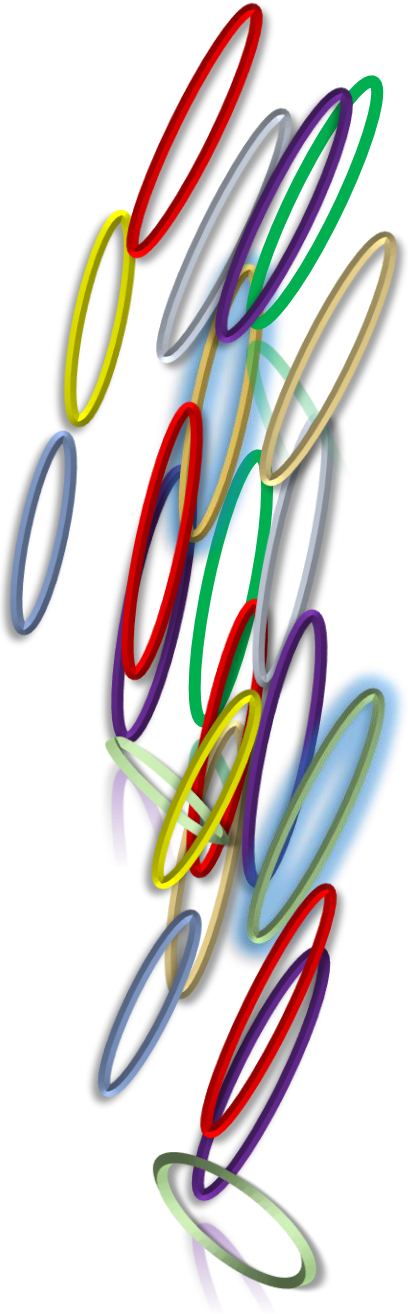


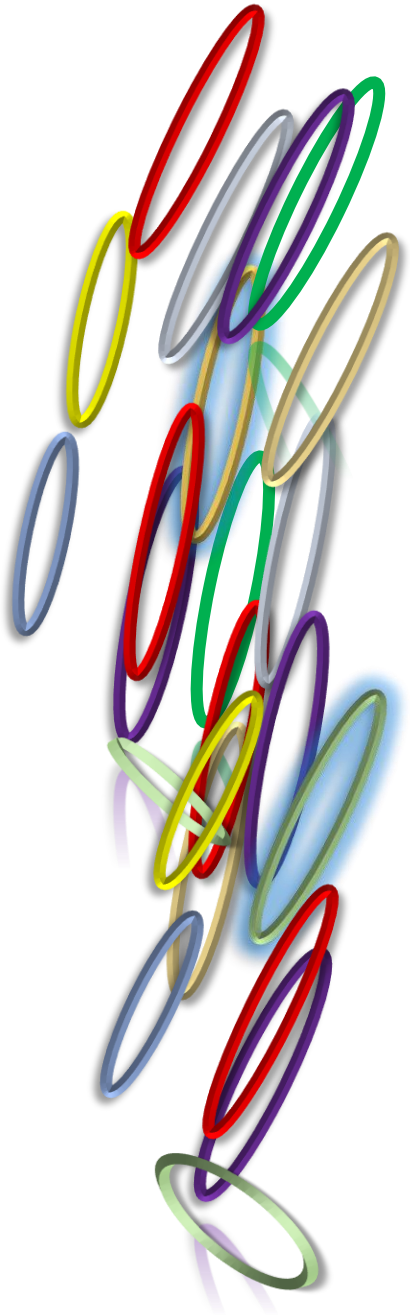
Unidad N°: 2

Programación avanzada en lenguaje C

Abril 2023

Curso: 2R3

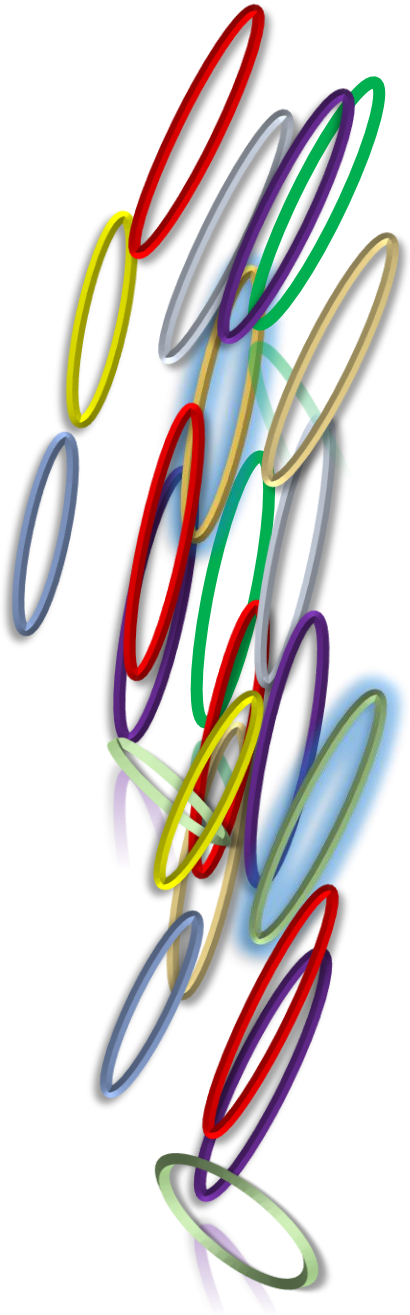
Docente: Mgter. Ing. Silvia Carrera



Unidad N°: 2

Contenido:

- Arreglos estáticos y dinámicos.
- Arreglos y punteros. Su relación.
- Aritmética de puntero.
- Parámetros de punteros en funciones.
- Clases de almacenamiento y persistencia de variables, palabras reservadas auto, register, static, extern.
- Persistencia automática y estática. Alcance de variables. Calificadores volatile y const.
- Calificador const con punteros.



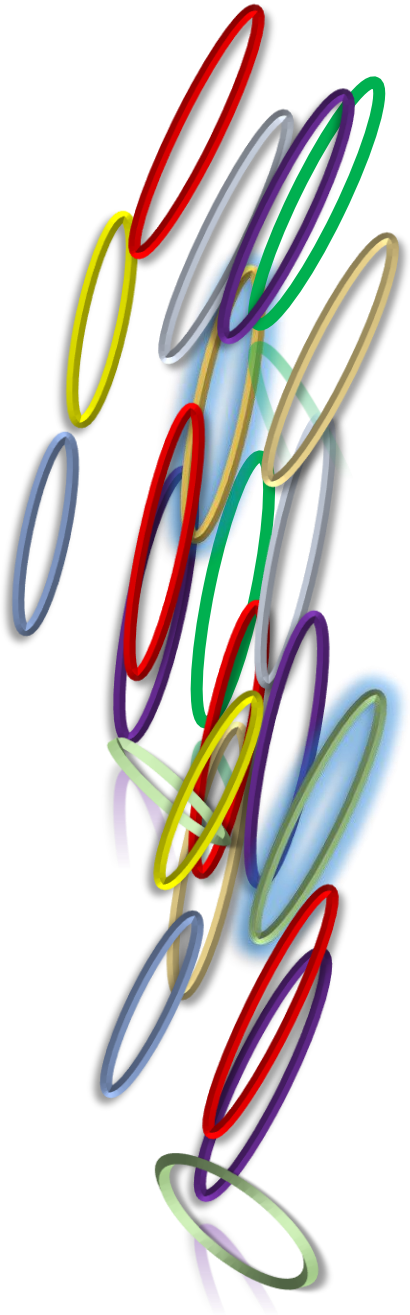
Arreglos

- ❑ **Definición**
- ❑ **Características**
- ❑ **Tipos de arreglos: estáticos y dinámicos**
- ❑ **Operaciones con arreglos**

Abril 2023

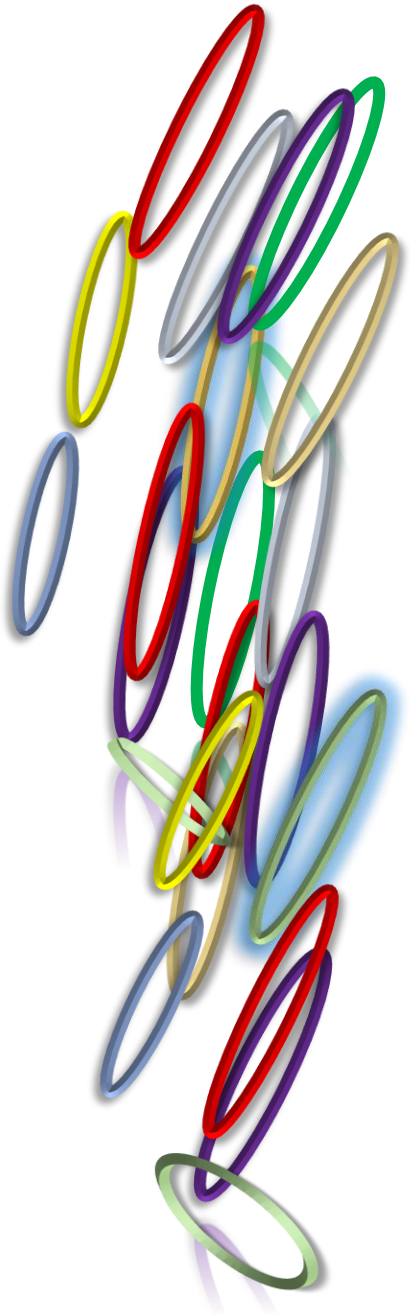
Curso: 2R3

Docente: Mgter. Ing. Silvia Carrera



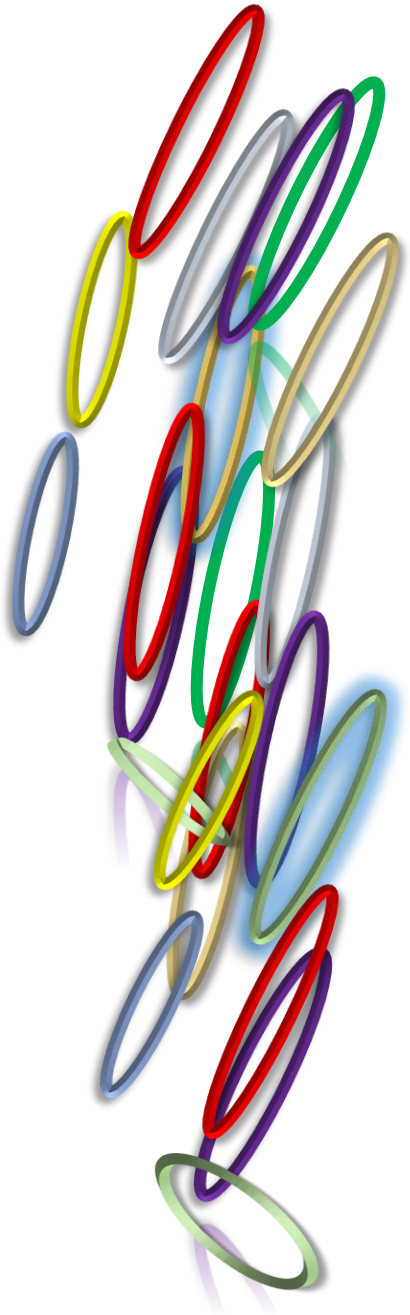
Arreglos dinámicos

- **Se crean arreglos dinámicos en tiempo de ejecución.**
- **En C se crean arreglos mediante la función malloc ( ) y se libera memoria mediante free ( ), ambas de la librería stdlib.h**



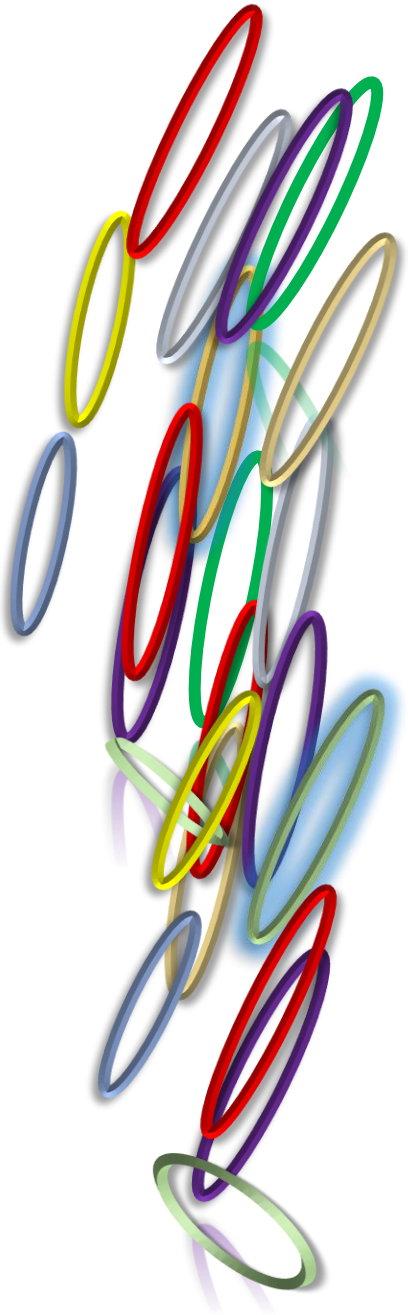
Arreglos dinámicos

- **Los arreglos dinámicos ocupan posiciones físicamente contiguas en memoria ram dinámico.**



Arreglos dinámicos y estáticos

Discusión: Conjeture sobre las ventajas y desventajas de emplear unos u otros.



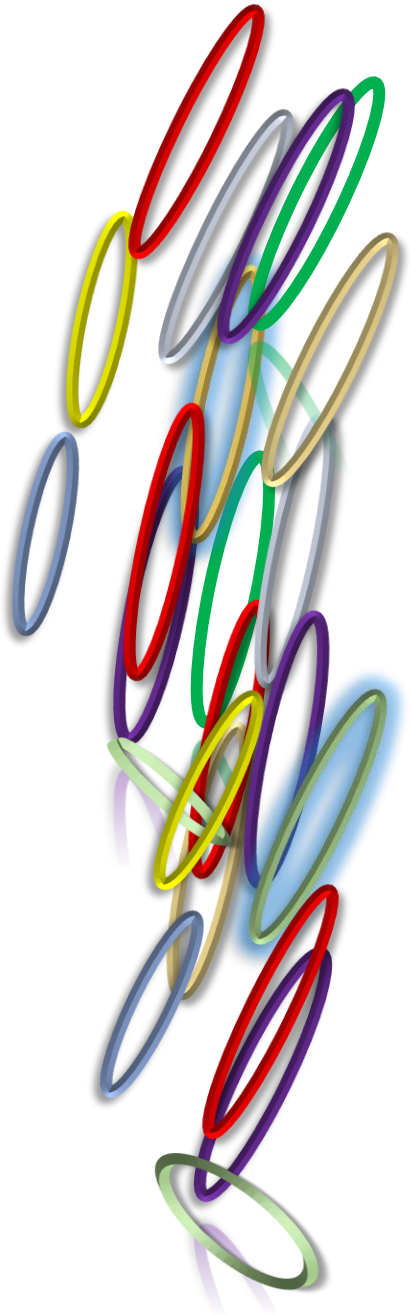
Punteros

- ❑ **Definición**
- ❑ **Características**
- ❑ **Operaciones con arreglos**

Abril 2023

Curso: 2R3

Docente: Mgter. Ing. Silvia Carrera



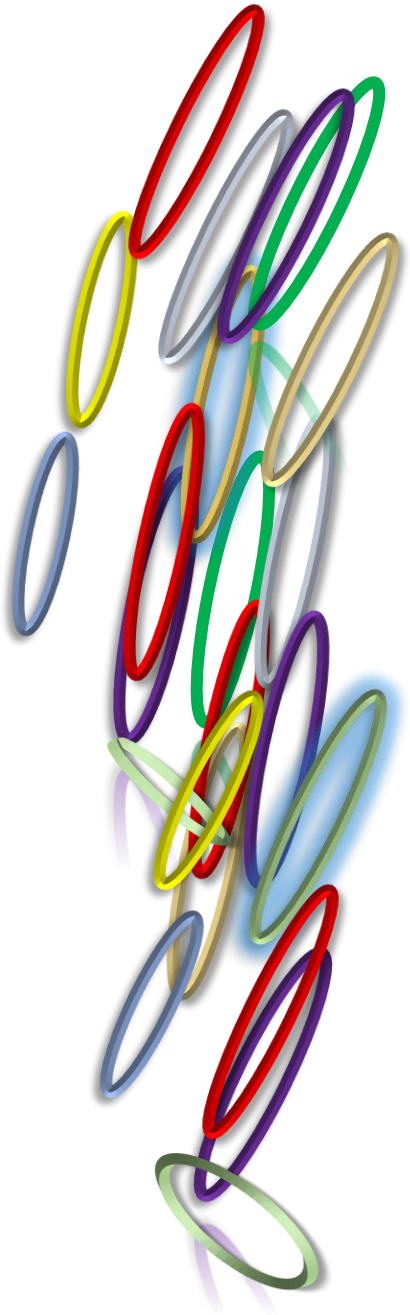
Arreglos y Punteros

Un puntero es un objeto que tiene la dirección de, es decir “apunta a”, otro objeto. El destino de un puntero puede ser cualquier objeto, incluyendo una variable, un array, una cadena, una estructura, una unión u otro puntero.

