

Programación en ANSI C

Sesión 1

Verano 2017

Por: J. Guadalupe Olascuaga Cabrera

Moodle

- Propedéutico MSC programación 1/2 | Guadalupe Olascuaga | V.2017
- Matriculación: propeprogra1/2

¿A quién le sirve este curso?

- Los que participarán en el desarrollo de:
 - Software base
 - Compiladores, sistemas operativos, drivers, antivirus (mucha interacción con hardware).
 - Sistemas embebidos:
 - Programas para dispositivos con recursos limitados.
 - Aplicaciones que requieren una gestión eficiente de la memoria real
 - SMD, editores de texto, navegadores, videojuegos, algoritmos para big-data, etc.

¿A quién le sirve un poco?

- Los que participarán en el desarrollo de:
 - Software de aplicación
 - Sistemas administrativos, páginas web, aplicaciones móviles.
 - Java, C#, Ruby, PHP, Swift, ...
 - Se apoyan de un framework que resuelve la gestión de los recursos.
- ¿Por qué les sirve un poco?
 - Adquirirán habilidades de programación: si lo entendí en C ... Java será pan comido.

Saberes generales

- Tipos de datos primitivos y estructurados
 - Representación, declaración, uso, optimización
- Aritmética de apuntadores
- Manejo dinámico de la memoria
- Tipos de datos abstractos
- Listas enlazadas: simples, dobles, circulares
- Pilas, colas, recursión

Prerrequisitos

- Diseño de algoritmos
- Programación estructurada con ANSI C
 - Sintaxis, tipos de datos, modificadores, declaración de variables, constantes
 - Entrada/salida en consola
 - Estructuras de control
 - Funciones
- Familiarizado con Eclipse + MinGW

Tipos de datos primitivos

- También llamados tipos *atómicos*
 - No se pueden dividir y conservar el significado de sus partes.
 - Si el número 1,576 se descompone en cuatro datos, cada uno perdería el significado original: 70 se convierte en 7, 500 en 5, 1000 en 1.
- Los proporciona el lenguaje de programación. En C existen:
 - Números enteros: `int`
 - Números reales: `float`, `double`
 - Caracter: `char`

Tipos de datos primitivos

- C permite añadir modificadores al tipo de dato para añadir información acerca de la cantidad de memoria utilizada para guardar el dato (o el tamaño en bits):
 - `short int` (o sólo `short`) \leq `int`
 - `long int` (o sólo `long`) \geq `int`
 - `long long` \geq `long int`
 - `long double` \geq `double`
- El tamaño real depende de la arquitectura de la computadora. Ejemplo:

| | | | | | | | | | |
|----------------------|----|---------------------|----|--------------------------|----|-----------------------|----|------------------------|----|
| ▫ <code>char</code> | 8 | <code>short</code> | 16 | <code>int</code> | 32 | <code>long int</code> | 32 | <code>long long</code> | 64 |
| ▫ <code>float</code> | 32 | <code>double</code> | 64 | <code>long double</code> | 96 | | | | |

Tipos de datos primitivos

- C permite añadir el modificador **unsigned** a los tipos de datos enteros y carácter para indicar que el rango de valores incluye no incluye a los números negativos. Por omisión, sí los incluye.

- Ejemplos:

| | | |
|------------------------|------------------|--------------------------------|
| ▫ unsigned char | 0 a 255 | (de 0 a $2^8 - 1$) |
| ▫ unsigned int | 0 a 65,535 | (de 0 a $2^{16} - 1$) |
| ▫ char | -128 a 127 | (de -2^7 a $2^7 - 1$) |
| ▫ int | -32,768 a 32,767 | (de -2^{15} a $2^{15} - 1$) |

Obteniendo el tamaño con ANSI C

- El operador unario `sizeof` entrega el tamaño en bytes del operando derecho:

```
▫ int i = 50;  
▫ int s = sizeof i; // s = 4 bytes
```

- Se puede utilizar como función:

```
▫ int s = sizeof(i); // s = 4 bytes
```

- Puede actuar también con el tipo de dato, pero usado como función:

```
▫ int s1 = sizeof(short); // s = 2 bytes  
▫ int s2 = sizeof(long double); // s = 12 bytes
```

Representación de los primitivos

- En la computadora todo se almacena como una secuencia de bits: 010101.
- De **enteros sin signo** a secuencia binaria:
 - Dividir el número entre dos (*división entera*) hasta llegar a cero.
 - La secuencia binaria invertida estará formada por los residuos de todas las divisiones.
 - | | | | | | | | |
|-----|----|----|----|---|---|---|---|
| 120 | 60 | 30 | 15 | 7 | 3 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | |
 - $\text{Binario}(120) = 1111000 = 8 + 16 + 32 + 64$

Tipos de datos estructurados

- También llamados *no primitivos, complejos*
- Se componen de uno o más datos *primitivos* o *estructurados*.
- Tres grandes tipos:
 1. Arreglos
 - Arreglos de varias dimensiones
 - Cadenas de texto
 2. Estructuras o registros
 3. Uniones

