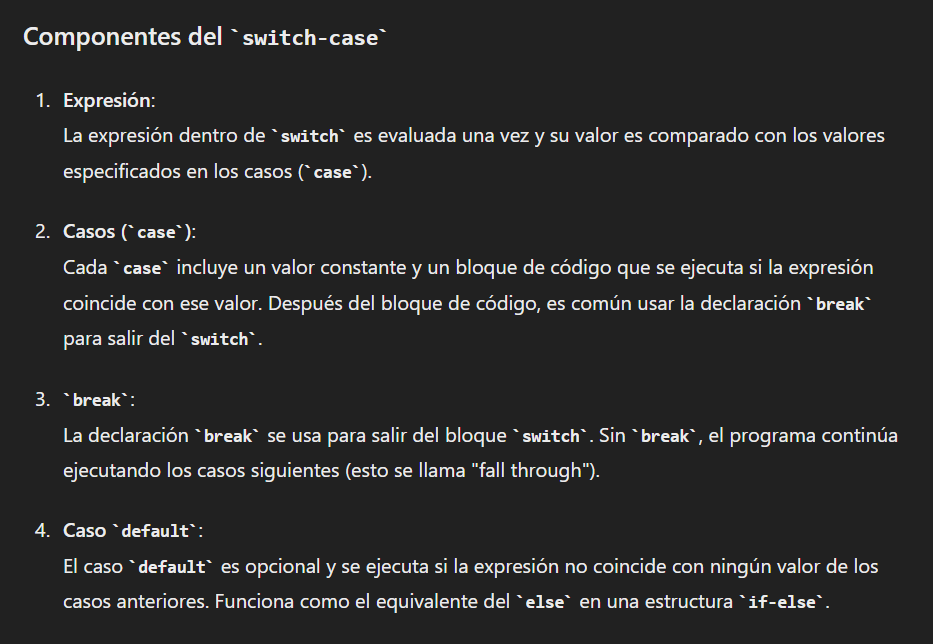
OJO! El operador sizeof (biblioteca estándar) es muy útil como sustituto del “len” en arrays, ya que devuelve el tamaño en bytes de un elemento. Puede usarse para calcular el tamaño de variables o identificadores; y de tipos (sizeof(char)) 🡪 **len = sizeof(array) – sizeof(array[0])**

**Tema 3 – estructuras y sentencias**

**Switch-case**

La estructura es switch(**entero**){ case 1: case 2: … case n: “pasa algo”; case n+1: … case n+m: “pasa otra cosa”; default: “pasa otra cosa”; }

Se trata de una especie de if-else anidado, preferible sobre este cuando hay muchas condiciones que pueden darse. Los case son valores que toma el elemento “cosa” dentro del switch. Para salir del switch, se debe usar un break o un return. El “default” es el equivalente al “else” y no es obligatorio (representa un caso cualquiera no evaluado en los case).



Un bucle “for” está formado por tres expresiones (inicialización ; comprobación; incremento). Puede omitirse la inicialización o la acción, pero deben permanecer los punto y coma. Si se omite la comprobación, se repetirá infinitamente hasta llegar a un break o un return dentro del bucle.

While vs for 🡪 normalmente, si no hay inicialización, while es más natural. El for se prefiere cuando hay una inicialización y un incremento claros que quieren tenerse controlados.

**Operador coma “,”:**

Se usa sobre todo en proposiciones “for” para colocar múltiples expresiones en una misma “parte” del bucle. Las expresiones se evalúan de izquierda a derecha. Permite, por ejemplo, operar con dos índices a la vez. No recomendable usarlo. También se puede usar para instrucciones secuenciales (en vez de poner una detrás de otra, las pones en la misma línea con comas). Menor legibilidad del código.

**Ciclos do-while:**

* Primero se hace la acción (do), y luego se comprueba la condición de término.
* Do (proposición) while (expresión)

En cualquier tipo de ciclos, puede usarse el break para salir de él si se da alguna situación, independientemente de la condición de parada; o continue para pasar a la siguiente iteración (si sucede algo, ignorarlo y pasar al siguiente elemento; lo que permite mediante una sola comprobación saber si se tiene que ejecutar el código posterior dentro del bucle en esa iteración, o no).