C/C++ posee aritmética de punteros. Esta es una poderosa herramienta para generar rápidamente un código eficiente.

Por ejemplo:

```
int  *p;  // puntero a un entero
int ar [10]; //arreglo creado en compilación
p = &ar; // p = &ar [0];
```



Arreglos y Punteros

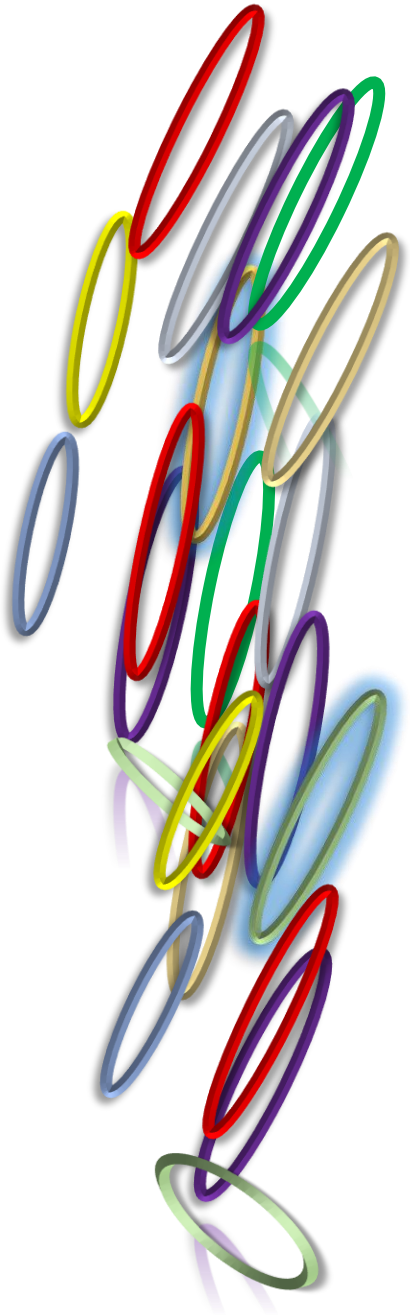
El nombre de un arreglo es un apuntador a su primera posición, por lo tanto podríamos usar la sintaxis de aritmética de punteros para recorrer un arreglo unidimensional.
int ar [7];

para guardar valores en el arreglo:

```
for ( int i =0 ; i < 7; i++)
    scanf (“%d”,&(ar +i) ) ;
```

para recorrer el arreglo y mostrar sus elementos por pantalla haríamos así:

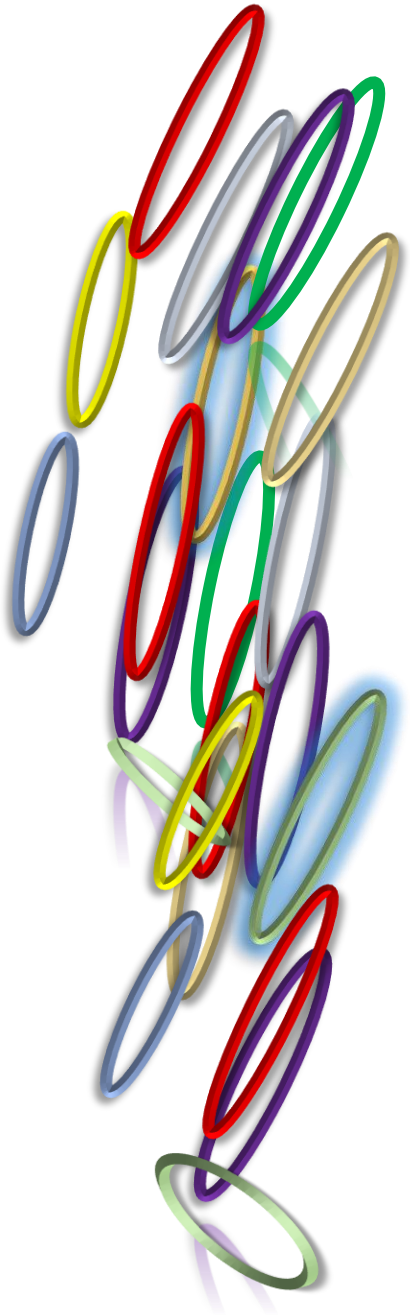
```
for (i=0; i< 7 ;i++){
        printf ( “ %d”, * (ar+i)); // el asterisco indica contenido de
        }
……
```



Aritmética de punteros

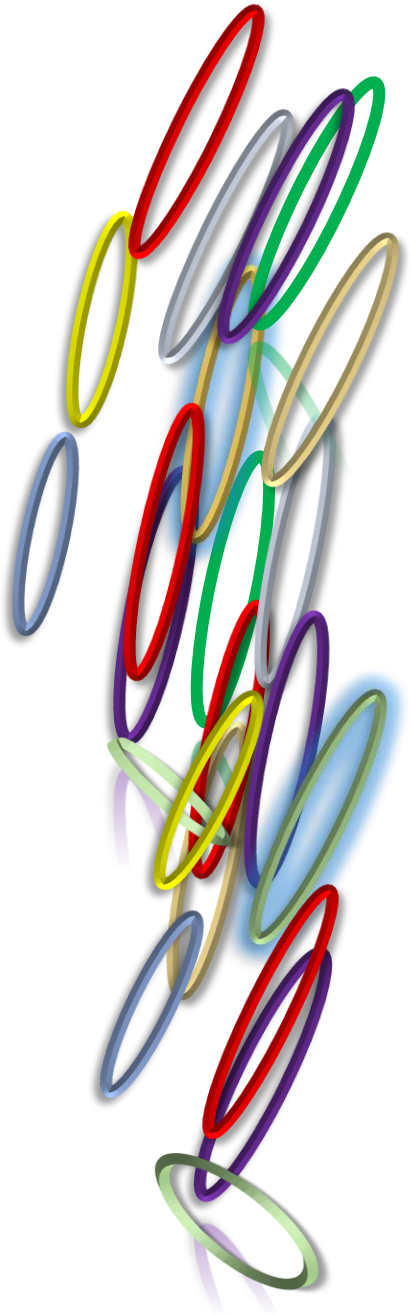
Operaciones permitidas

```
int *p2, *p1, n;
int ar [ ]= [ 1,2,3,4,5,6,7,8,9,10];
p1 = &ar;
p1 ++;//desplazamiento ascendente de 1 elemento
p1 - -;//desplazamiento descendente de un elemento
p1 + n; //desplazamiento ascendente de n elementos
p1 – n;//desplazamiento descendente de n elementos
p1 – p2; //distancia entre elementos
p2=p1;// asignación
```



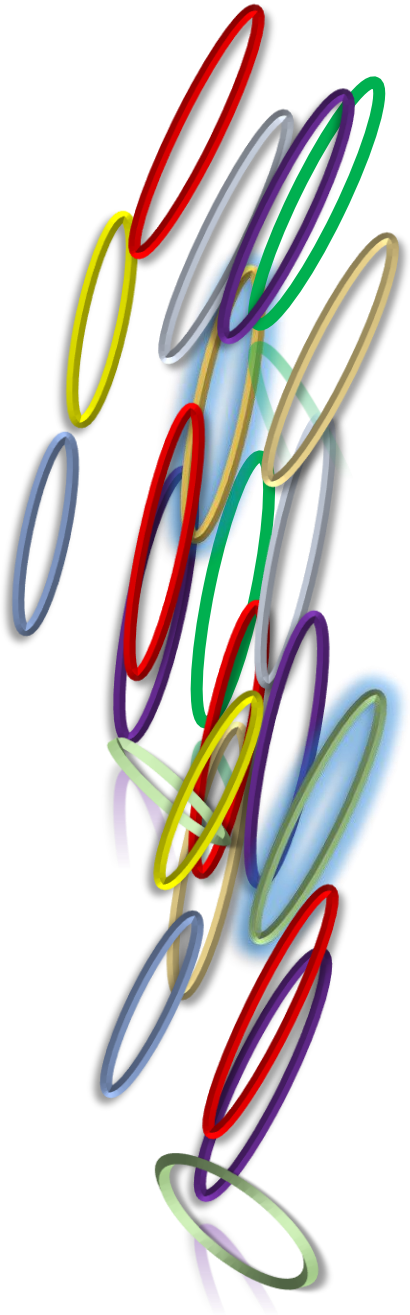
Arreglos y Punteros

La sintaxis:
ptro + K es un puntero que apunta a la dirección de ptro, sumándole k veces el espacio de almacenamiento que ocupa según su tipo.



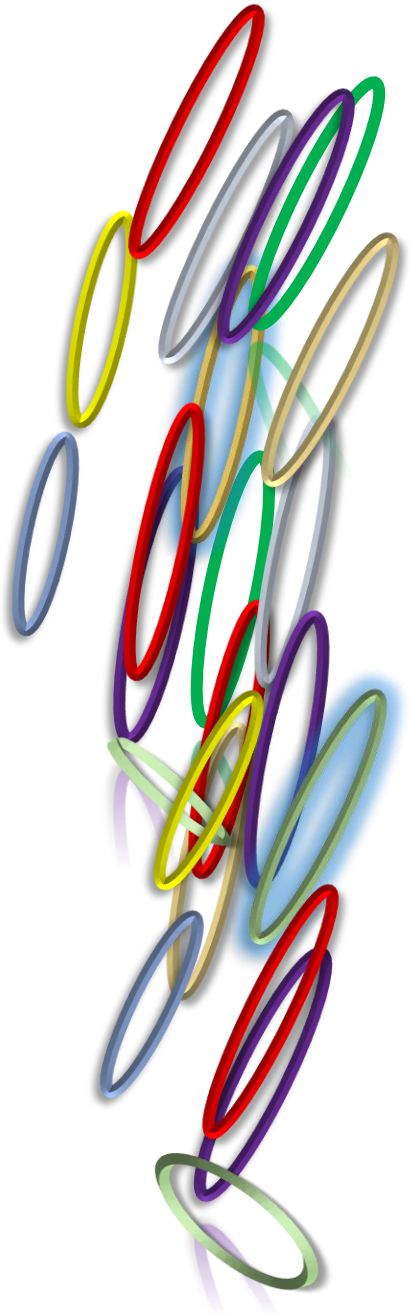
Punteros como parámetros: C

```
#include <stdio.h>
void  Cambia_mayus_a_minus ( char *); // prototipo de la función
void main ( ) {  char dato;  //declaración de variables
                    do { printf ( " Ingresar una letra");
                    scanf ( " %c", &dato);
                    while ( !(dato>=' A' && dato < = 'Z' || dato > = ' a' && dato< = 'z') );
                    if (dato>=' A' && dato < = 'Z')
                   Cambia_mayus_a_minus ( &dato); // se pasa la dirección
                   printf (" la letra minúscula ingresada es: %c", dato);  }  // fin del main
 void Cambia-mayus_a_minus (char * dato)
{   *dato = *dato + 32;}   //al contenido de dato le sumo 32 y lo guardo en dato
```



Punteros como parámetros: C ++

```
#include <stdio.h>
void  Cambia_mayus_a_minus ( char &); // prototipo de la función
void main ( ) {  char dato;  //declaración de variables
                    do { printf ( " Ingresar una letra");
                    scanf ( " %c", &dato);
                    while ( !(dato>=' A' && dato < = 'Z' || dato > = ' a' && dato< = 'z') );
                    if (dato>=' A' && dato < = 'Z')
                   Cambia_mayus_a_minus ( dato);
                    printf (" la letra minúscula ingresada es: %c", dato);  }  // fin del main
 void Cambia-mayus_a_minus (char &dato)
{   dato = dato + 32;}   //al contenido de dato le sumo 32 y lo guardo en dato
```



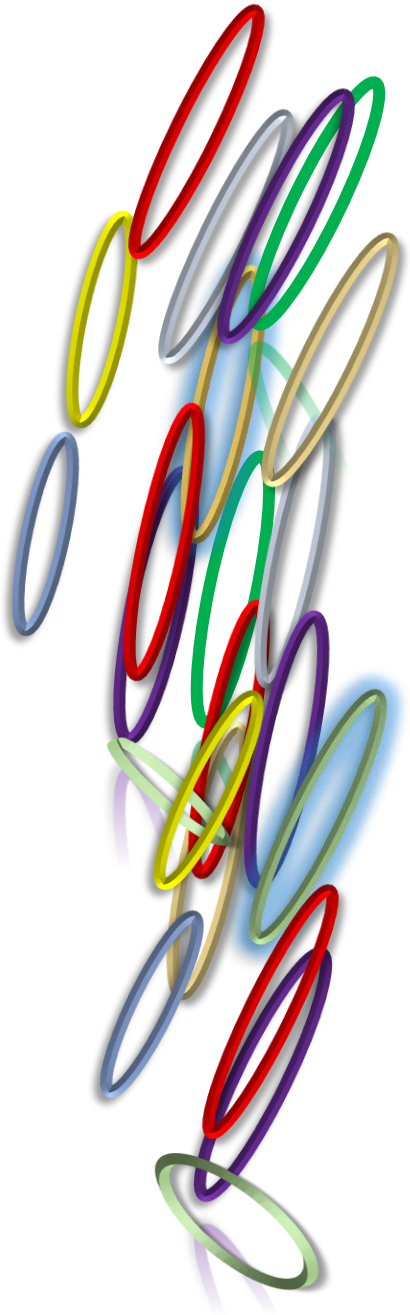
Punteros a funciones en C

En info 1:

a) Uso de punteros vinculados datos de un programa

b) Definición de variables para almacenar direcciones de memoria en cuyas celdas se encuentran almacenados datos.

