Asignatura obligatoria Primer cuatrimestre Créditos ECTS: 6 [Moodle de la asignatura](https://www.aulavirtual.urjc.es/) [Guía docente](http://miportal.urjc.es/guiasdocentes/)

Procesadores Gráﬁcos Avanzados

David Concha Gómez Raúl Cabido Valladolid

# Motivación

* *CUDA comes with a software environment that allows developers to use C++ as a high-level programming language* [[CUDA C Programming Guide]](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

## Si ya sabemos C++... ¿por qué revisamos C entonces?

* + Rutinas de gestión de memoria en CUDA mantienen estilo C
  + Paso de parámetros de algunas rutinas también mantienen estilo C
  + Pequeñas diferencias con respecto a C++ que conviene revisar

## Aprovechamos para repasar punteros

* + CUDA hace uso intensivo de punteros y gestión de memoria dinámica
  + Re-visitamos conceptos vistos en PA
    - aseguramos que todo partimos del mismo punto

cudaError\_t cudaMalloc ( void\*\* devPtr, size\_t size ) cudaError\_t cudaFree ( void\* devPtr )

cudaError\_t cudaMemcpy ( void\* dst, const void\* src, size\_t count, cudaMemcpyKind kind )

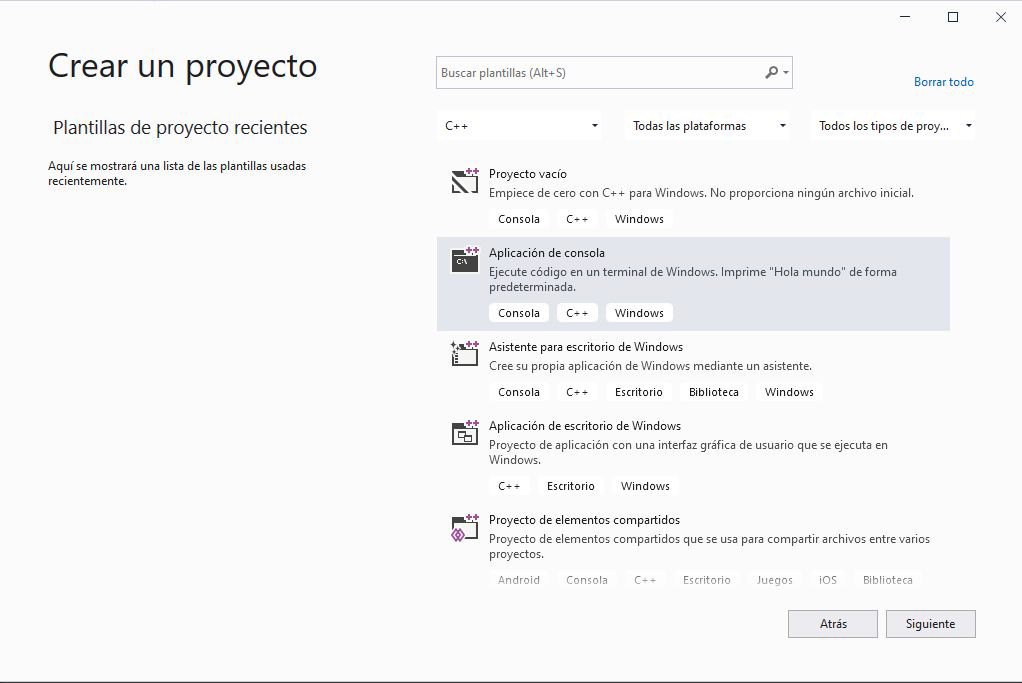
* Punteros (repaso rápido)
* Gestión de memoria dinámica
* Paso de argumentos
* Codelabs

C

¿Qué necesito saber antes de CUDA?