**Tema 4 – funciones**

* Las funciones pueden definirse de forma externa con extern tipo function, donde la función function puede estar escrita en otro fichero.
* Las variables externas recordamos que se definen fuera de cualquier función, y se declaran de manera opcional dentro de las funciones con extern.
* Si se omite el tipo que regresa una función, se asume que es un int.
* Si tenemos varias funciones en el fichero que deben acceder o modificar una misma variable, conviene que dicha variable sea EXTERNA. Por ejemplo, si hay una variable que almacena una cadena que viene construida por la función getline, y recorrida por otra función getop, la variable cadena será externa.
* Si se pasa una variable como parámetro a una función, aunque la función la cambie internamente, esta no cambia en el programa principal (**call by value**). Si queremos modificarla, le pasamos un PUNTERO a la variable (**call by reference**).
* La función swap solo modifica las variables si se define como **swap(int \*i, int \*j)**.
* Si una función debe recibir un int y se le pasa float, se convierte de float a int automáticamente (truncando).

Podemos escribir un programa en distintos ficheros y compilarlos a la vez así:

* gcc fic1.c fic2.c fic3.c (-o nombre)
* El compilador genera tres ficheros objeto y los carga dentro de un ejecutable llamado a.out (o el nombre especificado).
* Si un mismo programa está en varios ficheros, hay que hacer un archivo.h (header) que reúna todas las definiciones y declaraciones **compartidas** entre archivos. Se pondrá un: # include “archivo.h” en cada fichero que conforme el programa.
* ¡MUCHO OJO! Si tenemos variables externas que queremos que se compartan entre varios ficheros, las definimos normalmente en uno (fuera de toda función), y luego las declaramos en los demás con extern. En el fichero de definición no hace falta declararlas.

**Declaración vs definición**

* Una variable se tiene que definir una vez indicando su tipo, longitud en caso de arrays… Se reserva espacio para ella en memoria. También puede inicializarse en la definición.
* La declaración se hace en teoría tantas veces como sitios en los que se vaya a usar una variable. Si es externa y se define antes de las funciones donde se usa, no hace falta declararla, sino que se podrá usar directamente. Si se usa antes de definirla, o se define en otro fichero, hay que declararla con extern (el tamaño de los arrays en las declaraciones no es necesario ponerlo).
* Las funciones tienen que declararse primero en un .h o al principio del programa; y luego se definen programándolas. En la declaración de funciones no es necesario poner nombres de variables, solo los tipos.

**Variables estáticas:**

* Si dentro de uno de los múltiples archivos fuente definimos variables como “static”, entonces serán externas para las funciones de ese fichero; pero invisibles para el resto de ficheros. Impide conflictos con otras funciones del programa.
* Se aplica lo mismo a declaración de funciones.
* Como variables internas en funciones, las variables definidas como “static” siguen siendo solo visibles para la función (igual que las automáticas), pero **se mantienen durante toda la ejecución del programa** (no se borran cuando acaba la llamada a la función). Se inicializan solo una vez en la primera llamada.
  + EJEMPLO: Si defines un índice como static, la función puede devolver algo y recordar por dónde iba para la siguiente llamada (útil para separar una línea en palabras o para concatenar cadenas).
* Variables temporales 🡪 \_variable

**Estructura por bloques y alcance de variables:**

* Al igual que al definir y declarar variables dentro de funciones, su alcance es en dicha función, lo mismo se aplica para estructuras como “if”, “for”… Si dentro de sus corchetes definimos una variable, esta existirá solo dentro de la estructura.
* Un array puede inicializarse al definirse ({}) y no necesitará el tamaño. El tamaño de un array de caracteres inicializado a “hola” es 5 (número de caracteres + finalizador \0).

**Preprocesado:**

* Para los include del principio de un fichero, “” se usan para buscar el fichero que quiere incluirse en la ruta actual; y <> se usa para buscarlo en una serie de rutas definidas por el sistema.
* Los define sirven para sustituir macros. No tienen por qué ser únicamente constantes simbólicas; sino que puede condensarse código, como en los siguientes ejemplos:
  + **# define swap(t, x, y) t \_temp = x; x = y; y = \_temp; 🡪** t es el tipo, necesario para definir la variable temporal para el intercambio.
  + **# define max (A, B) ((A) > (B) ? (A) : (B) 🡪** Cada vez que se ponga max entre dos ocurrencias, se sustituirá por el código de comparación para devolver el resultado deseado.
  + **# undef** sirve para eliminar una macro predefinida.
  + **## sirve para concatenar DENTRO de la definición de un macro.**