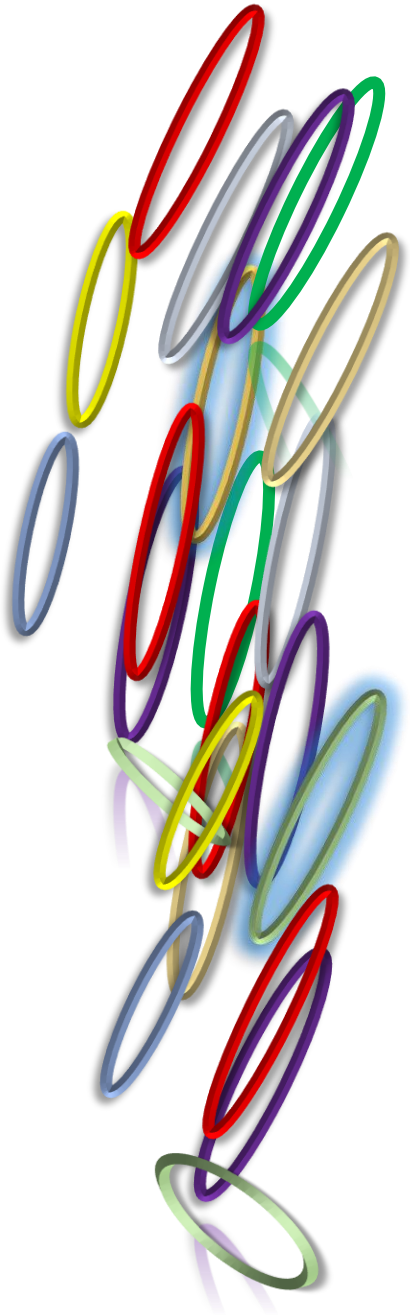
c) Declaración de variables de tipo puntero y asignarles direcciones válidas de manera dinámica (new o malloc) o de manera estática (asignando la dirección de memoria de otra variable)



Punteros a funciones en C

En info 2: (ampliando el concepto de puntero)

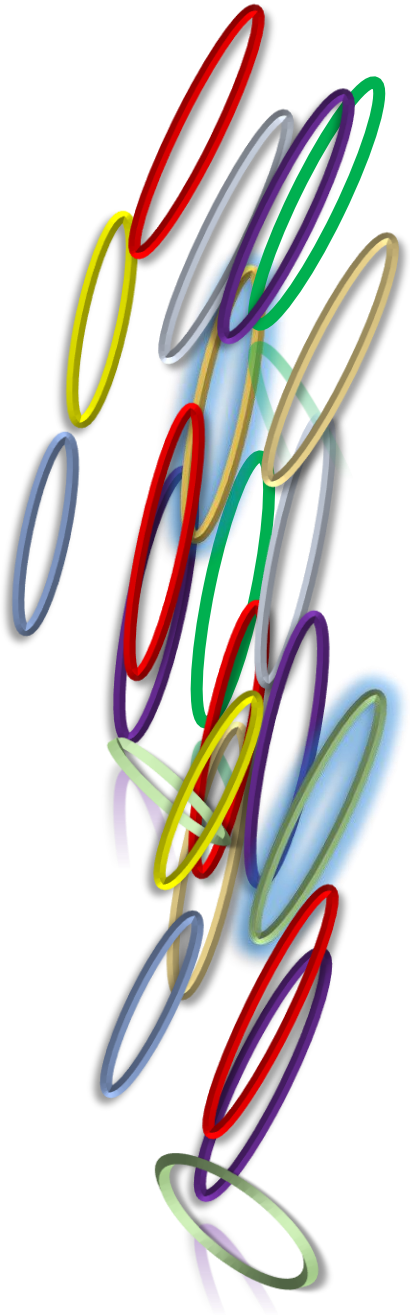
La dirección que se le asigna a un puntero puede ser la de cualquier parte memoria y en esa dirección podemos hallar información que se puede **interpretar como un dato, como ejecutable.**



Punteros a funciones en C

En info 2: (ampliando el concepto de puntero)

- El identificador de un puntero es un nombre relacionado con una **posición de memoria** y esto nos es útil para referir a variables y funciones.
- El **identificador de una función** está relacionado con la dirección de inicio del código de la función.
- El identificador de una función se puede emplear para **asignar una dirección de memoria a un puntero.**
- Un puntero a función es puntero, con **una forma de declaración especial**, que puede recibir la dirección de inicio del código de una función y ejecutarla a través de él.



Punteros a funciones en C

Declaración de un puntero a función:

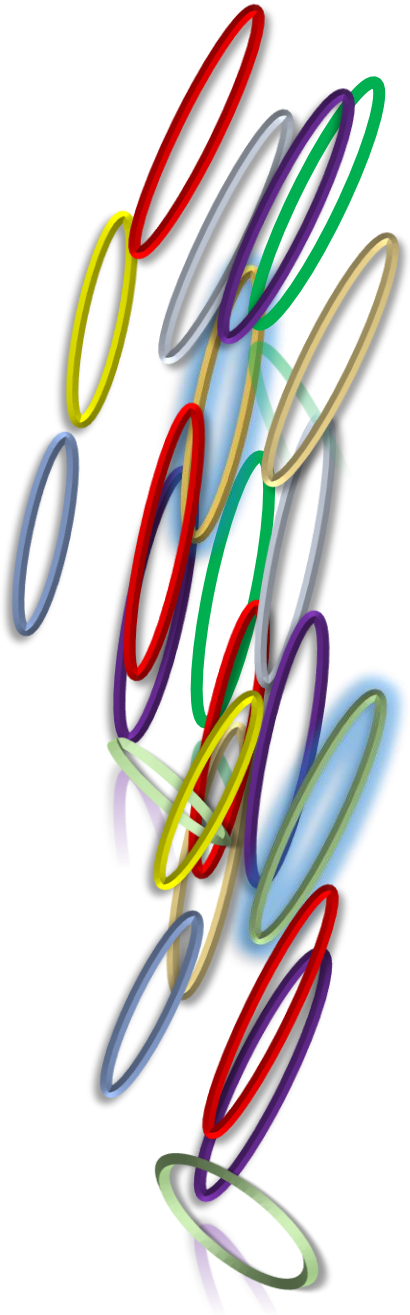
tipo_retorno (* func) (lista_parámetros)

Por ejemplo:

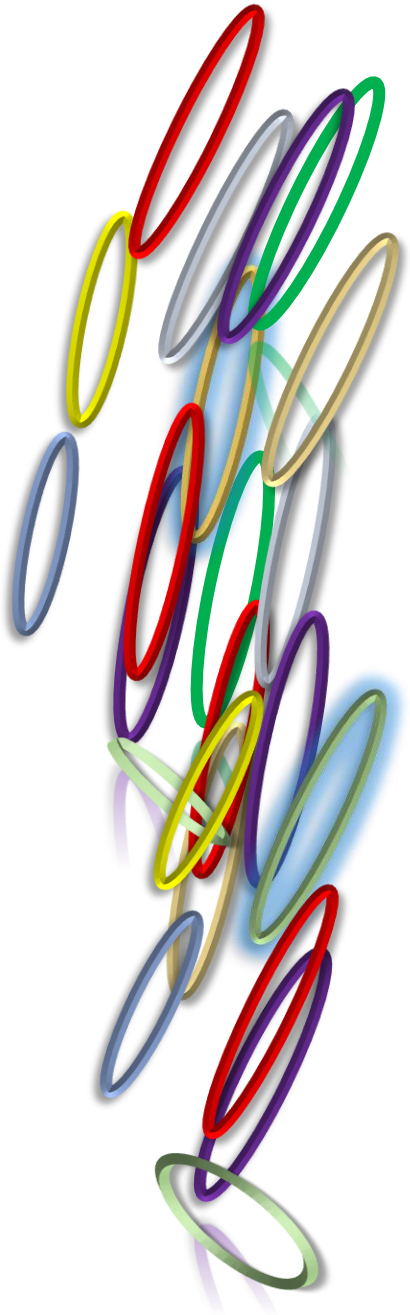
float (*func)(float, float, int) → **func** **es el identificador capaz de apuntar al inicio del código de cualquier función que tenga como parámetro dos float y un entero y devuelve un valor float**

Cuidado: float *func (float, float, int) → **func** sin paréntesis es interpretado por el compilador como una función y **NO COMO UN PUNTERO.**

Punteros a funciones en C



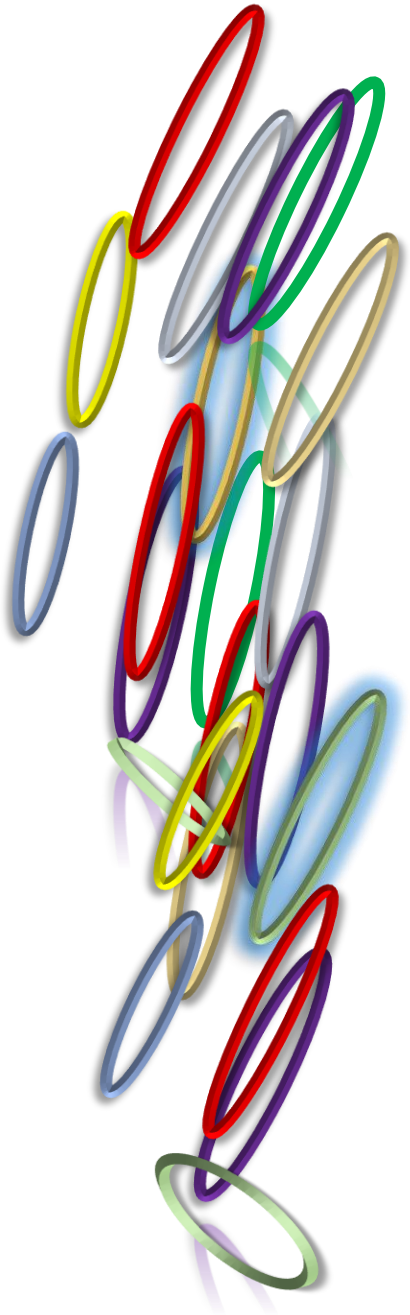
- Un puntero a función es una **variable** que almacena la **dirección de una función**.
- Esta función puede ser llamada a través del puntero.
- Es útil para encapsular comportamiento.
- EL ASPERISCO JUNTO CON EL IDENTIFICADOR VAN OBLIGATORIAMENTE ENTRE PARÉNTESIS.



Punteros a funciones en C

Ejemplo 1

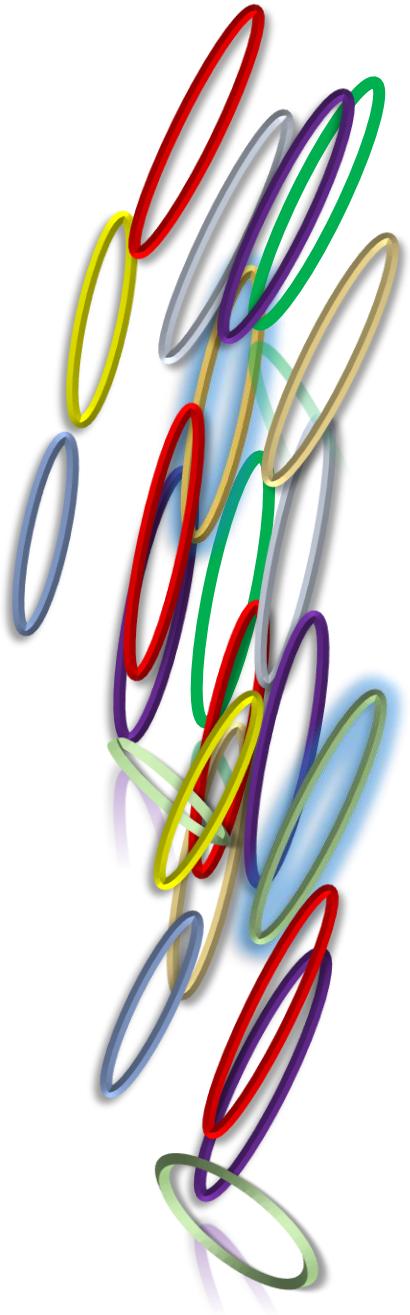
```
#include <stdio.h>
void  print ( ) {  printf ( “mensajexx\ n”); }
int main ( ) {  void ( *ptro_a_func) (void) = print;
                    ptro_a_func ( ); //invoca a print
                return 0; }
```



Punteros a funciones en C

Ejemplo 2

```
#include <stdio.h>
int  doble (int x ) {  return 2*x; }
int main ( ) {  int ( *ptro_a_func) (int) ; // puntero a función
   int x= 2; int   var1, var2; var1 = doble (x);
   printf ( “función doble de %d es %d\ n”, x, var1);
   ptro_a_func = doble; // asigna dirección de inicio
   var2= ptro_a_func (x); //devolución de doble
   printf ( “función doble de %d es %d\ n”, x, var2);  return 0; }
```

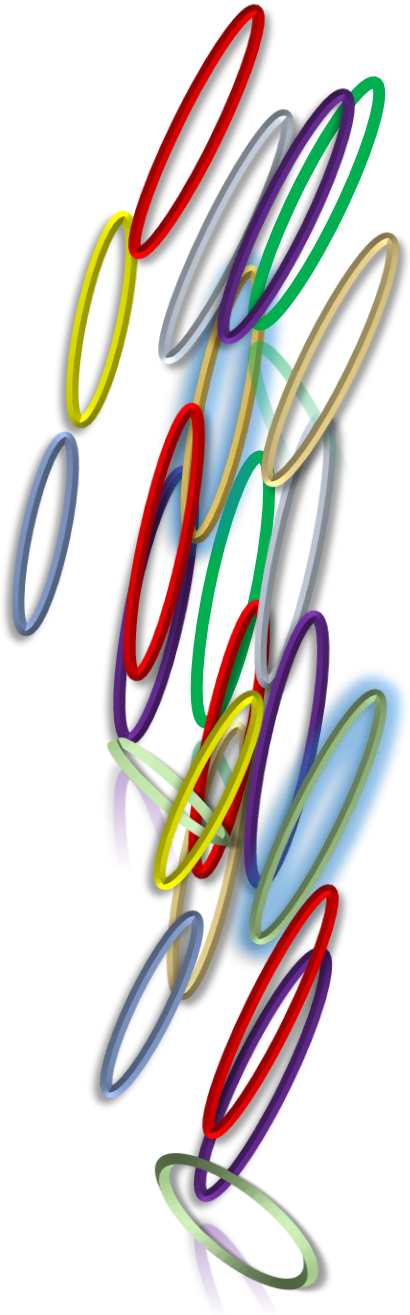


¡Ahora trabaja usted!

Codificar un ejercicio en C que permita ingresar un número entero de entre 4 o 5 dígitos, desde el teclado. Una función haciendo uso de un algoritmo lo descomponga en sus dígitos y los muestre en pantalla en orden invertido.

Usar punteros, punteros a funciones y otros elementos de código vistos y que sean pertinentes para la resolución de la tarea solicitada.