Arreglos

- Conjunto de elementos del mismo tipo (primitivo o estructurado).
 - La longitud del arreglo N se conoce en su creación y no puede cambiar
- Creando arreglos con el modo tradicional:
 - `unsigned short edades[20];`
 - `float estaturas[N];`
- ANSI C no permite definir la longitud tamaño con una variable, es decir, el compilador debe conocer la longitud: N debe ser constante
(hay que comprobarlo)
 - `#define N 100`

Arreglos

- ¿Sí lo permitió?
 - Los compiladores nuevos de C han eliminado algunas restricciones.
- Modifiquemos el modo de compilación de forma que se atenga a las reglas definidas por los puristas del lenguaje C; es decir, lo hacemos *pedante*
- En Eclipse:
 - Project – Properties – C/C++ Build – GCC C Compiler – Warnings
 - Habilitar: Pedantic warning as errors (*-pedantic-errors*)

Arreglos

- A cada elemento se accede con su posición, un entero en el rango $[0.. N-1]$
 - `estaturas[5] = 1.75;`
 - `printf("%u", edades[i]);`
- A diferencia de lenguajes de programación de alto nivel, el espacio ocupado en memoria por un arreglo en C se obtiene con la fórmula:
 - Longitud del arreglo (N) \times Tamaño del tipo de dato (sizeof)
 - C no añade información adicional al arreglo (metadatos) que lo haga más grande

Arreglos

- Podemos crear arreglos especificando datos iniciales en lugar de longitud:
 - `int primos[] = {2, 3, 5, 7, 11};`
 - La longitud del arreglo será 5
- Podemos crear arreglos especificando datos iniciales y longitud:
 - `int primos[10] = {2, 3, 5, 7, 11};`
 - La longitud del arreglo será 10
 - Las posiciones 5 a $N - 1$ se llenan con el valor *default* del tipo de dato: 0.

Arreglos

- Podemos crear arreglos de 2 o más dimensiones, especificando la longitud de cada dimensión:

- `double matriz[100][100];`
 - `int cubo[75][100][30];`

- A cada elemento se accede con su posición en cada dimensión:

- `matriz[5][90] = 3.5;`
 - `int v = cubo[70][80][25];`
 - `matriz[100][5] = 3.5;`

`// Ocasiona error en tiempo de ejecución`

Arreglos

- Podemos crear arreglos de 2 o más dimensiones, con ausencia de la longitud de la dimensión más *contenedora*, especificando datos iniciales:

- En el ejemplo, la longitud faltante denota el número de pares (3):

```
int mat2[][2] = { {11, 12}, {21, 22}, {31, 32} };  
int mat2[][2] = { 11, 12, 21, 22, 31, 32 }; //Mismo resultado  
mat2[2][1] almacena el valor 32
```

- En el ejemplo, la longitud faltante denota el número de filas (2):

```
int mat3[][2][3] = { { {1, 2, 3}, {4, 5, 6} },  
                     { {7, 8, 9}, {10, 11, 12} } };  
mat3[1][0][2] almacena el valor 9
```

Arreglos

- Los elementos faltantes se rellenan con el valor *default* del tipo de dato:

- En el ejemplo, al tercer elemento le falta su segundo valor:

```
int mat2[][2] = { {11, 12}, {21, 22}, {31} };  
mat2[2][1] contiene 0
```

- En el ejemplo, a la primer fila le falta una terna:

```
int mat3[][2][3] = { { {1, 2, 3} },  
                     { {7, 8, 9}, {10, 11, 12} } };  
mat3[0][1][0] = mat3[0][1][1] = mat3[0][1][2] = 0
```

Ejercicios

- ¿Cuál es el tamaño de los siguientes arreglos?

Compruébelo en la práctica

- `float af[2400];`
- `int pesos[] = {75, 83, 56, 64};`
- `long double ald[800];`
- `unsigned char auc[9600];`
- `double matriz[30][40];`
- `int cubo[40][10][6];`
- `int mat2[][2] = { {11, 12}, {21, 22}, {31} };`

Cadenas de texto

- En C, una cadena de texto es un arreglo de caracteres con trato *especial*.
- En lugar de escribirlo así:
 - `char palabra1[] = {'H', 'o', 'l', 'a' };`
- Lo podemos escribir así:
 - `char palabra1[] = "Hola";`
 - **Diferencia:** construye un arreglo de 5 caracteres.
 - El último carácter es el de fin de cadena: `\0`.