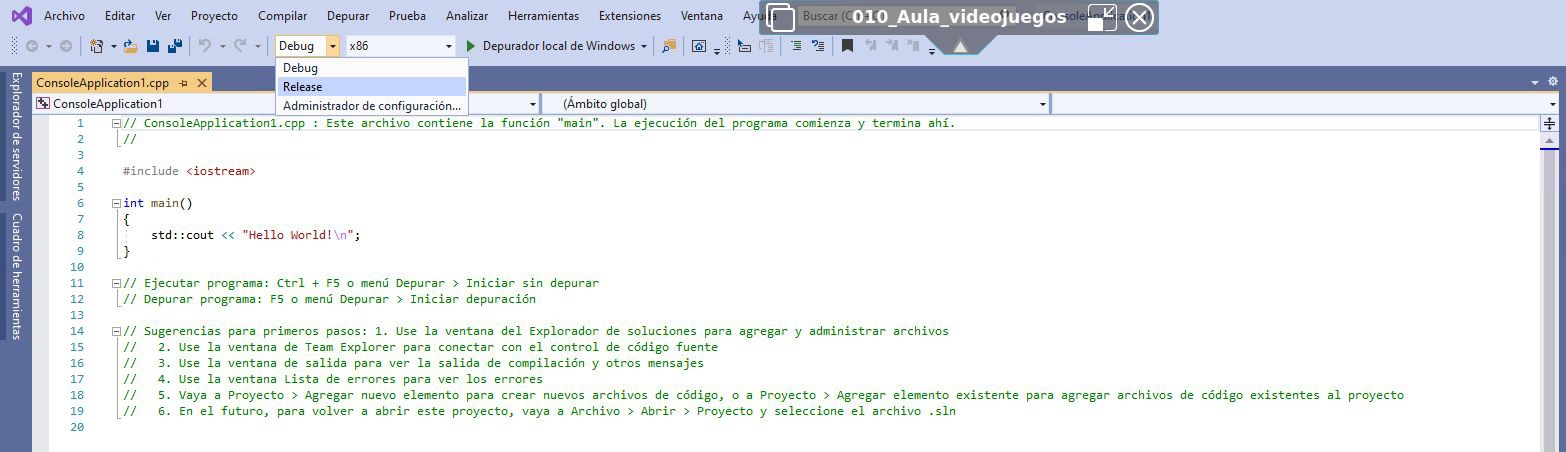
## Visual Studio 2019



Recuerda que el proyecto ha de estar vacío. Desmarca las casillas de opciones adicionales tales

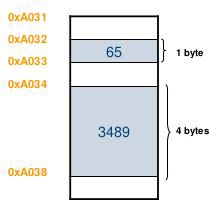
como “Encabezados precompilados”

* Modos de compilación: debug vs. release
  + *Debug* se crea especialmente para ser usado durante el desarrollo de una aplicación
    - El compilador crea un binario más pesado
    - Incluye ciertas instrucciones y símbolos para ayudar durante la depuración
  + *Release*
    - no introduce información de depuración
    - activa opciones de [optimización de código](https://docs.microsoft.com/en-us/cpp/build/optimization-best-practices?view=vs-2019)
      * mejor rendimiento / binario menos pesado



# Punteros

## Punteros, deﬁnición y operadores

* + Concepto de variable
    - Región en memoria reservada para almacenar un dato de un tipo determinado
      * Posición de memoria inicial
      * Tamaño en bytes

char letra = 'A';

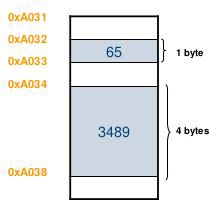
unsigned int entero = 3489;

# Punteros

printf incluido en el módulo <stdio.h>

## Punteros, deﬁnición y operadores

* + Operador de dirección (&)
    - Permite consultar la dirección de memoria inicial a partir de la cual se encuentra almacenada una variable



char letra = 'A';

unsigned int entero = 3849;

...