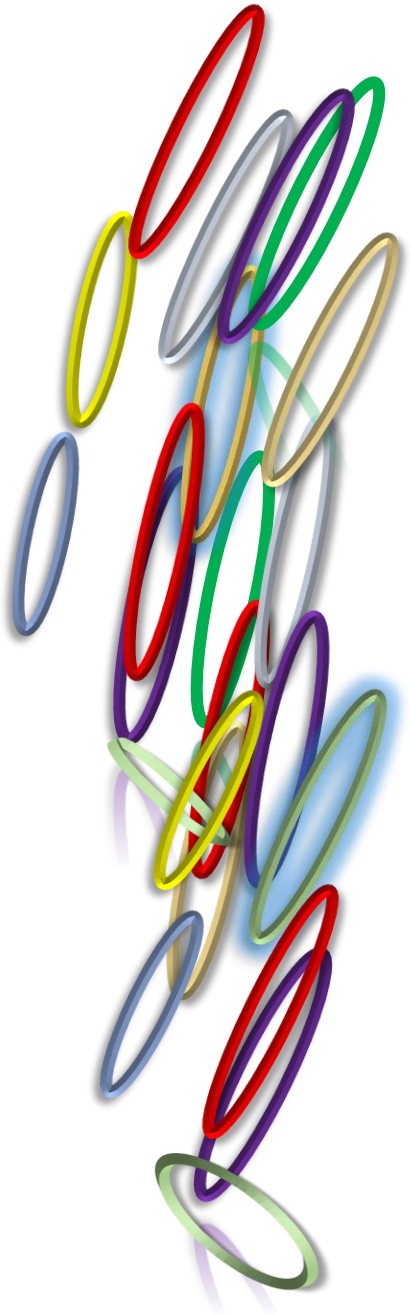
Ejemplo dato = 1256
Muestra en pantalla: 6 5 2 1



Clase de almacenamiento

Características:

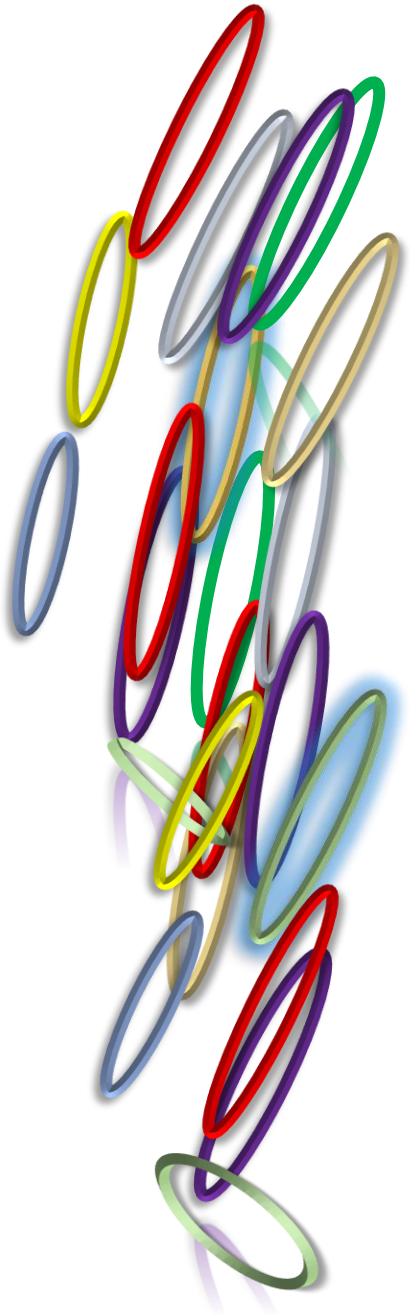
- Se visualizan en la declaraciones.
- **Identificadores de los programas tienen atributos**: tipo, tamaño, valor, tipo de almacenamiento, alcance y enlace.
- C++ tiene 4 tipos especificadores de clase de almacenamiento: auto, register, extern y static.
- El especificador clase de almacenamiento de un identificador determina: su clase de almacenamiento, alcance y enlace.
-



Clase de almacenamiento

Características:

- El especificador clase de almacenamiento determina: el período durante el cual existe en memoria un identificador.
- Pueden existir: a) durante un período breve, otros b) se crean y destruyen repetidamente y c) otros existen durante toda la ejecución del programa.
-

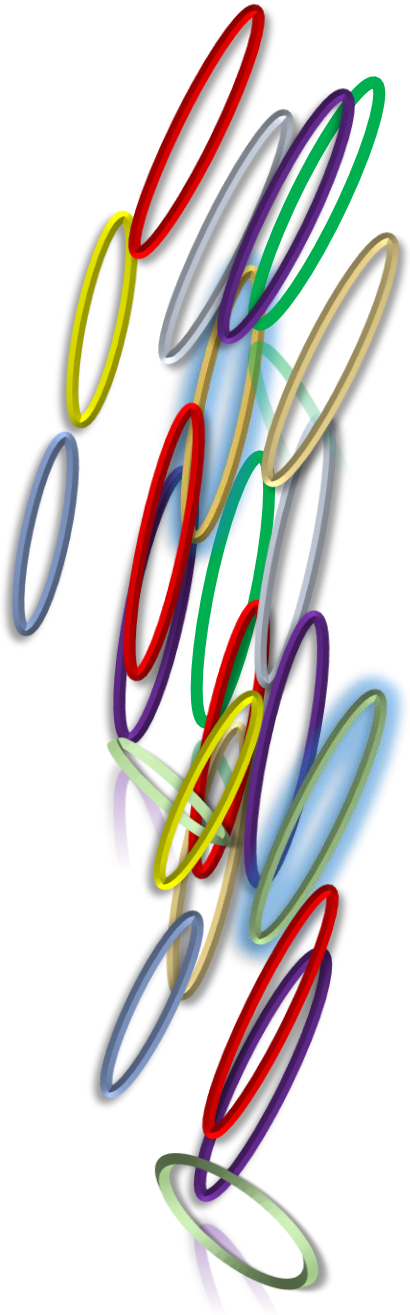


Alcance

Características:

El alcance de un identificador determina donde puede ser referenciado en el programa:

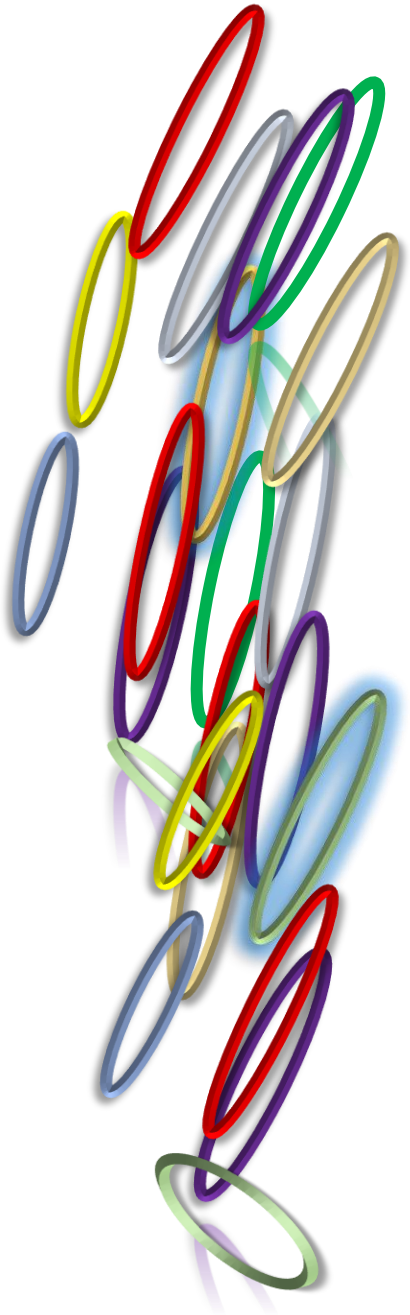
- ❖ Algunos, a lo largo de todo el programa.
- ❖ Algunos en ciertas o algunas partes del programa.



Enlace

Características:

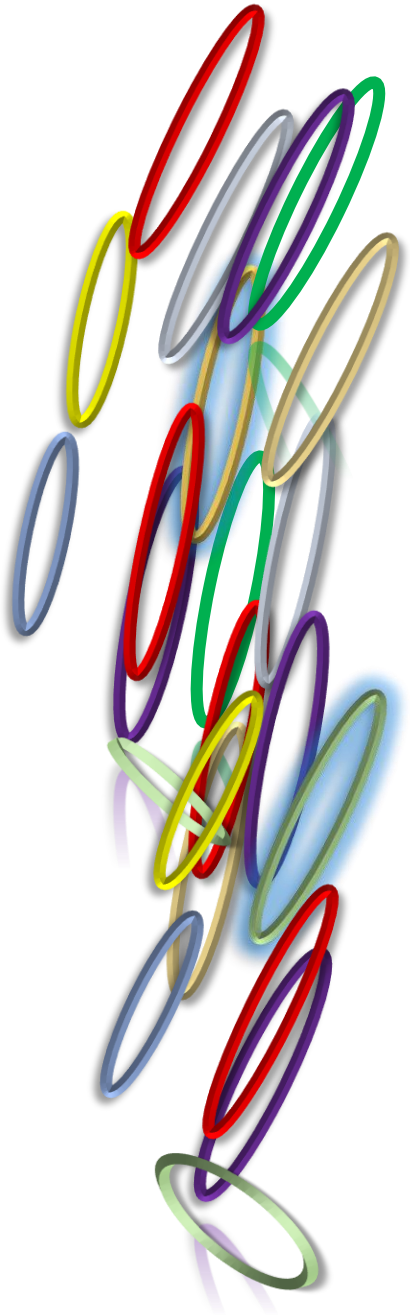
El enlace de un identificador en programa con múltiples archivos fuentes, si es conocido sólo o únicamente en el archivo fuente actual o en cualquier archivo fuente para que tenga las declaraciones apropiadas el programa.



Especificadores de clase de almacenaniento

Clase de almacenamiento automático

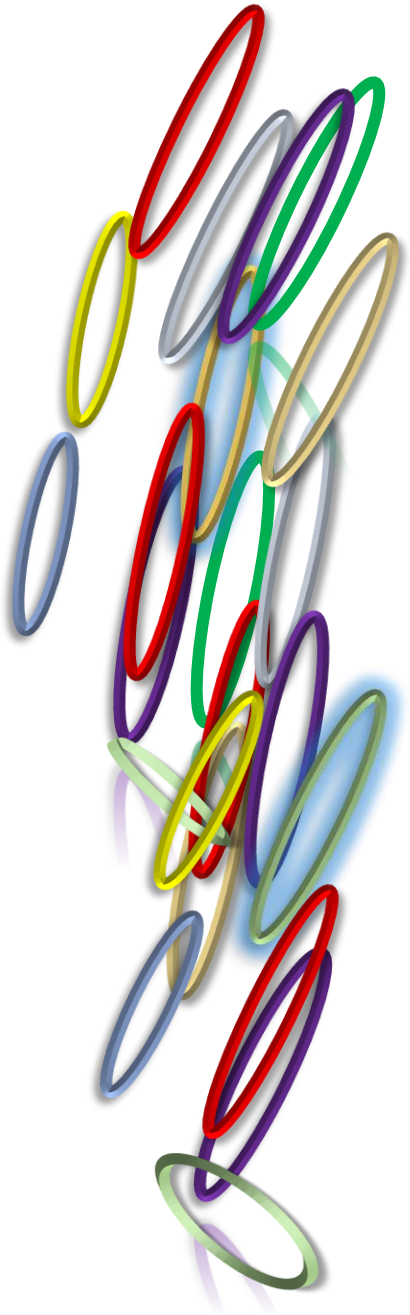
- Sólo las variables pueden ser de clase de almacenamiento automático.
- Las variables locales y los parámetros de funciones son generalmente de almacenamiento automático.
- auto int dato, info;
  expresa que dato e info son variables locales de clase de almacenamiento automático, lo que implica que sólo existen en el cuerpo de la función en la que aparece su definición.
- De forma predeterminada las variables locales son de la clase de almacenamiento automático.



Especificadores de clase de almacenaniento

Clase de almacenamiento automático

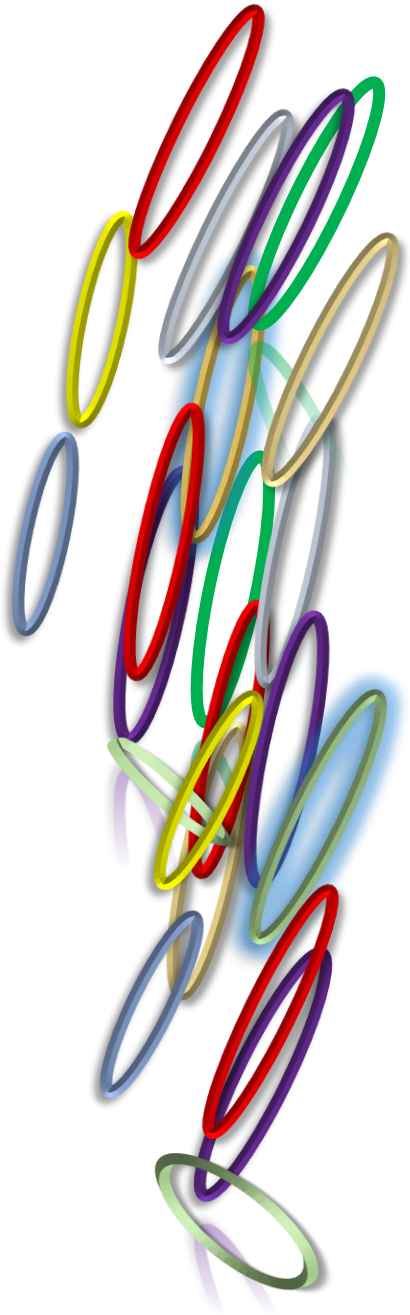
- Se emplea para conservar memoria. Se crean al entrar al bloque en el que se declaran al salir de él.
- Almacenamiento auto es un ejemplo de principio de menor privilegio.



Especificadores de clase de almacenaniento

Clase de almacenamiento register

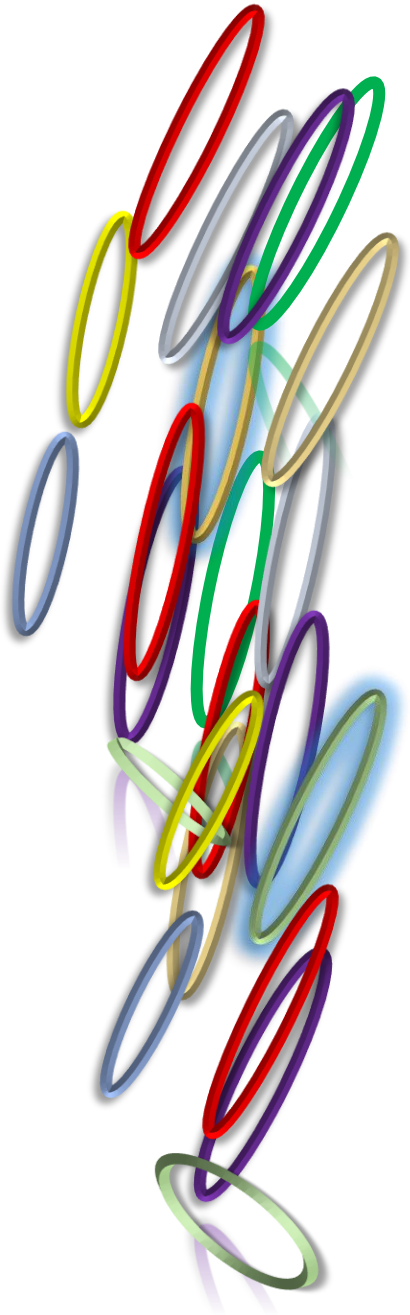
- Puede utilizarse anteponiéndolo a la declaración de una variable automática para indicarle al hardware que la almacene en un registro del micro, que es de mayor velocidad que la memoria (sistema).
- Si hay registros disponibles, variables de uso frecuente como contadores se puede mediante la palabra reservada register hacer que permanezcan en registros internos del micro con lo cual se evita la sobrecarga de obtener repetidamente las variables de la memoria y cargarlas en los registros y luego volver a almacenar los contenidos en memoria.