**Tema 5 – Punteros**

* Operador ‘&’ devuelve la dirección de memoria de un objeto. Si declaramos una variable p como un puntero a un entero (u otro tipo) 🡪 int \*p; entonces para inicializarla como puntero a un objeto c se haría: p = &c. Sólo aplicable a objetos en memoria (no constantes ni expresiones).
* OJO!!! Una función solo podrá modificar una variable que se le pasa como argumento si se le pasa como puntero (si se le pasa el valor de la variable en sí, modifica una copia).
* En el código, cuando le asignamos un valor a p, estamos cambiando la variable a la que apunta p.
* Cuando asignamos un valor a \*p, estamos cambiando el valor de la variable a la que apunta p.
* Por tanto: p = dirección de una variable; \*p = valor de dicha variable.
* Si una función toma como argumento un puntero, se pone como 🡪 func(char \*) 🡪 puntero a un carácter.
* OJO: Para aumentar en uno el valor de la variable a la que apunta p, se hace con \*p+=1; ++\*p ó (\*p)++ (si hiciéramos \*p++ estaríamos incrementando p y luego desreferenciando).

**Funciones con punteros:**

* Si hacemos una función swap(int x, int y); y la llamamos con dos variables (swap(a, b)), no se intercambian los valores de las variables a y b, ya que la función afecta a copias de las mismas.
* Para cambiar los valores directamente, hay que usar punteros y cambiar las referencias de estos (swap(int \*px, int \*py)).
* La llamada a esta función se haría pasándole como argumentos las direcciones de memoria 🡪 swap(&a, &b)
* IMPORTANTE! Si queremos que una función modifique cualquier variable, tenemos que pasársela tratándola como un PUNTERO.

**Punteros y arrays:**

* Cualquier operación que pueda lograrse indexando un array puede hacerse por punteros también (puntero a la posición i del array).
* Si tenemos un array de tamaño n, y definimos el puntero \*pa como: pa = &a[0]; entonces \*pa apunta ahora a la posición 0 del array.
* Igualmente, si hacemos \*(pa + i), con i < n, este puntero apuntará a la posición i. Correspondencia entre indexación y aritmética de punteros.
* IMPORTANTE: Por definición, una variable del tipo array es en realidad la dirección de memoria de su primer elemento, por lo que, tal como hemos definido el puntero “pa” y el array “a” son iguales (apuntan al mismo sitio). Por tanto, podemos asignar un puntero al primer elemento del array con: pa = a (equivalente a pa = &a[0]).
* **Son equivalentes**: a[i] == \*(pa+i) == \*(a+i) == pa[i].

Recordemos que el array “a” apunta a la dirección de memoria de su primer elemento. La diferencia entre un puntero y el nombre de un array es que el puntero es una variable (admite asignación y operador ++).

* Cuando una función recibe como argumento el nombre de un array, en realidad recibe la localidad de su elemento inicial (puntero).
* Se puede pasar a una función parte de un array, pasándole como argumento el puntero a la posición i 🡪 (array + i) ó (&array[i]).
* Al definir una función, es equivalente 🡪 (type pt[]) == (type \*pt)

**Aritmética de punteros:**

* Un puntero p a una posición de un array “a” admite operaciones de comparación como:
  + p >= a 🡪 p apunta a un elemento que está más adelante en memoria que el primero.
  + p < (tamaño a) 🡪 p apunta a un elemento dentro del espacio reservado para el array.
* Si p y q punteros 🡪 p < q será verdadero si p apunta a un elemento que está antes en el array que al que apunta q.
* Si n es un entero, independientemente del tipo de dato al que apunte p, p+n apuntará al enésimo dato después de p. Se adapta la escala de n al tipo de dato.
* Si p y q punteros y p < q 🡪 q-p+1 es el número de elementos entre p y q, incluyendo q.
* Cuando le sumas 1 a un puntero, se le sumarán tantas posiciones de memoria como ocupe una unidad de dato del tipo al que apunte p. Por ejemplo, si p apunta a int y hacemos p+1, se le suman 4 bytes a p.
* Las operaciones legales son: asignación de apuntadores del mismo tipo, suma y resta de enteros; y resta y comparación con otro puntero a otro elemento del mismo array.

**Punteros y cadenas de caracteres:**

* Al inicializar una cadena como array 🡪 char s[] = “hola”. Es un array (puntero a la primera dirección de memoria). Puede modificar letras de la cadena.
* Si tienes un array[5] y lo inicializas con = {0}, todas las posiciones son 0.
* Al inicializarla como puntero 🡪 char \*ps = “hola”. Es un puntero inicializado para apuntar a una cadena constante. No se puede modificar, pero sí puede hacerse que el puntero apunte a otro sitio.
* OJO!! char \*s indica que s es una cadena (array de caracteres, puntero a primera posición); char \*s[] indica que s es un array de punteros a char.