printf("%p", &letra);0xA032 printf("%p", &entero);0xA034-0xA038

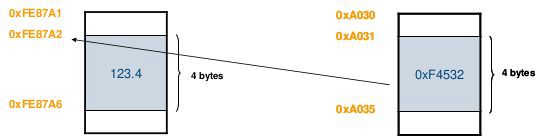
## Punteros, deﬁnición y operadores

* + Declaración

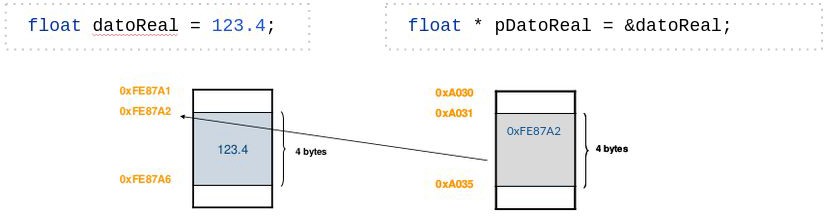
tipo \* identificador; (int\* num)

float datoReal = 123.4;

float \* pDatoReal = &datoReal;



0xFE87A2



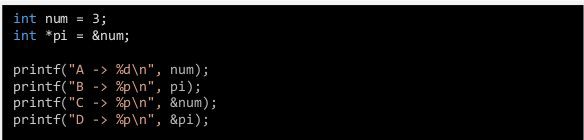
## Punteros, deﬁnición y operadores

* + Operador de indirección \*
    - Permite acceder al contenido de la dirección de memoria almacenada por el puntero.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| float datoReal= 123.4;  float\* pDatoReal ;  pDatoReal =&datoReal;// | | | | |
| printf("Contenido | de | la | dir. almacenada en | pDatoReal: %f(define el tipo) \n", |
| \*pDatoReal ); printf("Contenido | de | la | variable datoReal: | %f \n", datoReal); |
| \*pDatoReal =10.0; printf("Contenido | de | la | dir. almacenada en | pDatoReal: %f \n", |
| \*pDatoReal ); |  |  |  |  |

# Codelab

## Dado el siguiente código



* ¿Qué imprimen esas instrucciones? 3,&num,&num,&pi
* ¿Por qué B y C imprimen el mismo valor? Porque el puntero pi apunta a la dirrec de num
* Crea una sentencia equivalente al primer printf pero usando el puntero pi
  + \*pi

# Punteros

C++ -> nullptr

## Punteros, deﬁnición y operadores

* + Inicialización
    - Acceso a punteros no inicializados
      * Errores en tiempo de ejecución
      * Comportamientos anómalos
    - Solución: inicialización del puntero a una dirección de memoria nula
      * NULL constante entera deﬁnida en stdio.h, stedf.h, stdlib.h, string.h

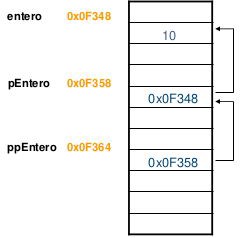
#include <stdio.h>

...

float\* pDatoReal = **NULL**;

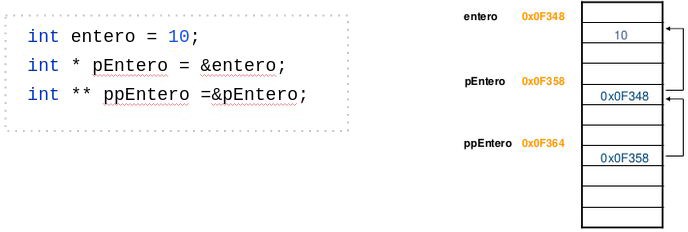
## Puntero a puntero

* + El tipo base de un puntero puede ser cualquier tipo de dato, entre ellos un puntero



int entero = 10;

int \* pEntero = &entero; int \*\* ppEntero =&pEntero;



## Puntero a puntero

int entero=10;

int \* pEntero = &entero; int \*\* ppEntero = &pEntero;