Cadenas de texto

- En lugar de escribirlo así:
 - `char palabra1[20] = {'H', 'o', 'l', 'a' };`
- Lo podemos escribir así:
 - `char palabra1[20] = "Hola";`
- No hay diferencia.
- Para ambos, las posiciones 4 en adelante se llenan con el valor *default* que a su vez es el fin de cadena: `\0`.
- `char buf[5] = "";` → `char buf[5] = {0,0,0,0,0}`
- `char buf[5] = " ";` → `char buf[5] = {' ',0,0,0,0}`
- `char buf[5] = "a";` → `char buf[5] = {'a',0,0,0,0}`

Cadenas de texto

- En lugar de utilizar un algoritmo para imprimir cada caracter:
 - `printf("%s\n", palabra1);`
 - `puts(palabra1);`
 - `fprintf(stdout, "%s\n", palabra1);`
 - `fputs(palabra1, stdout);`
 - La impresión termina hasta encontrar el caracter '`\0`'.
- En lugar de utilizar un algoritmo para solicitar cada caracter:
 - `scanf("%s\n", palabra1);` // La cadena termina con el primer espacio
 - `scanf("%[^\n]s", palabra1);` //leer espacios
 - No buffer overflow protection
 - `gets(palabra1);` // La cadena termina con el enter
 - No buffer overflow protection
 - `fgets(buffer, MAX_SZ, stdin);` //recomendado

Cadenas de texto

- Para efectuar operaciones típicas con cadenas de texto (igualar, comparar, concatenar, obtener tamaño), podemos utilizar funciones de `<string.h>`

```
char s1[] = "hola mundo";  
char s2[20];  
strcpy(s2, s1);  
strcpy(s2, "otra cosa");
```

- El copiado termina hasta encontrar el caracter `\0` en s1.

```
if(strcmp(s1, s2) == 0) printf("Son iguales");
```

- La comparación termina hasta encontrar el primer caracter `\0`.

Estructuras o registros

- Permite que a partir de una variable podamos acceder a muchos datos.
- Cada uno de los datos puede ser primitivo o estructurado.
 - Ojo: no se permite asignar valores iniciales.
- En C:

```
struct Date {  
    short day, month, year;  
    char mname[10];  
};
```

| Date |
|--|
| day : int month : int year : int mname : char[10] |

Estructuras o registros

- En el ejemplo anterior, el tipo de dato se llama `struct Date` completo.
- ¿Cómo creamos variables que almacenen una fecha?

```
struct Date date1, date2;
```

- En la declaración, se crean los registros en memoria con datos aleatorios.

- ¿Cómo los lleno al gusto?

```
date1.day = 23;
```

```
date1.month = 7;
```

```
strcpy(date2.mname, "Enero");
```

| date1 : Date | date2 : Date |
|--|--|
| day = 23 month = 7 year = 40 mname = "oîlt" | day = 0 month = 10600 year = 64 mname = "Enero" |

Estructuras o registros

- Los datos de una estructura se almacenan de forma consecutiva en memoria, por tanto, podemos llenar la estructura como si fuera un arreglo:

```
struct Date date1 = {25, 12, 2015, "Diciembre"};
```

- Sin embargo, esto ya no es posible porque *date1* no es un arreglo:

```
date1[0] = 23; ❌
```

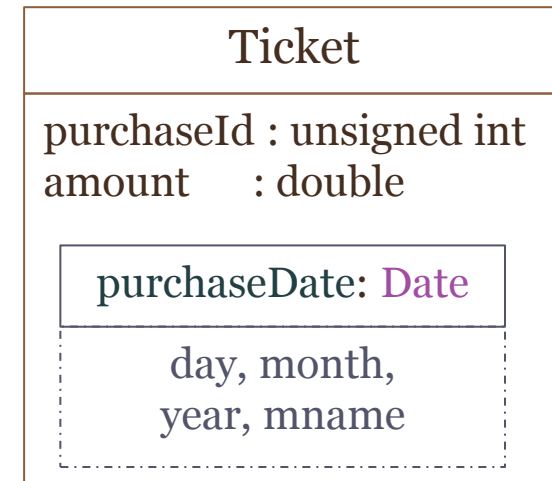
Estructuras anidadas

- ¿Puedo tener estructuras dentro de otra?
 - Sin ningún problema y siguiendo las reglas de sintaxis conocidas.

```
struct Ticket {  
    unsigned int purchaseId;  
    struct Date purchaseDate;  
    double amount;  
};
```

- ¿Y cómo accedo a ellas?

```
struct Ticket ticket1;  
ticket1.purchaseDate.day = 23;
```



Definición y declaración

- Podemos declarar variables de tipo `struct` a la vez que se define la estructura.

```
struct Ticket {  
    unsigned int purchaseId;  
    struct Date purchaseDate;  
    double amount;  
} ticket1, ticket2;
```

Arreglos de estructuras

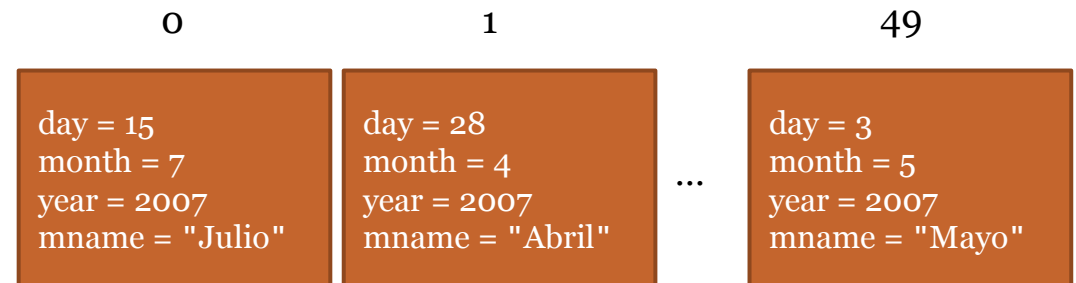
- Ya vimos que una estructura puede tener un arreglo. Pero, ¿podemos tener un arreglo de estructuras?
 - Sin ningún problema y siguiendo las reglas de sintaxis conocidas.

```
struct Date fiftyDates[50];
```

- Se crean cincuenta registros consecutivos en memoria con datos aleatorios.