**Arrays multidimensionales:**

* Se definen poniendo array[i][j] 🡪 array [filas] [columnas].
* Pueden inicializarse poniendo entre corchetes los subarrays (filas) y rellenando los arrays 🡪
  + static int daytab[2] [13] = {

{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},

{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} };

* Para pasarle a una función un array multidimensional como argumento, hay que especificar el número de columnas obligatoriamente. El de filas se puede omitir si es irrelevante 🡪 f(int daytab [] [13]) == f(int (\*daytab) [13])
* Es distinto definir un array bidimensional (array [i] [j]) y un array de punteros (\*array[i]); ya que el primero reserva i\*j posiciones de un mismo tipo y longitud, mientras que el segundo reserva i punteros sin inicializar, pudiendo apuntar cada uno a un elemento de distinta longitud.
* El uso más frecuente de arrays de punteros es el de almacenar cadenas.

**Pasarle argumentos a main:**

* Main puede recibir argumentos que deben ser introducidos al llamar al programa desde el terminal (./program.exe arg1 arg2).
* Cuando esto ocurre, main tiene como parámetros:
  + int argc (argument count) 🡪 número de argumentos en la línea de órdenes. Se calcula automáticamente (no se pone en la llamada en la terminal).
  + char \*argv[] (argument vector) 🡪 se define como array de punteros a caracteres. Argv[argc] = NULL (indicador de fin de argumentos).
  + Si queremos definir un puntero al primer argumento (teniendo en cuenta que argv[0] apunta al nombre del programa) 🡪 char \*\*p = argv + 1 es un puntero a un puntero a una posición de argv, que a su vez es un puntero a una cadena (puntero a puntero a char).

**MUCHO OJO!!!** Printf toma un puntero para imprimir como cadena (NO HAY QUE DESREFERENCIAR)

OJO CON EL USO DE PUNTEROS Y DESREFERENCIACIÓN. Si \*argv[] es un puntero a un array de punteros a cadenas, entonces:

* (\*++argv)[0] accede al puntero de la segunda palabra de argv, toma la palabra en sí desreferenciando, y se queda con su primera letra.
* \*++argv[0] primero accede al puntero a la primera palabra de argv (argv[0], que al ser un puntero accede por defecto a la primera letra). Luego, aumenta el puntero, pasando a apuntar a la palabra en sí sin la primera letra, y lo desreferencia para acceder a la primera letra de esta “subpalabra”.

En ambos casos, la salida es un carácter; pero uno es primera letra de segunda palabra, y la otra es segunda letra de primera palabra. Notar que, al hacer ++argv la primera vez, argv ya apuntará a la segunda palabra (esta pasa a ser la primera).

El operador [] y () tiene más preferencia que el \*.

**Uso de punteros a funciones:**

* Imaginemos una función de nombre “name”:
  + Type (\*name)(args) 🡪 puntero a una función que regresa type.
  + Type \*name(args) 🡪 función que regresa un puntero a type (los paréntesis tienen más prioridad que \*).
* Una función puede recibir como argumento un puntero a otra función.
* Si queremos que los argumentos sean punteros lo más generales posible (que admitan cualquier tipo) 🡪 ponemos void \*. Lo mismo se aplica a arrays de punteros (void \*\*) array == void \*array[].
* Para llamar a la función apuntada dentro de la función principal, hacemos:
  + (\*name) (args) 🡪 los argumentos sin el nombre, solo el tipo.
* Dadas dos funciones, se puede elegir mediante una sentencia condicional si utilizar una u otra en el puntero:
  + qsort((void \*\*) lineptr, 0, nlines-1, (int (\*) (void\*, void\*)) (numeric ? numcmp : strcmp))
  + Para que funcione, numcmp y strcmp tienen que compartir argumentos (mismo número y tipo), y devolver ambas un int.
  + Ambos void\* sirven para que los argumentos puedan ser punteros a lo que sea, pero en ambas funciones tienen que coincidir de todas formas.

**Creación de alias (en mayúscula para distinguirlos)**

Podemos crear tipos/estructuras de datos con typedef. Uso: typedef struct {parámetros (char nombre[50], int edad…) } nombre\_del\_alias (Persona);

En este ejemplo, al crear una instancia del tipo de dato: Persona people[100] 🡪 Es un array de 100 personas, por lo que cada persona tendrá los atributos nombre y edad, a los que se puede acceder con people[i].nombre y people[i].edad.