Especificadores de clase de almacenaniento

Clase de almacenamiento

Sólo se puede aplicar un especificador de clase de almacenamiento a un especificador.
Ejemplo: si se utiliza la palabra register, no emplear la palabra auto.
Es un error de sintaxis utilizar varios especificadores de clase de almacenamiento para un mismo identificador



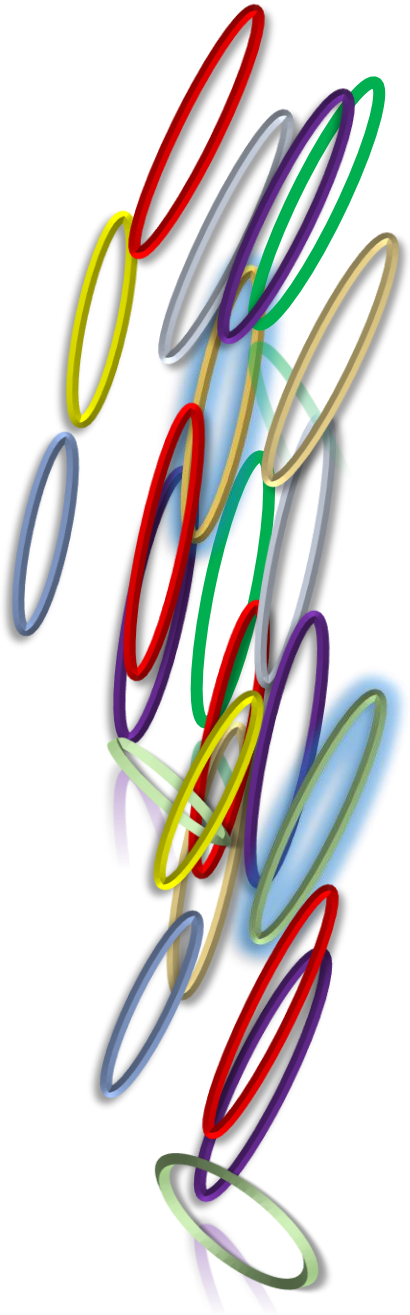
Especificadores de clase de almacenaniento

Clase de almacenamiento register

register sólo se puede* utilizar *con variables y parámetros de función locales.

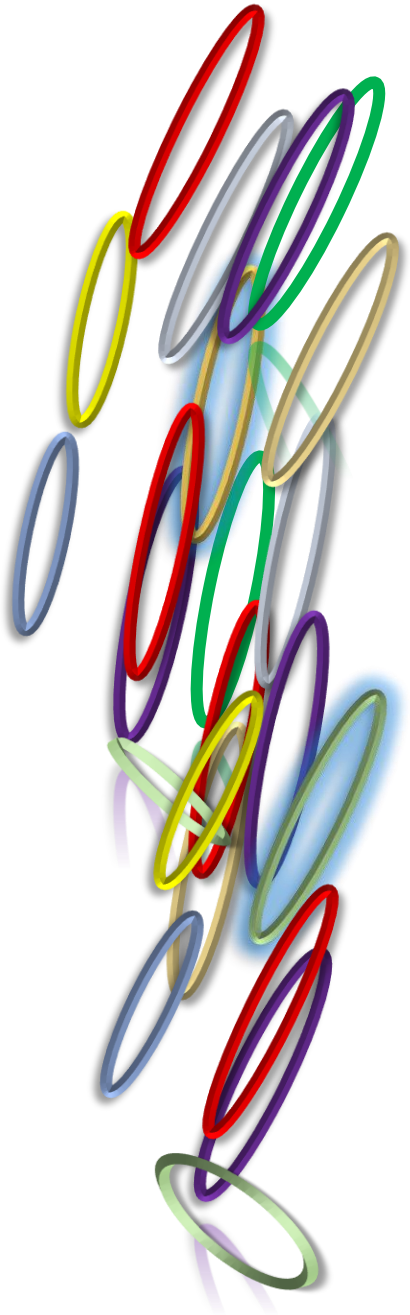
El compilador puede ignorar el efecto del uso de register, cuando no tiene disponible registros del hardware .

Ejemplo: register int contador1 = 0;



Palabra reservada extern

Extern tiene 2 significados diferentes en C++, cada uno con una determinada sintaxis: a) especificador de tipo de enlazado y b) especificador de tipo de almacenamiento.



Palabra reservada extern

Especificador de tipo de enlazado

Define el ámbito. Indica si el mismo identificador o nombre de variable o función en otro ámbito se refiere al mismo objeto u otro diferente.

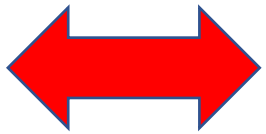
a) Enlazado externo
b) Enlazado interno
c) Sin enlazado

Palabra reservada extern

Especificador de tipo de enlazado: **Enlazado externo**

Las instancia de un identificador o nombre de objeto con **enlazado externo** representa el mismo objeto a través del total de archivos o ficheros y librerías que componen el programa.

Es el tipo de enlazado a utilizar con objetos cuyo identificador puede ser utilizado en unidades de compilación distinta de aquella en la que se ha definido. Por esta razón se dice que los identificadores con enlazado externo son "globales" para el programa.

Visibilidad global  **Enlazado externo**

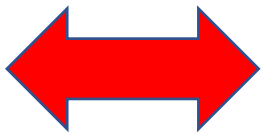
Palabra reservada extern

Especificador de tipo de enlazado: **Enlazado interno**

Las instancia de un nombre o identificador con **enlazado interno** se refiere al mismo objeto unicamente dentro del mismo fichero o archivo.

Los objetos con el mismo nombre en otros ficheros son objetos distintos.

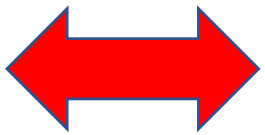
Estos tipo de objetos solo pueden utilizarse en la unidad de compilación en que se han definido, por lo que suele decirse que los identificadores con enlazado interno son "locales" a sus unidades de compilación.

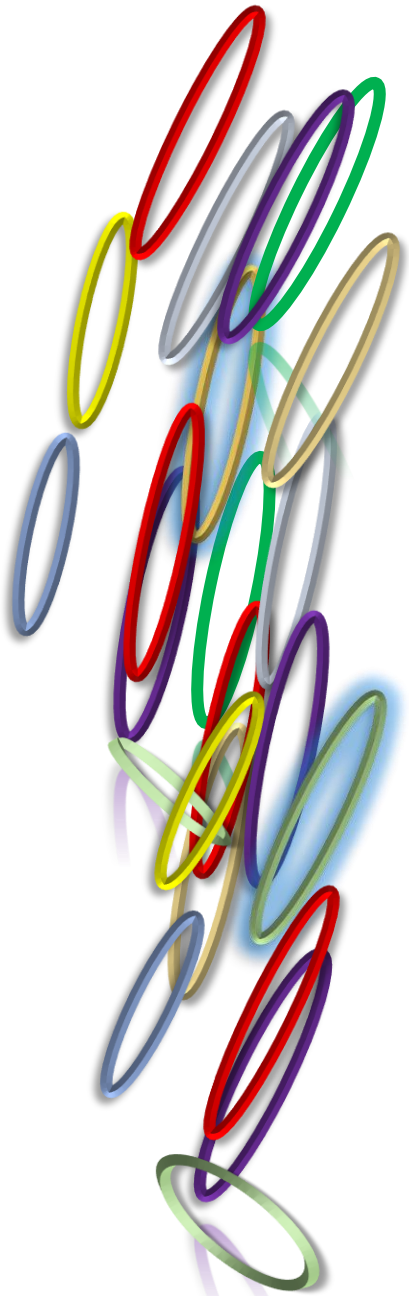
Visibilidad de archivo  **Enlazado interno**

Palabra reservada extern

Especificador de tipo de enlazado: **Sin enlazado**

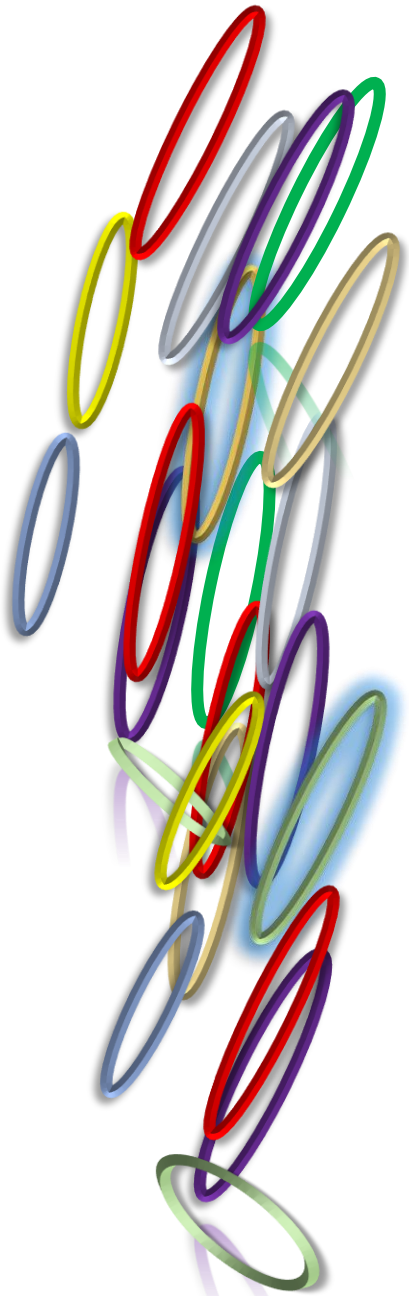
Los objetos **sin enlazado** representan entidades únicas.
Ejemplo: las variables declaradas dentro de un bloque (sin modificador **extern)**, son entidades únicas dentro del bloque, sin relación con el exterior del bloque.
Los objetos con el mismo nombre en otros bloques son objetos distintos.
NOTA: se puede asignar punteros a estos objetos sin enlazado, de forma que puedan ser accedidos desde cualquier punto del programa, incluso desde otras unidades de compilación.

Visibilidad de bloque  **Sin enlazado**



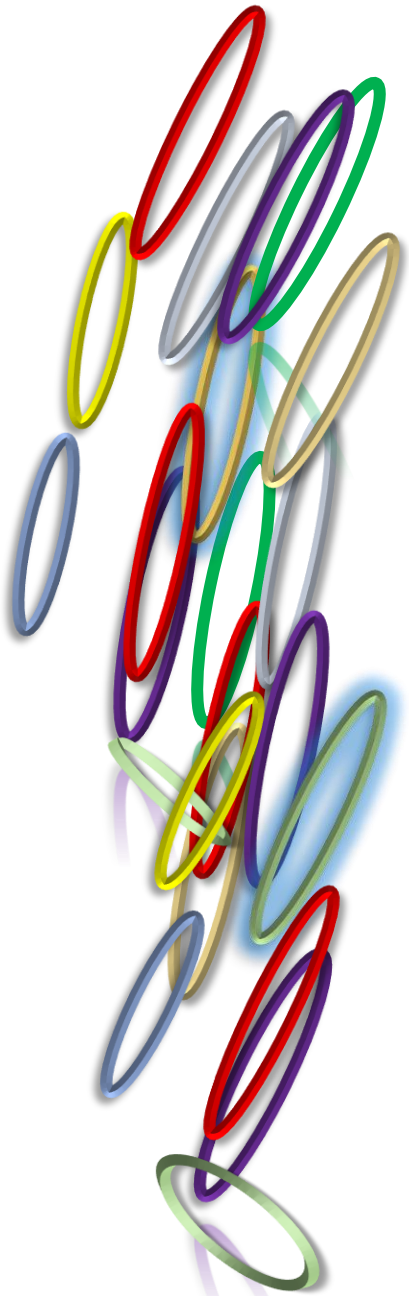
Reglas de enlazado

1. En un fichero, objeto o identificador con ámbito global deberá tener enlazado interno si su declaración contiene el especificador static.
2. Si el mismo identificador aparece con ambos enlazados externo e interno, dentro del mismo fichero, tendrá enlazado externo.
3. Cuando la declaración de un objeto o función aparece el especificador de tipo de almacenamiento extern, el identificador tiene el mismo enlazado que cualquier declaración visible del identificador con ámbito global. Si no existiera tal declaración visible, el identificador tiene enlazado externo.
4. Si una función es declarada sin especificador de tipo de almacenamiento, su enlazado es el que correspondería si se hubiese utilizado extern (extern se supone por defecto en los prototipos de funciones).



Reglas de enlazado

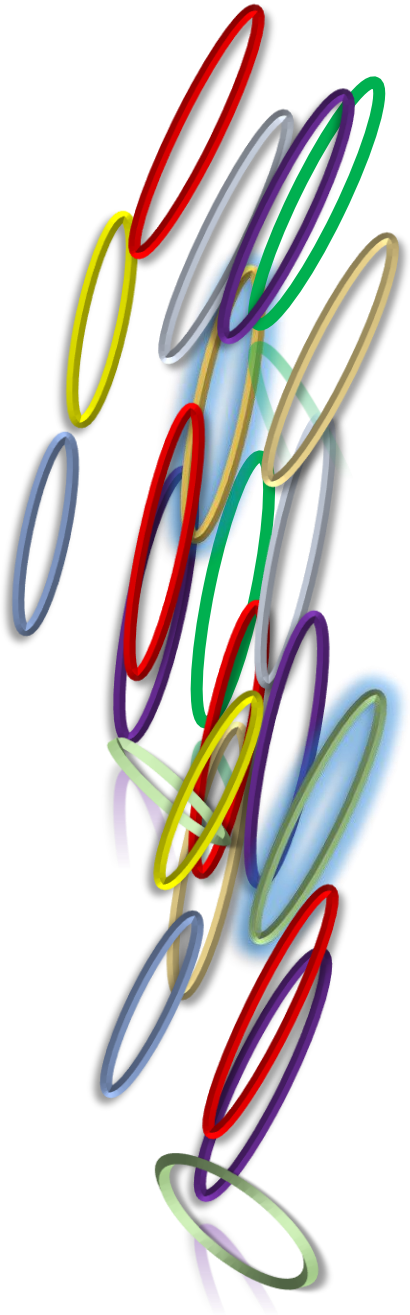
1. Si un objeto que no es una función y de ámbito global a un fichero es declarado sin especificar un tipo de almacenamiento, dicho identificador tendrá enlazado externo (ámbito de todo el programa). Excepto los objetos declarados const que no hayan sido declarados explícitamente extern tienen enlazado interno.
2. Los identificadores que respondan a alguna de las condiciones que siguen tienen un atributo sin enlazado:
   a) Cualquier identificador distinto de un objeto o una función (por ejemplo, un identificador typedef).
   b) parámetros de funciones.
   c) Identificadores para objetos de ámbito de bloque, entre corchetes { } , que sean declarados sin el especificador de clase extern.