\*\*ppEntero =9;

printf("Dir. almacenada en ppEntero: %p \n", ppEntero); printf("Dir. almacenada en pEntero: %p \n", pEntero);*// o \*ppEntero* printf("Valor almacenado en entero: %d \n", entero);*// o \*pEntero o*

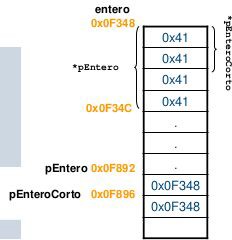
*\*\*ppEntero*

* Es posible llevar a cabo asignaciones entre punteros con distinto tipo base
  + Conversiones implícitas
  + Conversiones explícitas
* Estas asignaciones pueden hacer que el tipo base de un puntero no coincida con el tipo del valor almacenado en la dirección indicada por el puntero
* El valor devuelto al desreferenciar un puntero dependerá del tipo base establecido en su declaración
* Conversión implícita

int entero= 1094795585 ;*//0x41414141* int \* pEntero=&entero;

short\* pEnteroCorto = pEntero;*//es posible?*

printf("%p == %p" ,pEntero,pEnteroCorto ); printf("%d != %d" ,\*pEntero,\*pEnteroCorto );

* Conversión explícita 

short\* pEnteroCorto = (short\*)pEntero;

* Conversión entre punteros
  + C permite deﬁnir punteros genéricos, sin un tipo base asociado

void \* identificador ;

* Los punteros genéricos pueden inicializarse con el valor de otra variable puntero independientemente de cual sea su tipo base
* Al desreferenciar un puntero genérico es necesario indicar de forma explícita su tipo base

void \* identificador;

# Punteros

## Conversión entre punteros

* + C permite deﬁnir punteros genéricos, sin un tipo base asociado

int entero = 1094795585; int\* pEntero = &entero; **void\* p** = pEntero;

***//printf("%d\n", \*p); //indirección no válida!***

printf("%d\n", \***(int\*)**p);se necesita hacer un casting

\***(int\*)**p = 1;

printf("%p == %p == %p\n", &entero, pEntero, p); printf("%d == %d == %d", entero, \*pEntero, **\*(int\*)**p);

## Punteros a constantes y punteros constantes

* + const permitía deﬁnir variables constantes
  + Su uso con punteros deﬁne 3 comportamientos distintos según su posición:
    - Puntero a constante
      * const tipo \* identificador;
    - Puntero constante
      * tipo \* const identificador;
    - Puntero constante a constante
      * const tipo \* const identificador;

const int \* pValor = &valor1;

pValor = &valor2;

\*pValor = 3; *// No permite modificar el contenido*

puntero a constante

int \* const pValor =&valor1;

\*pValor = 3;

pValor = &valor2; *// No permite modificar la dirección de memoria*

puntero constante

const int \* const pValor =&valor1;

\*pValor = 3;*//No permite modificar el contenido*

pValor = &valor2; *// No permite modificar la dirección de memoria*

puntero constante a constante

# Punteros

Ejercicios **útiles a la hora de coger soltura** con punteros y operadores...pero no ayudan a entender necesidad de punteros

int entero1, entero2, entero3; int \*ptr1, \*ptr2;

void \* ptrGenerico ;

ptr1=&entero1;

\*ptr1=1; ptr2=&entero2;

\*ptr2=2; ptr1=ptr2;

\*ptr1=0;

...

ptr2=&entero3;

\*ptr2=3;

printf("%d %d %d \n",

entero1, entero2, entero3); ptrGenerico =&ptr1;

\*(int\*\*)ptrGenerico =ptr2;

\*ptr1=1;

printf("%d %d %d \n",

entero1,entero2, entero3);