Typedef int Entero 🡪 la palabra Entero equivale a int. El alias va al final

**Tema 6 – Estructuras**

Estructura = colección de varias variables de distinto tipo agrupadas bajo un solo nombre. Estas variables se llaman miembros.

Ejemplos:

* **Struct point** con dos enteros (coordenadas x e y).
* **Struct employee** {char \*name; int salary; int age…).

Luego pueden definirse variables que son estructuras previamente declaradas (p.e: struct point pt 🡪 pt es una variable point, con pt.x y pt.y. Puede inicializarse con 🡪 struct point pt = {0, 0}).

Se hace referencia a un miembro de la estructura con: estructura.miembro.

Se pueden anidar estructuras. Por ejemplo, podemos declarar una struct rectangle cuyos miembros sean dos struct point p1 y p2.

**Operaciones con estructuras:**

* Copiarla o asignarla como unidad 🡪 pasar como argumento a funciones.
* Inicialización.
* Tomar la dirección de memoria donde se aloja con el operador &.
* Acceder a miembros con “.”.
* Una función puede devolver una estructura 🡪 struct name function (parameters) {…}.
* Pueden definirse punteros a estructuras y pasarse a funciones como argumentos (eficiente si son estructuras grandes) 🡪 struct name \*var;
  + NOTA: para acceder a miembros de estructuras mediante un puntero, hay que poner paréntesis (primero desreferenciamos el puntero y luego accedemos al miembro).
  + NOTACIÓN ALTERNATIVA: Si pp es un puntero a una estructura, podemos hacer pp -> miembro para acceder a cualquier miembro de la estructura. Este operador tiene precedencia sobre cualquiera.
* Es ilegal que una estructura contenga como miembro una instancia de sí misma; pero es legal si es un puntero a la estructura.

Struct {int len; char \*str;} \*p; 🡪 creamos un puntero a una estructura arbitararia con len y str como miembros. Si hacemos ++p -> len, estamos incrementando el miembro len en 1.

Se puede definir una variable del tipo de la estructura a la vez que se declara la estructura (struct {…} variable) ó puede declararse la estructura y luego definirse la variable (struct name variable). Con typedef struct podemos definir variables del tipo de dicha struct sólo a través del nombre. En ambos casos, la variable puede inicializarse apropiadamente.

Uso de sizeof:

* sizeof(tipo de estructura de datos)
* sizeof un dato (p.e: array[0])

Para la gestión dinámica de structs (controlar el almacenamiento de la estructura en memoria dinámica Y MODIFICARLA, es decir, ampliarla, reducirla, cambiar un elemento, etc.) se debe hacer uso de la función malloc (librería stdlib) 🡪 (casting) malloc(sizeof(struct)). Esto se encapsula dentro de otra función.

Para alojar cadenas en miembros de structs, se usa la función strdup de stdlib (como malloc pero devuelve siempre punteros a cadenas)

**Listas enlazadas**

* Pueden usarse para crear tablas hash (ponemos la palabra que actuará como clave al ser lo que se busca, y el valor que representa, junto con una referencia a la siguiente entrada).
* Para iterar por estas listas, hay que declarar un puntero (struct X \*np) e inicializarlo a la lista. Mientras el puntero no sea NULL, hacemos que el puntero apunte al siguiente nodo (np = np -> next)

**Tema 7 – Entrada/salida**

* Printf(“%.\*s”, max, str) 🡪 imprime max caracteres de la cadena str. La función realmente tiene un return del número de palabras que se imprimen.
* Scanf(“%f%f”, float1, float2) 🡪 pide por teclado dos floats y los guarda en las variables. Se introducen separados por espacios. Esta también tiene un return, que será 1 si todo va bien.
* Podemos redireccionar la entrada de un programa (scanf) con program.exe < file; y la salida (printf) con program.exe > file.
* Puede usarse scanf para lectura de palabras hasta encontrar EOF en vez de getchar o getword (basada en buffer y todo eso). Consistiría en un bucle while ( scanf (“%s”, word) != EOF).
* Si usamos la librería <time.h>, la expresión time(NULL) toma el tiempo actual del sistema en segundos.
* La función srand sirve para fijar una semilla para llamar a la función pseudo-aleatoria “rand”.