Ejemplo de aplicación

```
int info;
static dato = 0;
void HACE( int);
int main() {   for (info = 0; info < 10; info++) HACE (info); }
void HACE (int j) {   dato += j;
  cout << dato << endl; }
```

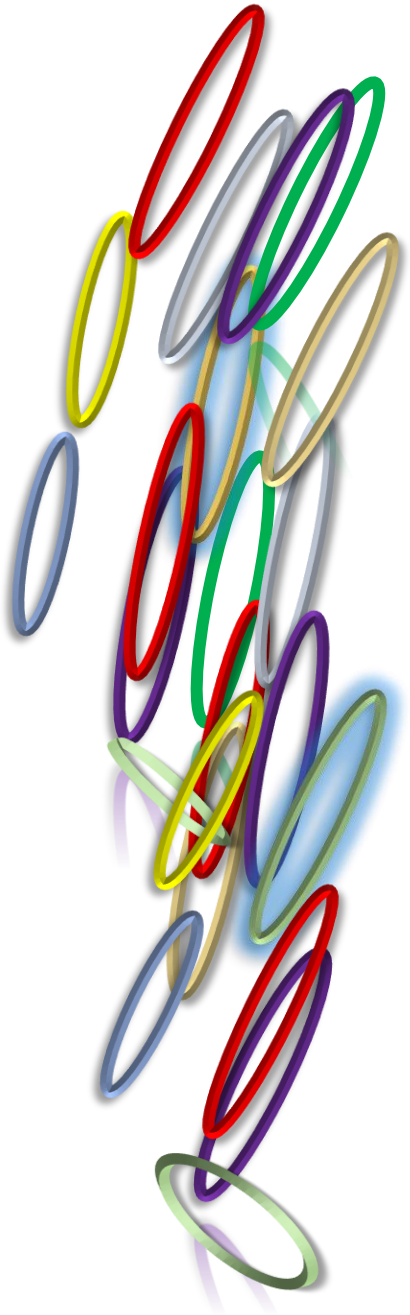
NOTA: a) info tiene enlazado externo → declarada fuera de cualquier bloque o función. Podría ser utilizada desde cualquier otro módulo que la declarase a su vez como extern b) j pertenece al ámbito de HACE, por lo que es sin enlazado c) dato tiene enlazado interno, está definida como static, es accesible desde dentro de su propio módulo



Especificadores de clase de almacenaniento

Clase de almacenamiento extern y static

- extern y static para variables y funciones de clase de almacenamiento estático.
- Por defecto (predeterminado) las variables y funciones globales son de clase de almacenamiento extern.
- Las variables globales se crean colocando su declaración fuera de toda función. Estas variables conservan su valor a lo largo de la ejecución del programa.
- **Las variables y funciones globales pueden ser referenciada por cualquier función que siga su declaración o definición en el archivo.**

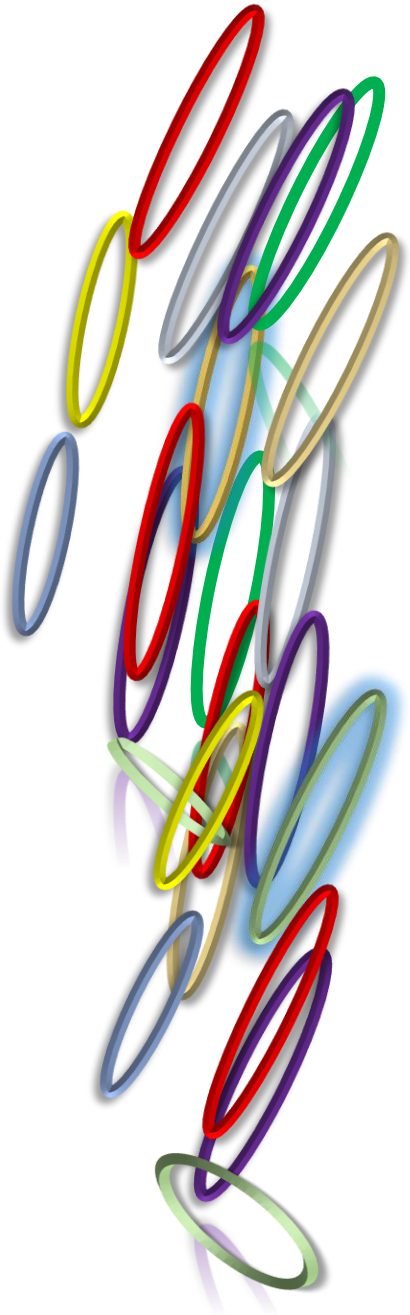


Especificadores de clase de almacenaniento

Clase de almacenamiento static

static int contador1=0;

La variable declarada de esta manera **sólo es conocida dentro de la función en la que se la define**. Dentro de una función se iniciará a 1 la primera vez que se ingrese a la función, luego **llamada tras llamada conservará su valor al salir de la función.**

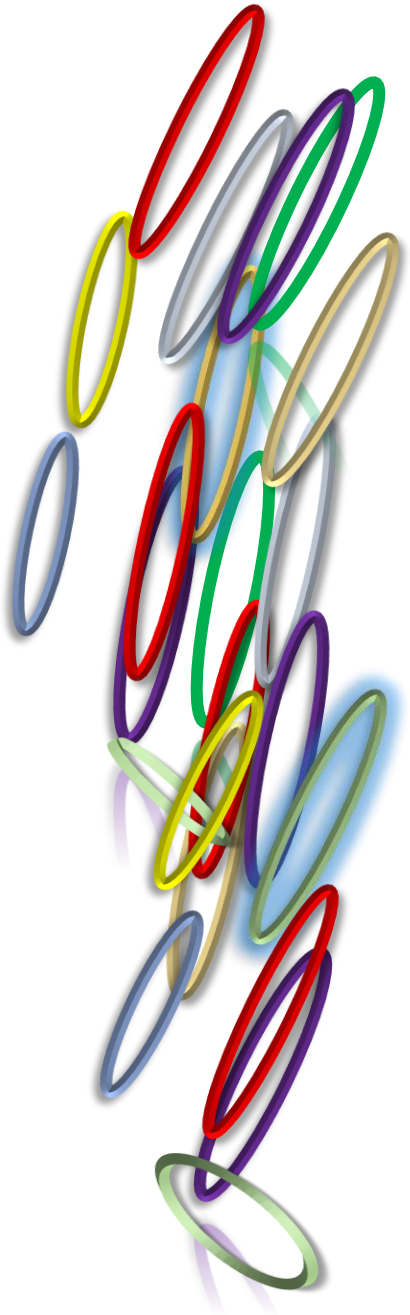


Especificadores de clase de almacenaniento

Clase de almacenamiento extern

static int contador1=0;

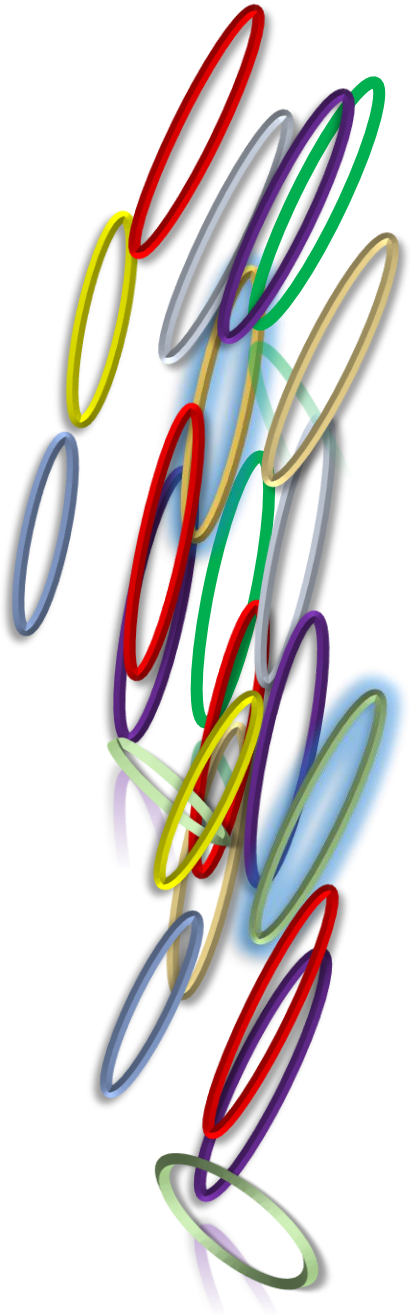
La variable declarada de esta manera **sólo es conocida dentro de la función en la que se la define**. Dentro de una función se iniciará a 1 la primera vez que se ingrese a la función, luego **llamada tras llamada conservará su valor al salir de la función.**



Alcance

Los 4 alcances de los identificadores:

- **Alcance de función**
- **Alcance de archivo**
- **Alcance de bloque**
- **Alcance de prototipo de función**
- **Alcance de clase (POO)**

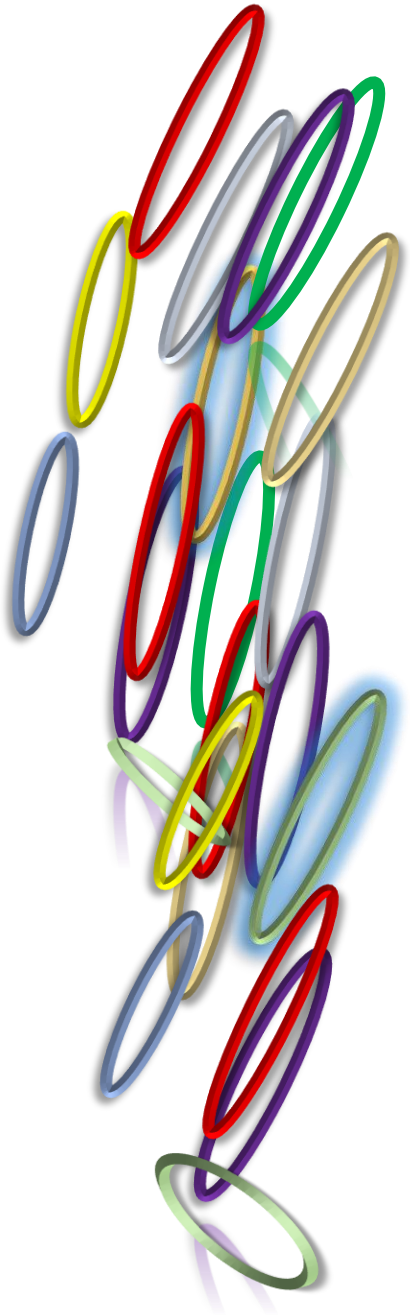


Alcance o visibilidad

El alcance de un objeto es su accesibilidad a otras partes del programa. Un objeto tiene uno de los 4 tipos de alcance.

Los 4 alcances de los identificadores:

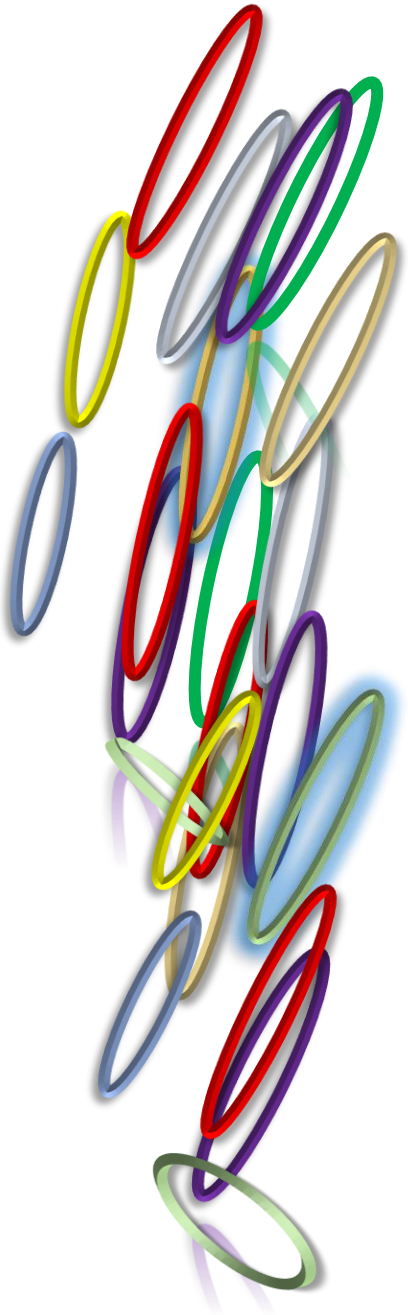
- **Alcance de función.**
- **Alcance de archivo (identificadores declarados fuera de toda función.**
- **Alcance de bloque.**
- **Alcance de prototipo de función (los identificadores en la lista de parámetros de un prototipo de función).**
- **Alcance de clase (POO).**



Lenguaje C y manejo de memoria. Persistencia

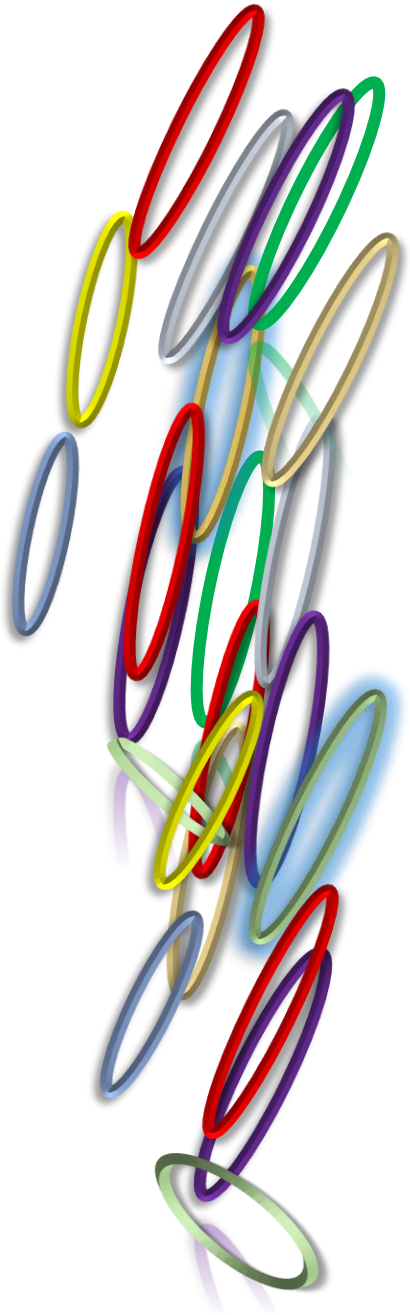
Todos los objetos tienen: tiempo de vida. Es el tiempo durante el cual se garantiza que el objeto exista. 3 tipos de duración: estática, automática y asignada.