## Es posible realizar operaciones aritméticas con variables de tipo puntero

* + Operaciones de incremento y decremento

## Permiten desplazamientos a posiciones de memoria anteriores y posteriores a la almacenada por un puntero

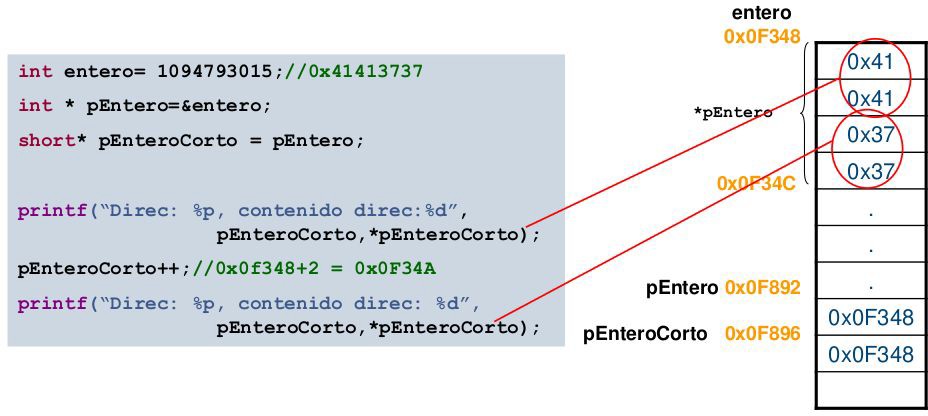
* El incremento o decremento sobre el valor actual del puntero dependerá del tipo base de este

tipoBase \* puntero;

puntero = puntero+i; *//puntero= direcPuntero+sizeof(tipoBase)\*i*

puntero = puntero-i; *//puntero= direcPuntero-sizeof(tipoBase)\*i*

|  |  |  |
| --- | --- | --- |
| Operación | Resultado | Comentario |
| ptr++ | puntero | Desplazamiento ascendente de un elemento |
| ptr-- | puntero | Desplazamiento descendente de un elemento |
| ptr+i | puntero | Desplazamiento ascendente de i elementos |
| ptr-i | puntero | Desplazamiento descendente de i elementos |
| ptr1-ptr2 | entero | Distancia entre elementos |



**0x0F34A**

# Codelab

## Utiliza el operador sizeof para conocer el tamaño en bytes de los siguientes tipos de datos en C

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Tipo | char | short | int | long | long long | float | double |
| # bytes |  |  |  |  |  |  |  |

* Calcula también el tamaño de su versión sin signo: unsigned char, unsigned short, unsigned int…
* Calcula también el tamaño de su versión puntero: char\*, short\*, int\*,...
* Preguntas
  + ¿quién/qué ﬁja el tamaño de los tipos? C++ standard
  + ¿pueden variar de una máquina a otra? Si se puede variar con diferentes compiladores
  + ¿de qué depende? La arquitectura, el sistema operativo, el compilador

<https://en.cppreference.com/w/cpp/language/types>

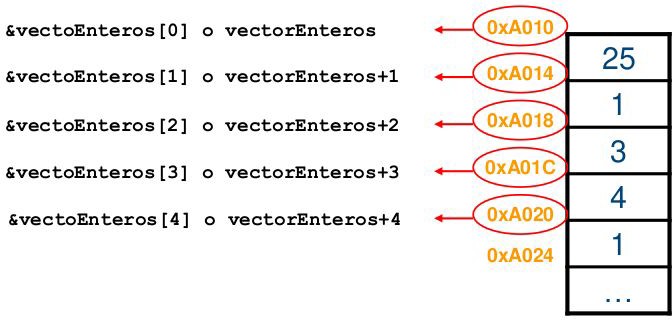
* El identiﬁcador de un array puede verse como un puntero
* Su valor se corresponde con la dirección de memoria del primer elemento

int vectorEnteros [10];