- ¿Cómo manipulo las fechas?

```
fiftyDates[0].day = 15;  
fiftyDates[i].year = 2007;
```



Iguando estructuras

- Cuando realizamos asignaciones entre variables de tipo estructura, estamos copiando los datos de un lugar a otro.
- Al ejecutar este código, ¿con qué valor de mes se queda la fecha 5?

```
fiftyDates[5].month = 7;  
struct Date aDate   = fiftyDates[5];  
aDate.month         = 5;
```

Igualandando estructuras

- Al ejecutar este código, ¿con qué valor de mes se queda la fecha 5?

```
fiftyDates[5].month = 7;  
struct Date aDate = fiftyDates[5];  
aDate.month = 5;
```

- El valor del mes queda intacto: 7
- aDate es un clon de fiftyDates[5] guardado en otro lugar de memoria.
- Si no se deseaba crear un clon, es necesario hacer uso de apuntadores.

Definiendo tipos

- Una conveniencia del lenguaje C es que podemos renombrar a nuestro gusto a cualquier tipo de dato simple o estructurado.
 - Es conveniencia cuando el nombre del tipo de dato consta de varias palabras.
- Sintaxis:
 - `typedef tipo_existente nuestro_nombre;`
- Ejemplo:
 - `typedef unsigned long int uint32;`

Definiendo tipos

- Y lo podemos utilizar de cualquiera de las dos formas:

- `unsigned long int poblacion;`
- `uint32 poblacion;`
- `uint32 poblacion_estado[10];`

- No aplica para arreglos de esta manera:

- ~~`typedef float[] floatArray;`~~

- Porque `float[]` no es un tipo de dato válido

Definiendo tipos

- Pero sí se puede si se especifica el tamaño en nuestro nombre:
 - `typedef float float15[15];`
 - `float15 averages;`
 - `averages[14] = 3.5;`
- El tipo existente puede ser un tipo propio previamente definido:
 - `typedef uint32 intMatrix[20][20];`
 - `intMatrix myMatrix;`
 - `myMatrix[3][2] = 5510;`

Definiendo tipos

- Podemos definir tipos **struct** existentes:
 - `typedef struct Date date;`
- Y usarlo sin anteponer la palabra **struct**:
 - `date date1;`
- En lugar de:
 - `struct Date date1;`
- Con este esquema tuvimos que inventar dos nombres: `Date` y `date`.
 - Esto no ayuda mucho a entender el código.

Definiendo tipos

- Podemos definir el tipo de dato a la vez que definimos la estructura

- Sólo tuvimos que inventar un nombre 😊

```
typedef struct {  
    short day, month, year;  
    char mname[10];  
} Date;
```

- Y lo usamos igual:

- `Date date1, date2;`
 - `date1.day = 10;`

Enumeraciones

- Seria interesante si pudiéramos definir una variable y los tipos de valores que esta puede tomar.
- Imagina que quieres una variable para almacenar solo colores primarios y no otros valores.
 - `enum {RED, YELLOW, BLUE};`

Enumeraciones

- Una enumeración es un conjunto de valores **constantes**
 - Como es conjunto, no admite valores repetidos
- En lenguaje C, los valores tienen representación entera
 - Error: enumerator value for 'VALUE' is not an integer constant
- Se puede declarar sin asignar un nombre al conjunto:

```
enum { SOLTERO, CASADO, VIUDO };
```

- Y lo podemos usar así:

```
printf("%d\n", SOLTERO);           // Imprime 0  
int s = CASADO;                   // s = 1
```

Enumeraciones

- Los valores asignados son números enteros del 0 (cero) en adelante.
- Por lo tanto, las siguientes líneas de código son equivalentes:

```
enum { SOLTERO, CASADO, VIUDO };  
int SOLTERO = 0, CASADO = 1, VIUDO = 2;
```

- Con la diferencia que la primera no admite cambios (son constantes):

```
SOLTERO = 5;
```

- Podemos cambiar la asignación por defecto:

```
enum { SOLTERO = 3, CASADO = 5, VIUDO = 1};
```


Enumeraciones

- Podemos incluir repetidos, pero no tendría razón de ser:
`enum { SOLTERO = 3, CASADO = 5, VIUDO = 3};`
- Podemos asignar valores a unos y otros no:
`enum { SOLTERO = -1, CASADO, VIUDO, DIVORCIADO = 1};`
- Cada elemento sin un valor asignado toma el valor del elemento anterior sumado en uno:
 - `CASADO = 0`
 - `VIUDO = 1`
- También se puede asignar char.
 - `enum escapes {backspace='\b', tab='\t', newline='\n'}`