- **Variables globales y locales** declaradas con el especificador **static tienen duración estática**. Se crean antes de que el programa inicie su ejecución y se destruyen cuando el programa finaliza.
- **Variables locales no static** tienen duración **automática**. Se crean al entrar al bloque en el que son declaradas y se destruyen al salir de ese bloque.
- **Duración asignada**: objetos cuya memoria se reserva dinámicamente. Esta memoria se **reserva y se libera de forma explícita** mediante invocación a funciones



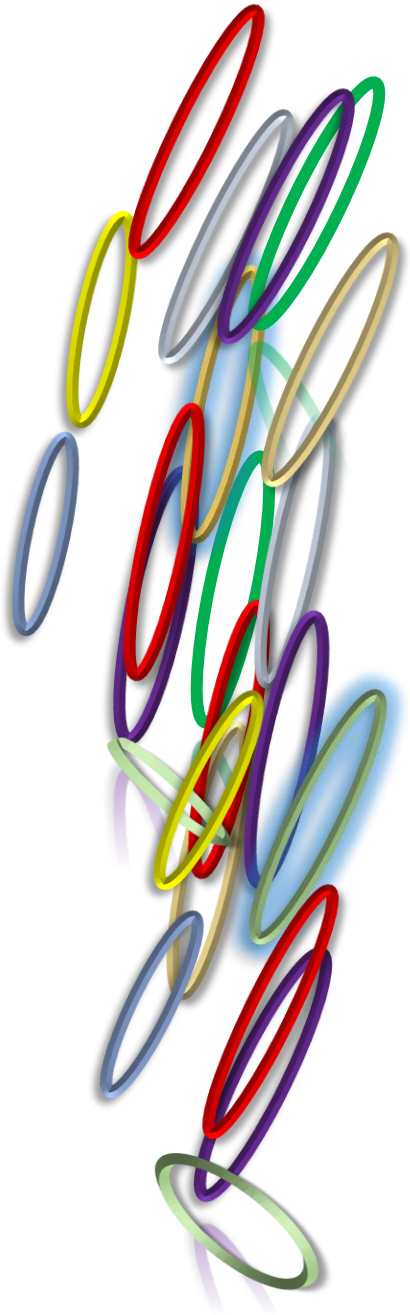
Calificador volatile y const

Son calificadores de tipo. Son independientes.



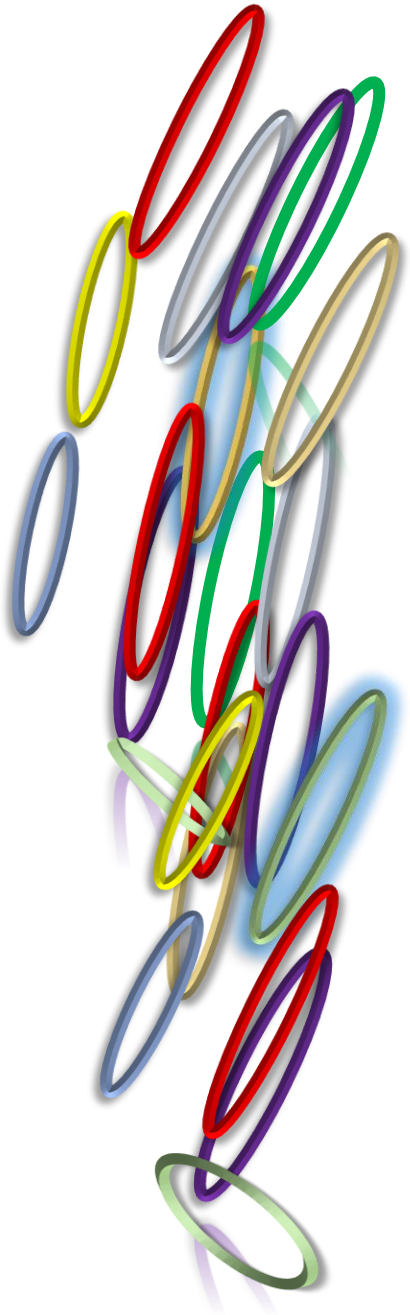
Calificador volatile

- Fue inventado por el estándar ASCII C (X3J11) en el año 1985.
- Se utiliza en código que manipula el hardware: sistemas embebidos y en los controladores de dispositivos.
- volatile aplicable a variables, significa que el objeto puede estar sujeto a cambios repentinos desde fuera del control del programa.
- La palabra reservada volatile notifica al compilador que el objeto al cual es aplicado no debe optimizarse para las operaciones de carga y siempre debe recuperarse de la memoria principal en lugar de hacerlo de los registro o caché.



Calificador volatile

- El objeto puede ser accedido desde una interrupción o señal externa. Se debe acceder al valor cambiado desde la ram, el valor en caché no será válido.
- Poco usado.
- Para programación a bajo nivel.
- El atributo volatile solo realiza determinadas advertencias al compilador.
- Mantiene inalterable el tipo de variable sobre el que se aplica.
- Se aplica a punteros.
- Puede usarse con miembros de clases (propiedades y métodos).



Calificador volatile

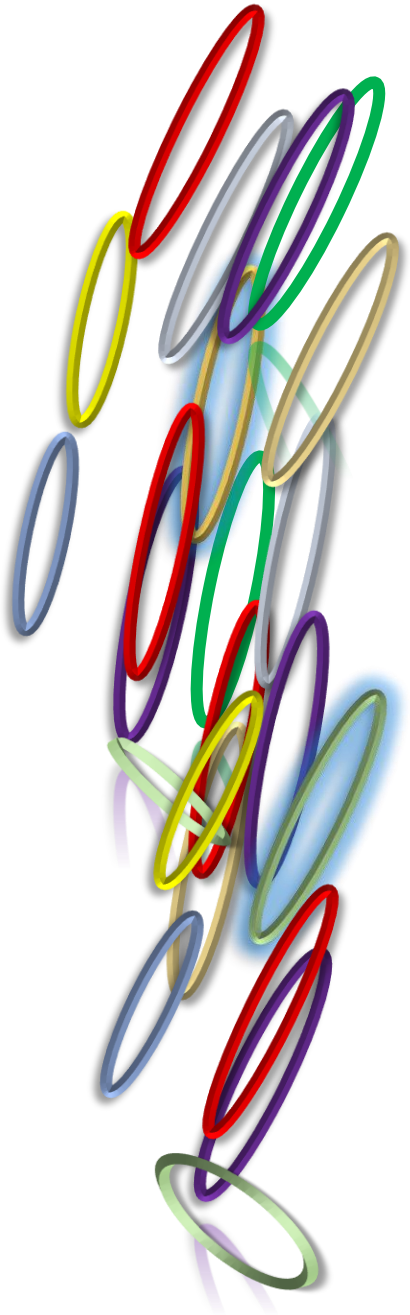
Sintaxis:

volatile tipo_dato identificador;
volatile tipo_dato identificador = valor;

Por ejemplo:
volatile int dato;
volatile char info1= ´A`;

Con punteros:
volatile int dato =2;
int *ptro_dato = & dato; // error ptro_doto no se declaro como volatile

volatile int * ptro_dato = & dato; // OK



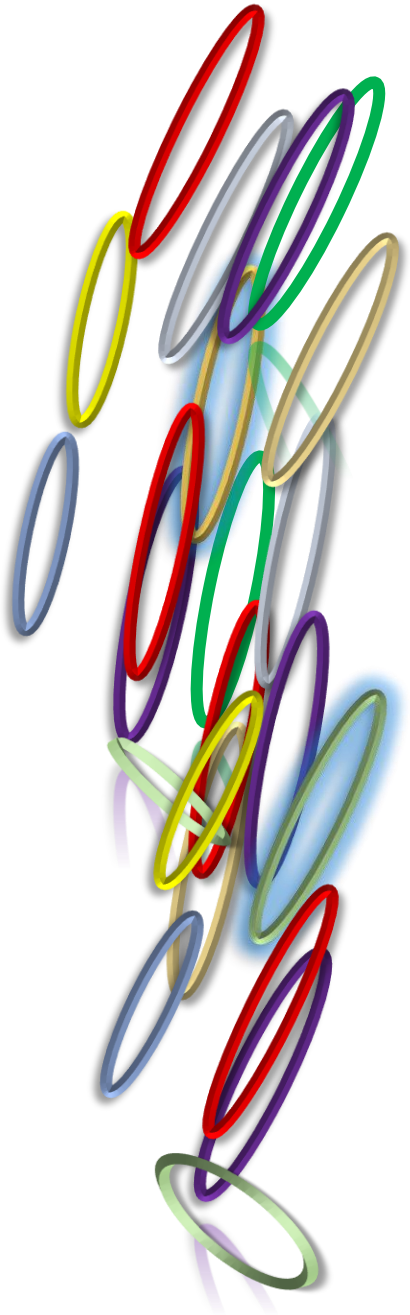
Calificador volatile

Ejemplo: Asumiremos que una placa de adquisición de datos proporciona el estado de una señal en una posición de memoria conocida. El valor depositado, es un unsigned short que en sus bit menos significativos, contiene el estado lógico de 8 sensores de tipo ON/OFF, donde ON =1 y OFF=0. Queremos disparar un proceso si los bit 4 y 6 se activan al mismo tiempo.

Discusión:

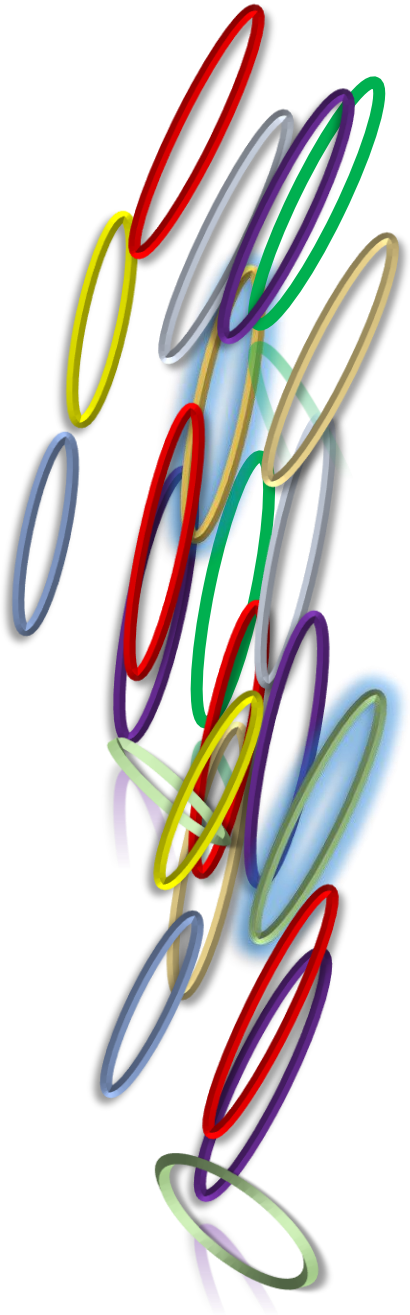
Unsigned short tiene el formato 0xxx xxxx xxxx xxxx, nos interesa disparar un proceso cuando adopte el formato 0xxx xxxx xx1x 1xxx

Sensor 4, formato unsigned short (8): 0000 0000 0000 1000
Sensor 6, formato unsigned short (32): 0000 0000 0010 0000



Calificador volatile

```
.......
volatile unsigned short* ptro = valor;
*ptro = 0;
 unsigned short bit4 = 8, bit6 = 32;

while( 1 ) {              // bucle verdadero
  if ( (*ptro & bit4) && (*ptro & bit6) ) { llama_proceso() }
}
..............
```



Calificador volatile

Variación: activación por bit4 ó bit6 se compone el decimal 40 → 0000 0000 0010 1000

…….

```
volatile unsigned short* ptro = valor;
*ptro = 0;
 unsigned short bit4_6 = 40;

while( 1 ) {              // bucle verdadero
  if ( *ptro & bit4_6 ) { llama_proceso() }
}
```

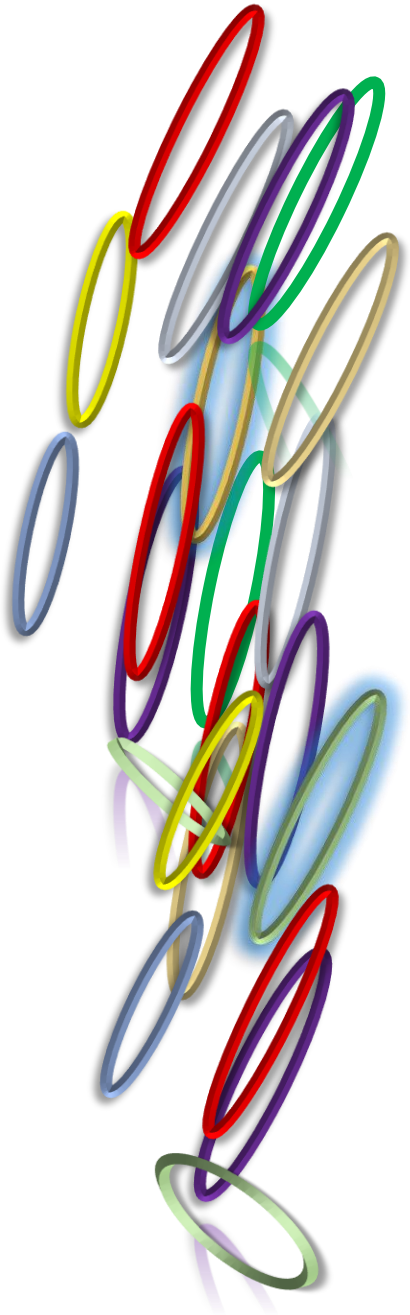
……..

Calificador volatile

El calificador const puede coexistir con volatile, parece un contrasentido…

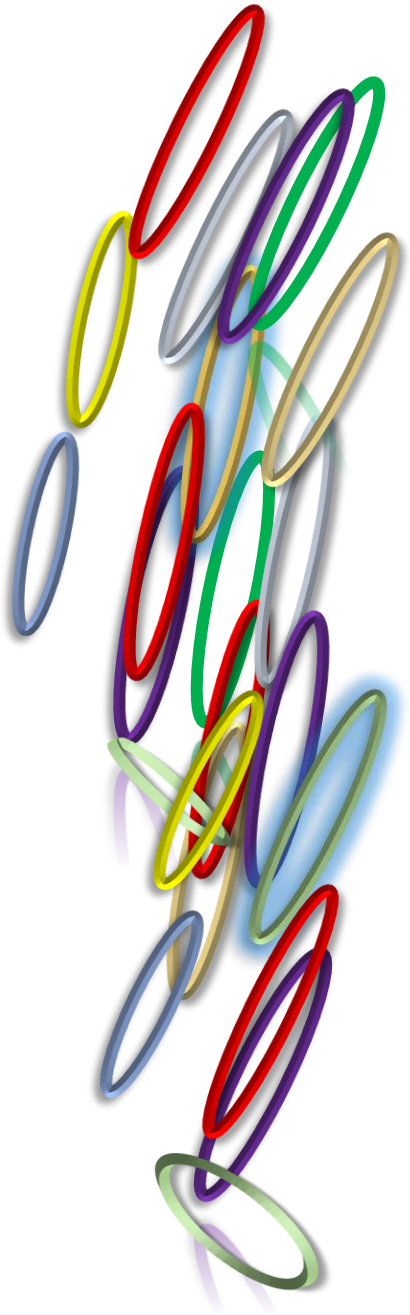
Por ejemplo: const volatile int dato=10;

Discusión de la línea de código: la variable dato no puede cambiar su contenido 10 a lo largo de todo el programa, pero puede aceptar modificaciones procedentes del exterior (causas externas), por lo tanto no puede suponerse su valor actual (se carga desde memoria).



Calificador const

- Palabra reservada
- Aplicado a un objeto, hace que permanezca constante (ram de sólo lectura), no puede modificarse.
- El especificador const se puede aplicar a cualquier objeto de cualquier tipo (variables, funciones, punteros, parámetros de función, miembros de clase..) produciendo un ”nuevo tipo” con idénticas propiedades que el original, pero que no puede modificar su contenido después de su inicialización.
- COMPORTAMIENTO POTENTE, ASUME VARIACIÓN DE SIGNIFICADOS O MATICES, SEGÚN SU USO.