**Abrir un fichero y volcarlo a un array:**

* Declarar en la función un puntero a un archivo 🡪 FILE \*file.
* file = fopen(“nombre.ext”, “r”). Si pongo “r+” o “w+”, se refiere a que puedo leer y escribir.
* while (fscanf (file, “%d”, &array[i++]) == 1) 🡪 copiar cada entrada a la posición correspondiente de un array NUMÉRICO. Es como scanf pero más general, y necesita el puntero file.
* Si la lectura falla (no hay más por leer), fscanf devuelve 0.
* La función **rewind**(puntero a fichero) te desplaza al inicio del fichero.
* Getc y putc son equivalentes a getchar y putchar, solo que hay que especificar la entrada y la salida (por el terminal son stdin y stdout).

**Assert:**

* Se usa como un punto de inflexión en el programa al comprobar una expresión. Si esta expresión es falsa, la ejecución se aborta. Útil para el debug.
* Están en la librería <assert.h>.
* También es muy útil para lanzar errores (por ejemplo, pedir una variable por teclado con scanf y hacer que no sea 0).
* La etiqueta NDEBUG (#define NDEBUG) anula los assert. Útil para poner asserts al debuggear y, una vez comprobamos que todo va bien, anularlos.

**MUCHO OJO!!** Para almacenar CADENAS en miembros de STRUCTS, tenemos que usar strcpy (librería string.h).

**Lenguaje C++**

* La librería estándar en este caso es <iostream>. Incluye a la de C.
* Hay que incluir al inicio de todos los programas: using namespace :: std; Esto se hace para poder obviar el indicativo “std” en las llamadas a funciones de la librería estándar.
* Se suele usar const type para definir constantes, en vez de #define (se intenta evitar usar el preprocesador lo máximo posible). Define sirve para sustitución textual.
* En vez de usar #define con funciones básicas (calcular un número aleatorio en un rango), se usan las funciones inline:
  + inline int rand\_side() {return (rand() % sides + 1);}
  + Consume menos recursos que una llamada a una función normal.
* La salida (imprimir por pantalla) se hace con cout << “cadena”.
* La entrada (pedir datos de teclado) se hace con cin >> variable\_para\_almacenar. No hace falta llamar por referencia (dirección de memoria).
* Se concatenan en la salida datos, como:
  + cout << "\nDistance is " << convert(miles) << " km." << endl;
  + Importante endl para terminar la línea y vaciar el buffer cuando se usa una función en la salida.
* Existe el tipo de dato “auto” para variables que se inicializan. El compilador decide el tipo de dato que mejor se ajusta a la variable inicializada. Útil para declaraciones complicadas.
* Cambio más importante: introducción de clases para crear TADs. Se diferencian de los struct en que permiten definir operaciones para trabajar con él sin que sean visibles para el usuario (encapsulación y ocultamiento). Puedes definir fragmentos (atributos y operaciones) public y private. Los atributos son private por convenio.
* 0 es el puntero nulo, equivalente a nullptr.

**Tema 1 – C vs C++**

* Malloc() y free() para el uso de memoria dinámica son reemplazados por new y delete[]:
  + int\* outcomes = new int[n\_dice\*sides + 1];
  + Esta línea inicializa un puntero a una región de memoria que contiene (n\_dice\*sides + 1) enteros.
  + El tamaño de la región se desconoce en tiempo de compilación y depende de las variables en tiempo de ejecución.
  + Se gestiona la memoria manualmente.
* Para hacer casting (conversión de tipos durante la ejecución), usamos **static\_cast<tipo>variable**. Esto es una conversión segura, no como la de C.
* En C, para que una función modificara uno de sus parámetros de entrada, teníamos que forzar un call-by-reference tratando el parámetro como un puntero y pasándole la dirección de la variable.
* En C++, hay una forma de hacer este call-by-reference, indicando en los parámetros de la función un & 🡪 función(int& i, int& j). Esto significa “referencia a un entero”. Esto hace que no necesitemos usar el operador de desreferenciación, ni que haya que pasarle a la función la dirección de memoria de la variable (lo que en C hacíamos con &).
* Por tanto, esta forma de pasar valores a funciones solo se distingue al programar la función (se programa con llamada por valor o por referencia).
* Además, si queremos hacer lo mismo con variables de distinto tipo, podemos programar la función CON EL MISMO NOMBRE para los distintos tipos, y en la llamada, el compilador sabrá elegir cuál debe usar atendiendo al tipo de parámetros en la llamada. Ayuda a mantener limpieza en el nombre.
* Generics programming 🡪 programar usando plantillas, la misma sirve para muchas cosas. Nueva palabra clave **TEMPLATE:**
  + template <class type>
  + Puedes añadir esta línea encima de cualquier función. Sustituyendo type por T y cada declaración de tipo por T. Así, la función sirve para cualquier tipo genérico.
  + T será el tipo genérico, al que se le puede dar cualquier nombre.