Enumeraciones

- No podemos tener dos o más elementos de la misma enumeración o de diferente enumeración con el mismo nombre, en el mismo programa.
- Tampoco con el mismo nombre a una variable global.

```
enum { SOLTERO, CASADO, VIUDO};
```

```
enum { FELIZ, TRISTE, SERIO, CASADO};
```

```
int FELIZ = 5;
```

Enumeraciones

- Como las estructuras y uniones, podemos ponerle nombre al conjunto:

```
enum EstadoCivil { SOLTERO, CASADO, VIUDO};
```

- Y así se usaría:

```
enum EstadoCivil ec = SOLTERO;
```

- Para prescindir del uso de `enum`, definimos el tipo de dato:

```
enum { SOLTERO, CASADO, VIUDO } EstadoCivil;  
typedef enum EstadoCivil { SOLTERO, CASADO, VIUDO } EstadoCivil;  
typedef enum { SOLTERO, CASADO, VIUDO } EstadoCivil;
```

Cual es la diferencia en las líneas anteriores?

- Y así se usaría suponiendo que `EstadoCivil` es un tipo de dato:

```
EstadoCivil miEstadoCivil = CASADO;  
EstadoCivil miEstadoCivil2 = (EstadoCivil)(CASADO-1); //es valido
```

Enumeraciones

- Una variable enumeración puede incluirse en la sentencia `switch`:

```
switch(miEstadoCivil) {  
    case CASADO : ... break;  
    case SOLTERO : ... break;
```

Hay que incluir un `case` por cada valor de la enumeración y/o un `default` que contemple los valores no considerados

```
}
```

- También se puede compartir el mismo valor.

```
enum EstadoCivil { no=0, off=0, yes=1, on=1};
```

- Ejercicio: Escribe un programa que imprima el número de días por cada mes usando tipos de datos enumerados.
 - El usuario proporciona el número del mes.

Apuntadores

- ¿Para qué nos pueden servir?
 1. Separar la declaración de la reserva en memoria de arreglos y estructuras.
 - Útil cuando la información se conoce en la ejecución del programa.
 2. Pase por referencia
 - No se clonan los datos cuando se pasan por una función. **Dos variables refieren al mismo dato.**
 - En la función estás trabajando con el dato original (no con una copia).
 3. Construir estructuras de datos de tamaño variable con el tiempo
 - Listas enlazadas, colas, pilas, árboles, grafos, colas de prioridad, heaps, tries, ...

Apuntadores

- ¿Qué es un apuntador?
 - Una variable capaz de almacenar la dirección de memoria de otra variable.
- El apuntador no sólo guarda esta dirección, también te permite manipular el contenido de la otra variable (de manera indirecta).
- Por ello, el apuntador debe conocer qué tipo de dato de la otra variable.
- Como toda variable, se tiene que declarar (nótese el asterisco):
 - `tipo_apuntado* nombre_apuntador;`

Apuntadores

- Declaramos un apuntador a una variable character existente:

```
char c1 = 'A';
```

```
char* pc1; char *pc1; char * pc1; (Las 3 son equivalentes)
```

- ¿Cómo indicamos que pc1 apuntará a c1?
- Con el operador **&** obtenemos la dirección de memoria de una variable.

```
char* pc1 = &c1;
```

```
printf("%p\n", pc1);
```

```
printf("%p\n", &pc1);
```

&Variable



| Variable | Dirección | Valor |
|----------|-----------|----------|
| pc1 | 0x28FF08 | 0x28FF0F |
| c1 | 0x28FF0F | 'A' |

Apuntadores

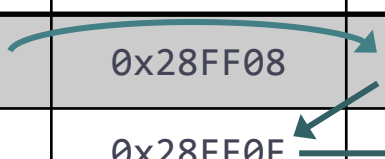
- Podemos manipular el dato (caracter) a partir del apuntador.
- Para acceder al valor apuntado se coloca * (para variar) previo al apuntador.

```
*pc1 = 'B';  
printf("%c\n", c1);
```

- Ya tenemos dos variables que refieren al mismo dato (el dueño y el metiche)

*Apuntador

| Variable | Dirección | Valor |
|----------|-----------|----------|
| pc1 | 0x28FF08 | 0x28FF0F |
| c1 | 0x28FF0F | 'A' |



Apuntadores

- Un mismo apuntador puede referirse a diferentes datos con el tiempo, siempre y cuando sean del tipo correspondiente al apuntador.

```
char c1 = 'A';  
char c2 = '6';  
char* pc = &c1;  
*pc = 'B';  
pc = &c2;  
*pc = '7';
```

| Variable | Dirección | Valor |
|----------|-----------|----------|
| pc | 0x28FF08 | 0x28FF0F |
| c1 | 0x28FF0E | 'B' |
| c2 | 0x28FF0F | '7' |

Apuntadores

- ¿De qué tamaño es una variable de tipo apuntador?
- Del tamaño de la **palabra de memoria** de la plataforma en uso.
- En casos típicos, de 64 bits.
- Compruebe que el tamaño de un apuntador a `char`, `short`, `int`, `double` y `long double` son iguales ... sin declarar variables ☺ ...
- Obsérvese en las tablas anteriores que la dirección de memoria asignada al apuntador `pc` es múltiplo de 8.