Calificador const

Existen 4 sintaxis posibles:

a) const tipo-de-variable identificador = valor
const int dato =4;

b) tipo Identificador_funcion (**const** tipo identificador_var1);
float MMM (const int a, float b);

c) identificador_metodo **const**;
void print ( ) const;

d) const instancia_clase; (POO)

Calificador const

Existen 4 sintaxis posibles:

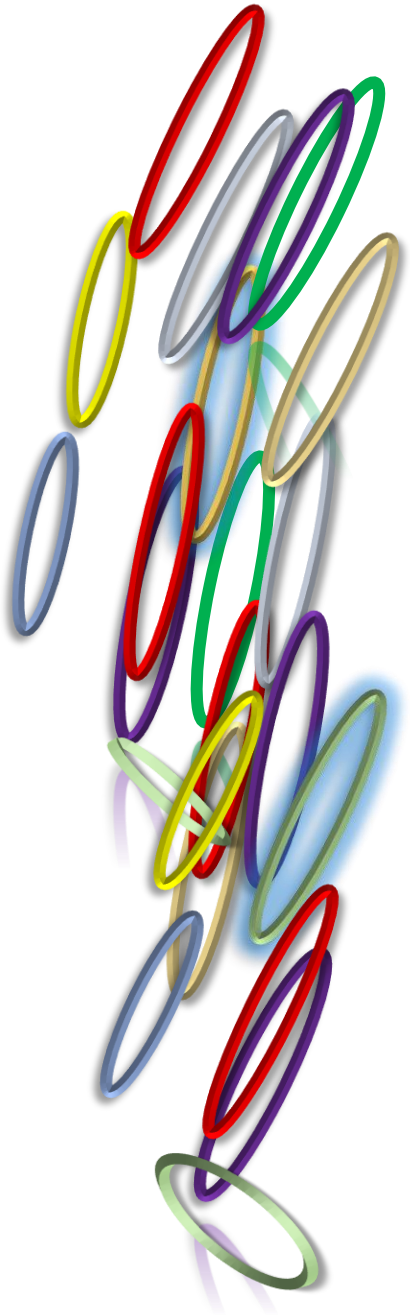
const tipo-de-variable identificador = valor
Identificador_funcion (**const** tipo identificador_var1);
identificador_metodo **const**;
const instancia_clase;

const tipo identificador_función (...);

¡Implicaría que una función devuelve un valor constante!

Calificador const

La asignación a un objeto const debe hacerse en la declaración, a excepción de que se trate de objetos extern



Ejemplos

```
const float num_pi = 3.14;      // una constante float

char * const palabra = “Mundo";   // pt1 puntero constante a tipo char

char const * mot = “gracias";// mot puntero a cadena char constante

const char *termino = “salud";

const peso = 45; // una constante int

const float peso;   // Error!! no inicializada

extern const int x; // Ok. Declarada extern
```

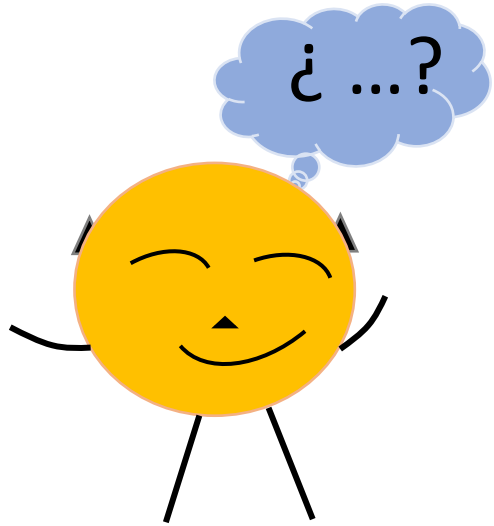
Ejemplos con punteros

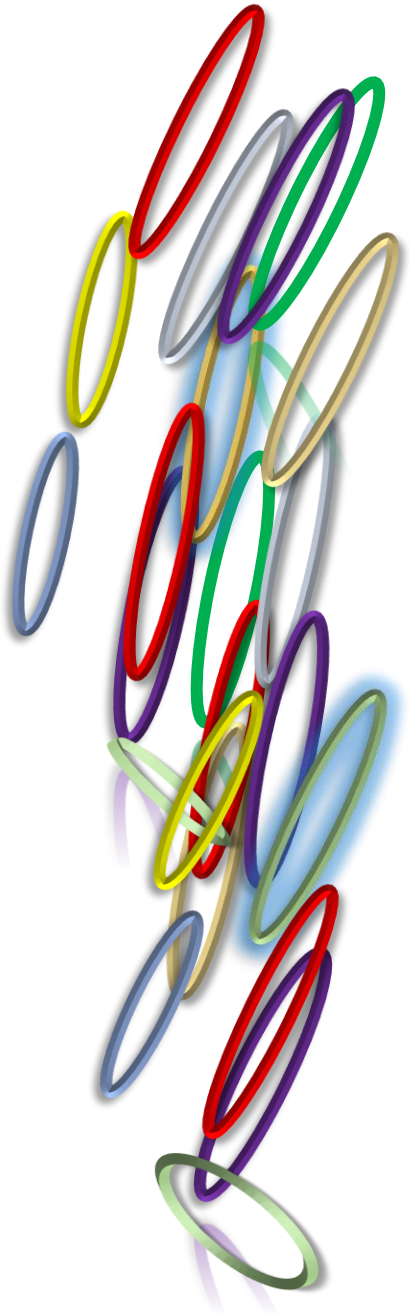
```
int dato1 = 10;
int* ptr_1 = &x;            // Ok
cons int  info = 10;
int* ptr_2 = &info;         // Error
const int* ptr_3 = &info;   // Ok
```

No se produce error si hacemos:

```
int ban = 10;
const int* ptr_ban = &y;     // Ok
```

¿Qué observamos?

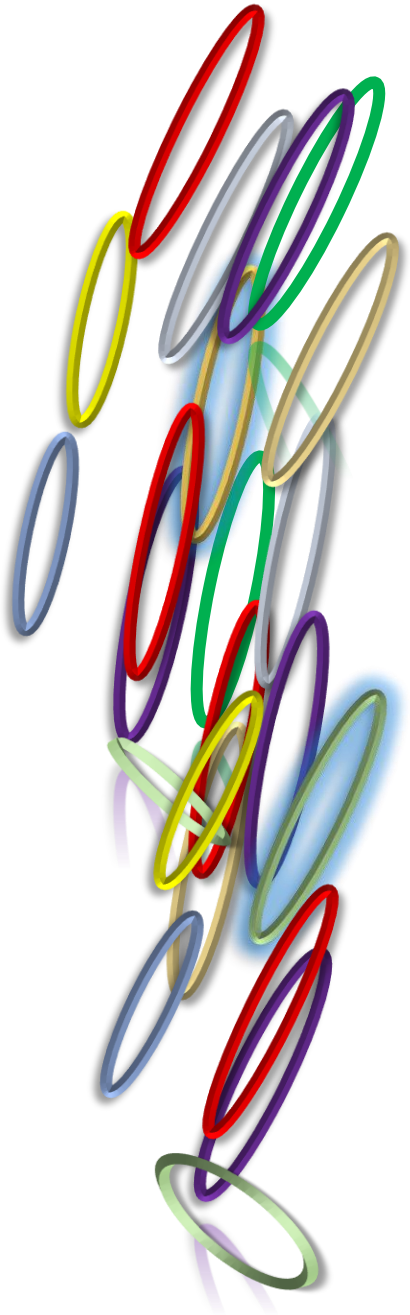




¿Qué observamos de la diapo anterior?

Un puntero declarado como constante no puede ser modificado, pero el objeto al que señala sí puede modificarse. Por ejemplo:

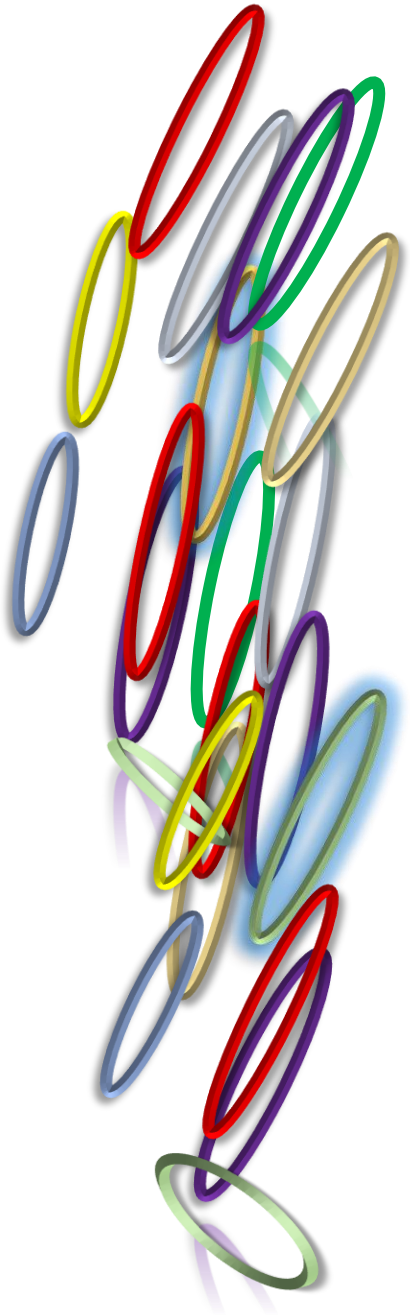
```
char *const ptro1 = “Mundo"; // ptro1 puntero constante-a-char

char *ptro2 = ptro1;// ptro2 señala a la cadena “Mundo"

ptro2++;                    // ptro2 señala a la cadena “undo"

*ptro2 = 'a';                 // modifica segundo carácter de “Mundo"

cout << ptro1;    //muestra Mando
```



Const en parámetros de funciones

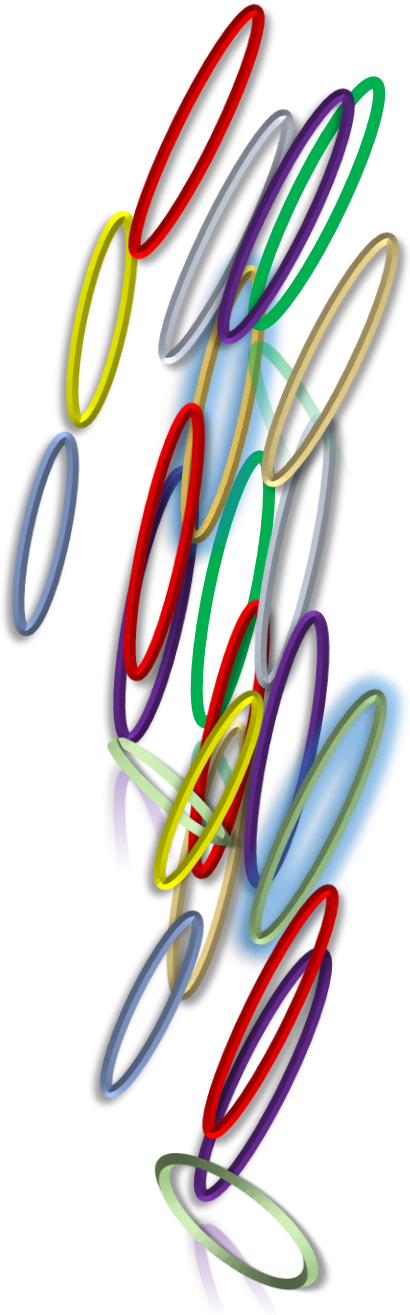
const puede emplearse en la definición de parámetros de funciones.
La función no puede cambiar los argumentos const

```
int MMM (const char *palabra, ...);
```



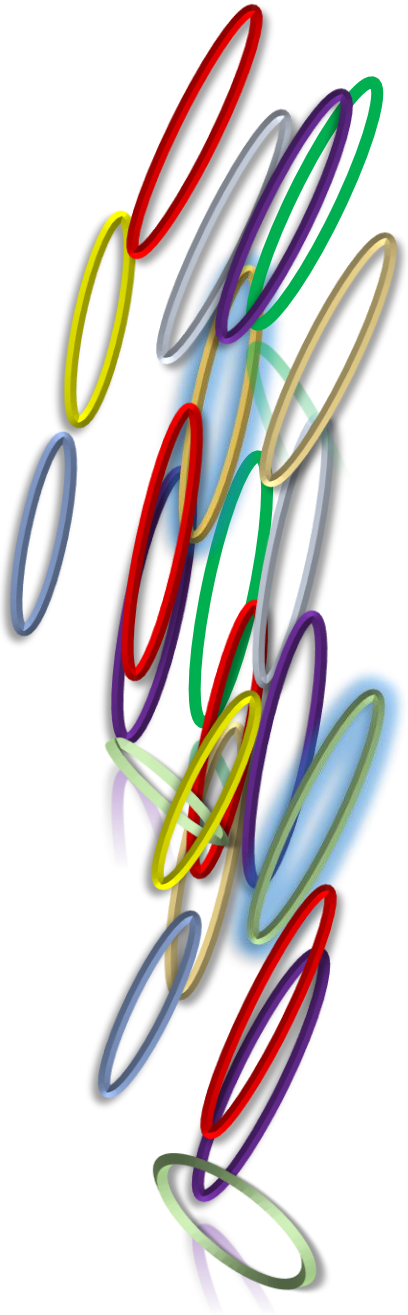
Conjeture sobre los significados de los const del código:

```
int  *ptroN; // puntero a entero
int  *const ptro1; //puntero constante a dato variable

const int * ptro2; //puntero a una constante entera

const char  * const ptro= “pero”; //puntero constante a dato
                                  //constante
```

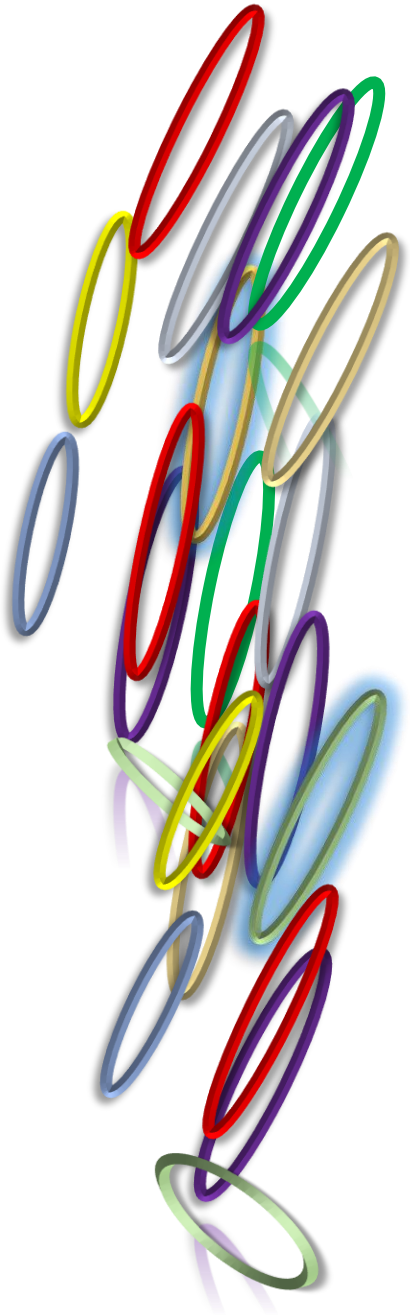


¡Ahora trabaja usted!

Proponer una situación práctica que se resuelva utilizando al menos dos palabras reservadas const con distinta funcionalidad



Gracias



Bibliografía

Deitel, H.M. y Deitel, P.J. (2010). *C++ cómo programar*. Prentice Hall.