**Tema 2 – programación genérica en C++ y grafos**

* AÑADIDO: podemos inicializar por defecto parámetros de funciones 🡪 func(int s=0) 🡪 el parámetro s es opcional. Si no se pasa, será 0. Si se pasa, tomará el valor dado (como en Python).
* Importante pasarle a las funciones variables const si estas no van a ser modificadas (la función solo las “lee”). Se hace para arrays, los int no hacen falta, ya que al pasarlos por valor (call by value) ya se deja implícito que no se van a modificar.
* Para los templates, pueden incluirse dos tipos genéricos en una función 🡪 template <class T1, class T2>. Esto se suele utilizar menos, pero hay casos en los que es útil.
* Podemos seguir usando typedef enum para constantes relacionadas entre ellas.

**Orientación a objetos:**

* Sobrecarga de operadores: Darle un nuevo significado adaptado al contexto de una clase a los operadores nativos del lenguaje:
  + inline [tipo de dato creado] operator [operador] (tipo de dato d)
  + Por ejemplo: inline days operator++ (days d) 🡪 para sobrecargar el operador ++ prefijo.
  + Entrada/salida (impresión) 🡪 (lo que está en rojo se pone siempre)

**ostream& operator** << (**ostream& out**, const type& d). Se devuelve out.

Necesita dos parámetros binarios, y se les llama por referencia por convenio. La variable que queremos imprimir no va a ser mutada, por lo que usamos const.

* + Siempre que trabajemos con un tipo que creamos, se espera que sobrecarguemos el operador << para ofrecer una forma de salida apropiada (es como el \_\_str\_\_ de Python).
* Un nuevo tipo de dato no tiene por qué ser con struct, también puede ser con typedef enum.
* La forma más común es con class, que se usa similar a struct. En C, al declarar la struct, debíamos declarar las variables que actuaban como miembros. En C++, se añaden funciones (operaciones) que pueden ser miembros dentro de la clase.
* La clase tiene dos partes: public y private. Los atributos son private siempre, y los métodos son public. Cuando escribes un método en una clase, este ya es inline por defecto.
* Métodos:
  + Los que se definen dentro de la clase serán normalmente get y set, junto con el constructor. Podrán acceder automáticamente a miembros privados.
  + Las sobrecargas de operadores van fuera de la clase. Para que puedan acceder a los miembros privados, dentro de la clase hay que declararlos como amigos 🡪 friend + declaración de la sobrecarga.
  + Se pueden sobrecargar operadores como funciones miembro de la clase, pero entonces si son binarios solo reciben un parámetro (el otro viene implícito, y se usan los atributos de la clase directamente o con this->).
  + Para definir operaciones específicas fuera del cuerpo de la clase, hay que poner classname::función. Además, hay que declararlas dentro.
  + Poner **const** en todos los parámetros de la clase que no vayan a ser modificados, y pasar por referencia (**&**) siempre (para evitar llamada por valor y copias locales indeseadas).
* El equivalente a self en Python es THIS en C++.
* Incluir siempre el CONSTRUCTOR de clase. Este puede ser el constructor ordinario (vacío) con una operación de **asignación**:
  + Clase() {miembro1 = x; miembro 2 = y;} ó
  + Clase() {this -> miembro1 = x; this->miembro2 = y;}

O puede ser el constructor específico para una clase, **inicializando** en vez de asignando:

* + Clase(arg1=default, arg2=default): miembro1(arg1), miembro2(arg2) {}.
  + Por ejemplo: point(double x=0, double y=0): x(x), y(y) {}
  + También se puede: point(): x(0.0), y(0.0) {}. 🡪 Este es el MEJOR!
  + No se hace uso del operador de asignación, sino que usa una “initialization list”, por lo que es necesaria esta forma para cuando un miembro es const.

Otro tipo de constructor es el de conversión. La idea es que este tome como parámetros otras variables (por ejemplo, un array y su tamaño), y use esa información para crear una instancia de la clase. La sintaxis es similar a la de inicialización, inicializando los atributos a algo “nulo”, y luego construyendo el objeto en el cuerpo ({}).