Apuntadores

```
int main() {  
    int var_int;  
    printf("Inserte Dato\n");  
    scanf("%d", &var_int);  
    return 0;  
}
```

- En la función *scanf* se usa la dirección de la variable *var_int*.

Apuntadores

- El operador `++` aplicado a un apuntador ocasiona que apunte a la siguiente dirección de memoria en donde se espera encontrar otro valor del tipo apuntado.
- Por ejemplo, si el tipo apuntado es un `float`, el operador `++` incrementa en 4 la dirección de memoria actual.
- Los operadores `--`, `+=`, `-=` aplicados a un operador modifican la dirección apuntada siguiendo la misma lógica.

Apuntadores

- Considere el siguiente ejemplo.

```
float    f = 3.14;
```

```
float* pf = &f;
```

```
pf ++;
```

```
pf += 3;
```

```
pf -= 2;
```

```
pf --;
```

```
pf = 0x0028FF0C
```

```
pf = 0x0028FF10
```

```
pf = 0x0028FF1C
```

```
pf = 0x0028FF14
```

```
pf = 0x0028FF10
```

```
anterior + 1(4B)
```

```
anterior + 3(4B)
```

```
anterior - 2(4B)
```

```
anterior - 1(4B)
```

Apuntadores

- Como no sabemos con certeza si en la dirección siguiente se encuentra un dato compatible con el tipo esperado, estos operadores son utilizados principalmente para navegar en arreglos.
- Una variable concebida como arreglo es en realidad un apuntador al primer elemento del arreglo.

```
int array[5] = {};  
printf("%p\n", array);
```

Apuntadores

- Por ser un apuntador, podemos acceder al valor apuntado con `"*"`:
`*array = 5;`
`printf("%d\n", array[0]);`
- Sin embargo, por ser un arreglo creado de manera estática no podemos cambiar la dirección apuntada. La dirección inicial debe conservarse.

~~`array++;`~~
~~`array += 4;`~~

Apuntadores


- Sin embargo, podemos crear un apuntador adicional para navegar libremente en el arreglo:

```
int* parray = array;           // Apuntan al mismo lugar
printf("%d\n", parray[0]);     // Refieren a los mismos datos
parray++;                     // parray apunta al segundo de array
*parray = 6;                  // El segundo de array guarda 6
printf("%d\n", parray[0]);     // Ahora, el primero de parray es el
                               // segundo de array
```


Apunadores

- Tal vez con una tabla se entienda mejor

| | Variable | Dirección | Valor |
|-------|----------|------------|------------|
| array | array[0] | 0x0028FEF8 | 5 |
| | array[1] | 0x0028FEFC | 8 |
| | array[2] | 0x0028FF00 | 0 |
| | array[3] | 0x0028FF04 | 0 |
| | array[4] | 0x0028FF08 | 0 |
| | parray | 0x0028FF0C | 0x0028FEFC |



```
parray ++  
parray[0] = *parray  
parray[1] = *(parray + 1)  
parray[2] = *(parray + 2)  
parray[3] = *(parray + 3)
```

Apuntadores

- Cabe mencionar que parray no está hecho exclusivamente para arreglos, es simplemente un apuntador a un entero. O sea que esto es posible:

```
int x = 100;  
parray = &x;
```

- Por lo tanto, lo siguiente también es permitido (pero no es normal):

```
parray[0] = 50;           // x cambia de 100 a 50
```

- No es normal porque podríamos usar cualquier índice y no tendríamos la certeza de qué sección de la memoria estamos cambiando.

Apuntadores a estructuras

- La declaración y asignación es semejante a los apuntadores a primitivos.

- `tipo_struct* nombre_apuntador;`
 - `nombre_apuntador = &variable_struct;`

- Ejemplo 1 [definiendo tipo]:

```
typedef struct {  
    float x, y;  
} Vec2;
```

```
Vec2 v = {3, 4};  
Vec2 *pv;  
pv = &v;
```

Apuntadores a estructuras

- Ejemplo 2 [sin definir tipo]:

```
struct Vec2 {  
    float x, y;  
};
```

```
struct Vec2 v = {3, 4};  
struct Vec2 *pv;  
pv = &v;
```

- Ejemplo 3 [declarando el apuntador de forma inmediata]:

```
struct Vec2 {  
    float x, y;
```

```
struct Vec2 v = {3, 4};  
pv = &v;
```

```
} *pv; // apunta a 0x00, (null). Si fuera un tipo primitivo no se inicializa a 0x00
```

Apuntadores a estructuras

- Al tener un apuntador a una estructura ya contamos con dos variables que refieren a los mismos datos en memoria.
- ¿Cómo accedemos a los datos a partir del apuntador?
 - **Forma 1:** `(*apuntador).atributo`
 - Primero se accede al dato apuntado (estructura), luego a su atributo.
 - Se utiliza paréntesis porque el operador “.” tiene mayor precedencia.
 - **Forma 2:** `apuntador->atributo`
 - Lo mismo pero con menos caracteres