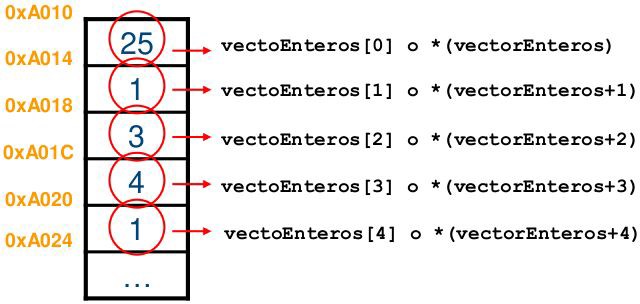
*//&vectorEnteros[0] == vectorEnteros*

* Este tipo de relaciones entre arrays y punteros dan lugar a una nueva forma de acceder a los elementos de un array
  + Usando aritmética de punteros

int vectorEnteros []={25,1,3,4,1};



int vectorEnteros []={25,1,3,4,1};



const size\_t size =10; int vector [size];

for (size\_t i=0; i<size; i++){ vector[i]=0;

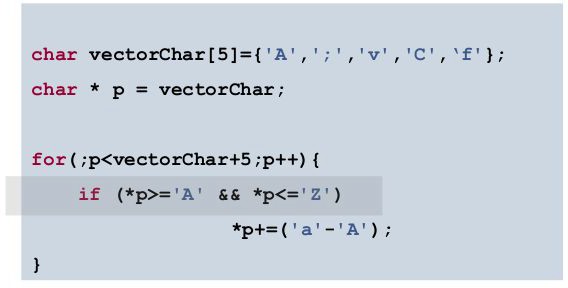
}

const size\_t size =10; int vector [size];

for (size\_t i=0; i<size; i++){

\*(vector+i)=0;

}



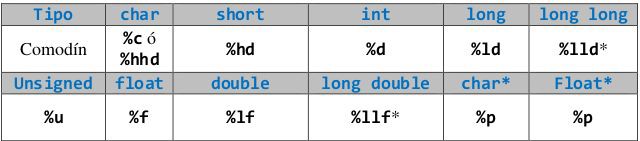
* Crea un array a de 10 elementos (tipo double).
* Crea un puntero pa del mismo tipo de dato y que apunte al inicio de dicho array.
* Rellena una tabla similar a la siguiente

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| &(a[0]) | pa+0 | &(a[1]) | pa+1 | &(a[2]) | pa+2 | &(a[3]) | pa+3 | ... |
|  |  |  |  | ... |

* Crea ahora un array s de 10 elementos (tipo short).
* Crea un puntero ps que apunte al inicio de dicho array
  + completa de nuevo la tabla
  + ¿se sigue cumpliendo que el puntero avanza por los elementos del array?

## Conclusión:

* + cuando avanzo un puntero una posición, ¿cuántos bytes recorre?



para imprimir una dirección usa el especiﬁcador de formato descrito en la tabla

[doc printf](http://www.cplusplus.com/reference/cstdio/printf/)

## La cantidad de memoria que ocupaban los datos a manejar por un programa se conoce en tiempo de compilación

* Esto no siempre es así…
  + Puede que esta información se desconozca hasta el momento de la ejecución
  + Surge la necesidad de algún tipo de mecanismo que nos permita reservar memoria en tiempo de ejecución -> **memoria dinámica**

## En C distinguimos [tres tipos](https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-and-C.html) de reserva de memoria

* + reserva estática
  + reserva automática
  + **reserva dinámica**

primer caso de uso interesante de punteros! -> manejo de memoria dinámica

## Gestión dinámica de la memoria en C

* + El módulo stdlib.h deﬁne un conjunto de rutinas que permiten la reserva y liberación de memoria en tiempo de ejecución
    - malloc

Operadores equivalentes en C++: new/delete

* + - free

int \* pEnteros = new int [nElementos];

...

delete [] pEnteros;

* + - …
  + ¿Cómo manejamos la memoria reservada?
    - punteros

Objetivo: prepararnos para lo que viene...

cudaError\_t cudaMalloc ( void\*\* devPtr, size\_t size ) cudaError\_t cudaFree ( void\* devPtr )

cudaError\_t cudaMemcpy ( void\* dst, const void\* src, size\_t count, cudaMemcpyKind kind )