* CONCLUSIÓN: Usar la sintaxis de inicialización en vez de la de asignación normalmente. El primero sirve para poder meter valores en la declaración (point(2, 3)). El segundo sirve para inicializarlo a 0 por defecto con una llamada a point(), y ya modificarlo posteriormente si se quiere.

**Tema 3 – Inicialización y algoritmos**

Uso de new y delete para asignar memoria dinámicamente:

* Array: char\* s = new char[size];
* Un solo elemento: int \*p = new int(9). La variable int p toma valor 9.
* Borrar array: delete [] s. Los corchetes sirven para desasignar el array.
* Borrar elemento: delete p;

Destructores (contrario a constructores) en clases:

* La sintaxis con una clase cualquiera (p.e: class point) es:
  + class ~point(void) {código que usa “delete” de alguna forma;}
* No usarlos puede derivar en errores por fugas de memoria cuando se trabaja con muchos datos.

Operaciones de manejo de vectores:

* Constructor: vector<type> name(size, (default\_content)). Si no ponemos size, creamos vector vacío.
* Vector.size()
* Vector.empty()
* Vector.begin() y vector.end() 🡪 punteros a inicio y fin. El inicio contiene el primer elemento, el fin es la posición tras el último elemento (sentinel).
* Vector.push\_back(element) 🡪 añadir elemento al final del vector.
* Vector.resize(int) 🡪 redimensionar vector. Si ponemos vector.resize(tam, X), entonces el segundo valor inicializa los valores redimensionados del vector a lo que pongamos. Si se omite, se inicializan a 0 o NULL.
* Se puede iterar con iteradores:
  + For (vector<int>::iterator p = v.begin(); p != v.end(); p++)
  + En vez de vector<int>::iterator, podemos poner auto p (inferir el tipo de la variable al ser complicada). MEJOR USAR AUTO!!
  + El iterador es una generalización del puntero. Para acceder al valor, desreferenciar.
* La función “sort” de la librería algorithms recibe vector.begin() y vector.end().

**Tema 4 – Grafos y algoritmos sobre grafos**

* Árbol generador de peso mínimo (minimum spanning tree) 🡪 subgrafo conexo sin ciclos que conecta todos los vértices del grafo original y las aristas tienen peso mínimo. Si el grafo tiene n vértices, el árbol deberá tener n-1 aristas obligatoriamente. Se calcula con el algoritmo de Jarnik-Prim.
* Plus: asumir que cada arista puede tomar un color (RGB, indicado con clase enum). Encontrar el árbol si todas las aristas son de uno o varios colores.
* Se pueden usar enum class Name, de forma que las constantes enum se convierten en miembros de Name (Name::RED).

Algoritmo de Prim:

* Empezar por un vértice (node 0) en el árbol (cerrados). De sus nodos vecinos, coger el de la arista con menos coste.
* Mantener un conjunto de abiertos (nodos alcanzables desde un nodo ya en cerrados).
* Terminar cuando todos los nodos estén en el árbol. Si no están todos y no hay arista para coger para salir de un nodo (U empty), entonces el grafo no es conexo (ret -1).
* Componentes:
  + Array dist 🡪 distancia desde el origen a cada nodo i. Se inicializa todo a infinito.
  + Array edges 🡪 Dado un vértice i, edges[i] almacena el vértice más cercano. Se inicializa todo a nullptr (0). Es como un vector de parents.
  + Array MST 🡪 lista de vértices por los que hemos pasado (cerrados).
  + PriorityQueue U 🡪 lista de vértices alcanzables (abiertos). Se sacarán en cada momento los de menor distancia (marcada por dist).
  + Recibe array vacío de aristas (struct con dos int vértices) que se rellenará con las aristas que componen el árbol.
* Inicializar dist[source] = 0. While MST no tenga el tamaño de los vértices del grafo, ir sacando vértices de U y añadiendo la arista formada por el vértice y su “padre” almacenado en edges. Evaluar sus neighbors.

Conversión en clases:

* Si hay una clase con un constructor que necesita un parámetro, la conversión implícita funciona si a una instancia de la clase le asignas una variable del tipo del parámetro.
* Si no quieres que esto funcione hay que añadir “explicit” en el constructor. Esto permitirá solo la conversión con static\_cast<class>.

Expresión “for”:

* Se puede iterar sobre un conjunto de elementos ordenado (array, vector, string…) con una sentencia equivalente a “for i in range” 🡪 for (declaración : expresión).
* Por ejemplo 🡪 for (double d : data), donde data es un array/vector de double.