Apuntadores a estructuras

- ¿Cómo está la información en memoria?

```
Vec2 v = {3, 4};
```

```
Vec2 *pv = &v;
```

```
float *pvy = &v.y;
```

| Variable | Dirección | Valor | Tamaño |
|----------|------------|------------|---------|
| v | 0x0028FF00 | | 8 bytes |
| v.x | 0x0028FF00 | 3.0 | 4 bytes |
| v.y | 0x0028FF04 | 4.0 | 4 bytes |
| pv | 0x0028FF08 | 0x0028FF00 | 4 bytes |
| pvy | 0x0028FF0C | 0x0028FF04 | 4 bytes |

Pase por valor

- El valor almacenado por una variable v se copia a otro lugar de memoria cuando v es pasada por una función.

- El valor original no puede cambiar.

- Ejemplo:

```
void duplicar(int i) {  
    i *= 2;  
}  
int v = 5;  
duplicar(v);
```

| Variable | Dirección | Valor |
|----------|------------|--------|
| i | 0x0028FEF0 | 5 → 10 |
| v | 0x0028FF0C | 5 |

Pase por referencia

- El valor de una variable v se puede obtener desde la función adonde es pasada, a través de un apuntador a v .
 - El valor original sí puede cambiar.

- Ejemplo:

```
void duplicar(int *i) {  
    *i *= 2;  
}  
int v = 5;  
duplicar(&v);
```

| Variable | Dirección | Valor |
|----------|------------|------------|
| i | 0x0028FEF0 | 0x0028FF0C |
| v | 0x0028FF0C | 5 → 10 |



Pase por referencia

- Para poder manipular el dato original (y no una copia) es importante acceder al valor apuntado con *. El uso excesivo de * puede generar errores de sintaxis o lógicos.
- Para reducir el uso de *, se pueden utilizar las variables temporales que sean necesarias. Sin embargo, el valor final tiene que guardarse en el apuntador recibido, con *.

```
void duplicar(int *i) {
```

```
    int t = *i;
```

Copiar el dato recibido a una variable temporal

```
    t = t * 2;
```

Procesar la variable temporal

```
    *i = t;
```

Escribir el resultado en el lugar original

```
}
```

Pase por referencia

- Los conceptos de pase por valor y referencia se conservan para uniones y estructuras.
- En el siguiente ejemplo, se pasa por referencia un vector para su modificación.

```
void duplicarV(Vec2 *pv) {  
    pv->x *= 2;  
    pv->y *= 2;  
}  
Vec2 v = {5.5, 8.0};  
duplicarV(&v);
```

| Variable | Dirección | Valor |
|----------|------------|----------------------------|
| pv | 0x0028FEE0 | 0x0028FF08 |
| v | 0x0028FF08 | {5.5, 8.0} {11.0, 16.0} |

Ejercicios

- Implemente una función *swap* capaz de intercambiar el valor entre dos números enteros de 32 bits a , b recibidos.
- Implemente una función *toUnit* que reciba un vector $2d$ V y lo haga unitario.
 - $Unit(V) = V / |V|$
 - $V=(3, 4)$, hallar un vector unitario
 - $|V|=\sqrt{3^2 + 4^2}=5$

Arreglos como argumentos

- ¿Cómo definimos que un argumento de una función es un arreglo?
 - Existen varias formas equivalentes que las estudiaremos a través de una función que imprima un arreglo en consola.
- Forma 1: definiendo el tamaño en su declaración usando una constante.
 - Se espera que el algoritmo que usa el arreglo considere dicho tamaño.

```
void print1(int array[10]) {                                     //int array[MAX]
    int i;
    for(i = 0; i < 10; i ++) printf("%d ", array[i]);           //i = 0; i < MAX
}
```

Arreglos como argumentos

- El problema con la forma 1 es que el proceso que invoca a la función no está obligado por el compilador a enviar un arreglo de dicho tamaño.

```
int array[] = {5, 6, 7};  
print1(array);           // Imprimirá siete valores basura
```

- Forma 2: indicando el tamaño como un segundo argumento.

```
void print2(int array[], int N) {  
    int i;  
    for(i = 0; i < N; i++) printf("%d ", array[i]);  
}
```

Arreglos como argumentos

- Obsérvese que en realidad N es el número de elementos que deseamos imprimir del arreglo. No hay garantías de que sea el tamaño real.

```
int array[] = {5, 6, 7, 8, 9};  
print2(array, 4);
```

- Forma 3: usando la notación apuntador y acompañado de N.

```
void print3(int *array, int N) { ... Misma implementación ... }  
//Mismo uso  
int array[] = {5, 6, 7, 8, 9};  
print3(array, 4);
```

Arreglos como argumentos

- Si en lugar de tener el arreglo, tenemos un apuntador a entero, éste puede pasarse por cualquiera de las 3 funciones mostradas:

```
int* parray = array;  
parray ++;  
print1(parray);  
print2(parray, 4);  
print3(parray, 4);
```