## Gestión dinámica de la memoria en C

* + malloc
    - Función que reserva un bloque de memoria contiguo dentro del montón
      * void \* malloc (size\_t)
    - El número de bytes que ocupa el bloque se pasa como parámetro
    - Si no es posible hacer la reserva devuelve un valor NULL
    - Si hay éxito, devuelve la dirección de memoria donde comienza el bloque

## Gestión dinámica de la memoria en C

* + free
    - Función que libera un bloque de memoria previamente reservado de forma dinámica
      * void free (void \*)
    - Es responsabilidad del programador liberar la memoria reservada de forma dinámica una vez que ya no es útil
      * para que esta pueda ser reutilizada en futuras reservas
      * para evitar posibles pérdidas de memoria

## Arrays dinámicos

* + Problema de los arrays estáticos: su tamaño ha de ser conocido en tiempo de compilación
    - int vectorEnteros [10];
  + El uso de punteros y memoria dinámica permiten la deﬁnición de arrays dinámicos.
    - Arrays cuyo tamaño puede ser establecido en tiempo de ejecución

int numElementos ; scanf("%d", &numElementos );

int \* pEnteros= (int\*) malloc ( sizeof(int)\*numElementos );

## Arrays dinámicos

* + Acceso a sus elementos
    - Aritmética de punteros
    - [ ]
  + La memoria reservada para un array dinámico debe ser liberada por el programador cuando ya no sea necesaria

free (pEnteros); pEnteros = **NULL**;

## Otras rutinas…

* + calloc

void \* calloc (size\_t, size\_t)

* + - Permite reservar un bloque de memoria de forma dinámica
      * igual que malloc
    - Recibe dos argumentos
      * número de elementos que se quiere almacenar en el bloque
      * tamaño que ocupa cada uno de los elementos
    - Si es posible realizar la reserva
      * todos los bits que conforman el bloque se ponen a 0
      * se devuelve la dirección de memoria donde comienza el bloque

## Otras rutinas…

* + calloc

void \* realloc (void\*, size\_t)

* + - Intentará cambiar el tamaño de una zona de memoria dinámica reservada previamente con malloc
    - El nuevo tamaño puede ser más grande o mas pequeño
      * Más grande → el contenido inicial permanece sin cambios y se agrega memoria al ﬁnal del bloque
      * Más pequeño → el contenido sobrante permanece sin cambios
    - Si no se puede redimensionar
      * realloc intentará reservar un nuevo bloque de memoria
        + Si hay memoria disponible, copiará el contenido del bloque inicial
        + Si no, devuelve un valor NULL

# Codelab

## Programa en C que calcule la suma de dos vectores de tamaño arbitrario

* + El tamaño de los vectores se ﬁja en tiempo de ejecución
  + Los vectores se inicializan con valores aleatorios (ver [rand](http://www.cplusplus.com/reference/cstdlib/rand/))
  + Imprimir los vectores de entrada y el vector resultante de la suma

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 5 | 7 | 8 | 1 |

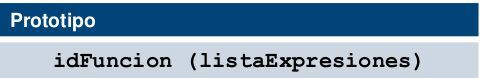
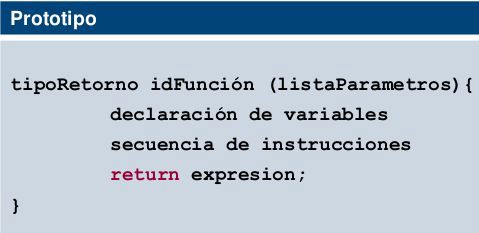
+

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 5 | 10 | 0 | 2 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 3 | 2 | 1 |

## Declaración, deﬁnición y llamada igual que C++

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



double potencia (double base, double exp); int main(){

double base, exp, resultado ;

...

resultado =potencia ( base, exp);

printf( "El resultado de elevar %f a %f es %f" , base, exp, resultado );

...