Matrices como argumentos

- Función que imprime una matriz especificando las longitudes de las dos dimensiones (filas = 3 y columnas = 3):

```
void print4(int matrix[3][3]) {  
    int r, c;  
    for(r = 0; r < 3; r++) {  
        for(c = 0; c < 3; c++) printf("%d\t", matrix[r][c]);  
    }  
    printf("\n");  
}
```


Matrices como argumentos

- Función que imprime una matriz omitiendo en su declaración la longitud más contenedora, pero incluyéndola como segundo argumento:

```
void print5(int matrix[][3], int R) {  
    int r, c;  
    for(r = 0; r < R; r++) {  
        for(c = 0; c < 3; c++) printf("%d\t", matrix[r][c]);  
    }  
    printf("\n");  
}
```

Matrices como argumentos

- Al igual que en las declaraciones de las matrices, sólo se puede omitir la longitud de la dimensión más contenedora.

- **Por lo tanto, esto no es posible:**

```
void print6(int matrix[][[]], int C, int R) {
```

- ¿Cómo se usarían los métodos?

```
int matrix[][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

```
print4(matrix);
```

```
print5(matrix, 2);    // Se imprimirán los datos de los 2 primeros renglones
```

Matrices como argumentos

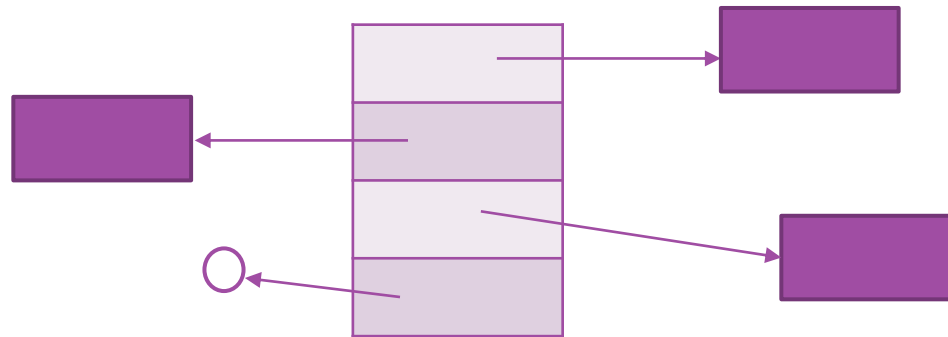
- Para imprimir los datos de una matriz de dimensiones variables, es necesario utilizar **memoria dinámica**.
 - Esto aplica para cualquier arreglo de $N > 2$ dimensiones.
- La declaración de la función sería con la notación de apuntador:

```
void print7(int** matrix, int R, int C) {
```

matrix es un apuntador a un arreglo de apuntadores a enteros
Reservar memoria para *matrix* y llamar a `print7`.
- La implementación sería igual a `print5`.

Arreglos de apuntadores

- Combinemos los conceptos que ya tenemos
- ¿Cómo puedo construir un arreglo capaz de almacenar las direcciones de memoria de otras variables?



Arreglos de apuntadores

- Sintaxis para crear un arreglo en modo estático:
 - `Tipo Nombre[Tamaño];`
 - `Tipo Nombre[] = { valor1, valor2, ... }`
 - `Tipo Nombre[Tamaño] = { valor1, valor2, ... } // Llena con 0 lo sobrante`
- Sintaxis para declarar un apuntador a un tipo:
 - `Tipo* Nombre;`
 - `Tipo* Nombre = &variable;`

Arreglos de apuntadores

- Por tanto, la sintaxis para crear un arreglo de apuntadores es:
 - `Tipo* Nombre[Tamaño];`
 - `Tipo* Nombre[] = { &variable1, &variable2, ... }`
 - `Tipo* Nombre[Tamaño] = { &variable1, &variable2, ... }`
- Crea dos enteros a , b y un arreglo de tamaño 4 que, en sus posiciones 0 y 1, almacene las direcciones de a , b . En las posiciones 2 y 3, el *default*.
- Crea un entero c . El arreglo almacenará la dirección de c en la posición 2.

Arreglos de apuntadores

- ¿Cómo reciben tal arreglo como argumento de una función?

1. `Funcion(Tipo* Nombre[4])`

- Se espera que el arreglo enviado tenga tamaño 4.

2. `Funcion(Tipo* Nombre[], int N)`

- N representa el número de apuntadores a procesar el arreglo

3. `Funcion(Tipo** Nombre, int N)`

- Este esquema puede confundir al arreglo de apuntadores con una matriz.
- Aunque técnicamente la matriz es un arreglo de apuntadores.

Arreglos de apuntadores

- Crear una función (con el esquema 2) que incremente en uno todos los valores apuntados por un arreglo de enteros recibido.
 - Excepto aquellos apuntadores nulos (\emptyset).
- Envíe el arreglo creado anteriormente por esta función e imprima su contenido después de la llamada a la función.
- Crear otra función (con el esquema 3) que haga la misma operación pero que manipule el arreglo como una matriz de N filas y una columna.