}

double potencia (double base, double exp){

...

return cont;

}

## Declaración, deﬁnición y llamada igual que C++

* Diferencias en paso de parámetros
  + en C el paso de parámetros es siempre por valor
  + el paso por referencia se consigue mediante el uso de punteros
    - en C++ usamos el tipo de dato referencia

void roots( float a, float b, float c, float\* root1, float\* root2 ) *//C*

void roots( float a, float b, float c, float& root1, float& root2 ) *//C++*

paso por referencia en C → otro caso de uso de punteros!

## Declaración, deﬁnición y llamada igual que C++

* Diferencias en paso de parámetros
  + en C el paso de parámetros es siempre por valor
  + el paso por referencia se consigue mediante el uso de punteros
    - en C++ usamos el tipo de dato referencia

cudaError\_t cudaGetDevice ( int\* device ); *//Returns which device is currently being used.* cudaError\_t cudaGetDeviceCount ( int\* count ); *// Returns the number of compute-capable devices.* cudaError\_t cudaGetDeviceFlags ( unsigned int \* flags ); *//Gets the flags for the current device.*

cudaError\_t cudaGetDeviceProperties ( cudaDeviceProp\* prop, int device ); *// Returns information about the*

*compute-device. ice.*

# Paso por referencia

* ¿consigue add incrementar el valor de i?
  + ¿motivo?
  + ¿solución?
* Dibujar mapa de memoria para el siguiente fragmento de código

void add(int value, int increment)

{

value += increment;

}

int main (){ int i = 3; add(i, 4);

printf( "i: %d\n", i);

}

# Paso por referencia

* en C el paso de parámetros es siempre por valor
* el paso por referencia se consigue mediante el uso de punteros
* Dibujar mapa de memoria para el siguiente fragmento de código

void add(int \*value, int increment)

{

\*value += increment;

}

int main (){ int i = 3; add(& i, 4);

printf( "i: %d\n", i);

}

# Paso de arrays a funciones

* Se pasa por valor un puntero al primer elemento del array (estático o dinámico)
  + Prototipo

int setZeros (int \* vector, int size);

int cumulativeSum (const int \* vector, int size); int print (const int \* vector, int size);

¿por qué setZeros no usa const?

* + Llamada

int\* vector = (int\*) malloc(sizeof(int)\*10); *// int vector [10];*

int err = setZeros(vector, 10);

int sum = cumulativeSum (vector, 10); int err = print (vector, 10);

* Programa en C que calcule la suma de dos vectores de tamaño arbitrario
  + El tamaño de los vectores se ﬁja en tiempo de ejecución
  + Los vectores se inicializan con valores aleatorios (ver [rand](http://www.cplusplus.com/reference/cstdlib/rand/))
  + Imprimir los vectores de entrada y el vector resultante de la suma

void allocate (...);

void init (...);

void print (...); void deallocate (...);

## Modiﬁca el código

* + Añade funciones y sus correspondientes llamadas en el código
    - función allocate/deallocate que permita reservar/liberar memoria para un vector
    - función init que permita inicializar los elementos de un vector con valores aleatorios
    - función print que permita imprimir los elementos de un vector
* Modiﬁca el código
  + Añade funciones y sus correspondientes llamadas en el código
    - función allocate que permite reservar memoria para un vector
    - función init que permita inicializar los elementos de un vector con valores aleatorios
    - función print que permita imprimir los elementos de un vector

## ¿Podríamos hacer una función allocate genérica?

* + válida para cualquier vector → independiente del tipo de dato que almacene

# Vista atrás...

## revisemos las cabeceras que vimos en la motivación…

* + ¿siguen siendo un problema?

cudaError\_t cudaMalloc ( void\*\* devPtr, size\_t size ) cudaError\_t cudaFree ( void\* devPtr )

cudaError\_t cudaMemcpy ( void\* dst, const void\* src, size\_t count,

cudaMemcpyKind kind )

**Recursos**

